Entity Framework Core IN ACTION

SECOND EDITION

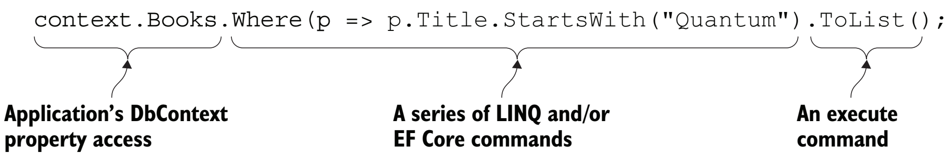
Jon P Smith

Foreword by Julie Lerman

MANNING

The following table provides a list of topics covered in this book, with a list of chapters where that topic is covered, with the primary chapter at the front. It also lists any key figures that go with this topic.

Topics	Chapters	Key figures (chapter.figure)
Setting up EF Core	1, 2, 7, 8, 5	1.4, 2.7, 1.5
Query the database	2, 5, 6	2.7, 2.8, 6.2
Create, Update, Delete	3, 5, 8, 11	3.1, 3.2, 6.8 to 10
How EF Core works inside	1, 6	1.6, 1.8, 1.10, 6.8 to 6.10
Business Logic	4, 5, 13	4.2, 4.4, 5.4
ASP.NET Core	5, 2	5.1, 5.4
Dependency injection	5, 14, 15	5.2, 5.3
Async/await	5, 14	5.8, 5.9
Configure non-relational	7	7.1, 7.2
Configure relationships	8	8.1, 8.2, 8.3, 8.4
Configure table mappings	8	8.12, 8.13
Database migrations	9, 5	9.2, 9.3, 9.5, 9.7, 9.8
Concurrency issues	10, 15	10.3, 10.4, 10.5, 10.6, 10.7, 15.7
Using raw SQL	11, 6	15.3
Domain-Driven design	13, 4	13.3, 4.2, 13.4, 13.5
Performance tuning	14, 15, 16	14.1, 15.2, 15.4, 15.9, 16.7, 16.8
Cosmos DB & other databases	16, 17	16.1, 16.3, 16.4, 17.5
Data Validation	4, 7, 10	
Unit testing	17	17.2, 17.3
LINQ language	Appendix, 2	A.2, A.1



```
context.Books.Where(p => p.Title.StartsWith("Quantum")).ToList();
```

An example of an Entity Framework Core database query

Praise for the first edition

This book helped highlight some issues my team was having with EF Core and has become an invaluable resource that takes pride of place on our bookshelves.

—Evan Wallace, senior software Developer at Berkley Insurance Australia

The most complete go-to book for anything you need to know about EF Core! The #1 must-have EF Core reference for your library!

—Mauro Quercioli, senior independent software architect/developer, Siena I.T. Consulting Corporation

Knowing that EF Core in Action *is right there on my desk, I am approaching my latest assignment—to build out a new WebAPI application using Entity Framework Core—with complete confidence. The book addresses everything we needed to know about EF Core and more!*

—Phil Taffet, senior .NET developer, California Earthquake Authority

Finally a book to learn all about EF Core. It's fun and engaging reading this. Be careful—whether you're a beginner or professional, you'll learn something.

—Raushan Kumar Jha, SE-2, Microsoft India (R&D)

This is a solid book dealing well with the topic in hand but also handling the wider concerns around using Entity Framework in real-world applications.

—Sebastian Rogers, technical director, Simple Innovations

Entity Framework Core in Action

SECOND EDITION

JON P SMITH

FOREWORD BY JULIE LERMAN

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Marina Michaels
Technical development editor: Joel Kotarski
Review editor: Aleksandar Dragosavljević
Production editor: Keri Hales
Copy editor: Keir Simpson
Proofreader: Melody Dolab
Technical proofreader: Julien Pohie
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617298363
Printed in the United States of America

contents

2 *Querying the database 27*

3 *Changing the database content 61*

4 *Using EF Core in business logic 94*

5 Using EF Core in ASP.NET Core web applications 125

6 *Tips and techniques for reading and writing with EF Core 159*

PART 2 189

7 Configuring nonrelational properties 191

8 *Configuring relationships 226*

9 Handling database migrations 268

10 *Configuring advanced features and handling concurrency conflicts 306*

11 Going deeper into the DbContext 340

PART 3 379

12 Using entity events to solve business problems 381

13 Domain-Driven Design and other architectural approaches 405

15 *Master class on performance-tuning database queries 463*

16 *Cosmos DB, CQRS, and other database types 492*

foreword

Have you ever worked on an application that doesn't use data and requires some means of interacting with a data store? In my decades as a software developer, every single application I have worked on or have helped others with has depended on reading and writing to some type of data store. When I became a solo entrepreneur in the 1990s, I came up with the name The Data Farm for my company. I am definitely a data nerd.

Over the past few decades, Microsoft has gone through many iterations of data access frameworks. If you've been working in this arena for a while, you might remember DAO and RDO, ADO, and ADO.NET. In 2006, Microsoft shared the first iterations of the as-yet-unnamed Entity Framework (EF), based on work done at Microsoft Research in a private meeting at TechEd. I was one of the few people invited to that meeting. It was my first time seeing an Object Relational Mapper (ORM), a library that focuses on relieving developers from the redundant drudgery of building connections and commands, writing SQL, transforming query results into objects, and transforming object changes into SQL to persist back to the database.

Many of us worried that this framework was yet another data access framework that Microsoft would give up on in short order, forcing us to learn yet another one down the road. But history has proved us wrong. Fifteen years later, Microsoft is still investing in Entity Framework, which has evolved into the cross-platform and open source Entity Framework Core and continues to be Microsoft's go-to data access library for .NET developers.

Over the 15 years that EF has been around and evolving, .NET has evolved as well. EF and EF Core have grown in capability, but at the same time, the library has become smarter about getting out of the developer's way when it comes to building modern software systems. We can customize mappings to support persistence with intricate database schema. As a Domain-Driven Design practitioner, I have been extremely happy with the attention that the team has paid to allowing EF Core to persist carefully designed entities, value objects, and aggregates that, by design, are not burdened with knowledge of database schema.

As an early adopter who worked closely with the EF team even before the initial release, I wrote four books on Entity Framework between 2008 and 2011. Though I do love to write, I eventually discovered that I also love creating videos, so I focused my own efforts on creating and publishing courses on EF Core and other topics as a Pluralsight author. I still write articles, but no more books, which is why I'm so happy that Jon P Smith found his way to writing for Manning.

When Jon published the first edition of *Entity Framework Core in Action*, I recognized in him a fellow "curious cat" who left no stone unturned in his quest to understand how EF Core works. Equally, he takes the role of sharing that information seriously, ensuring that his readers are able to follow along and truly gain expertise. His work is meticulous, and his knowledge is deep. As I have continued creating training resources for those who prefer to learn from videos, I've been pleased to recommend Jon's work to those who are looking for a trustworthy book on EF Core. Updating content to reflect the newest version, EF Core 5, is no small task. Jon has once again earned my respect (and the respect of many other people) with the edition you now have in hand.

With this book, you're really getting three books in one. First, Jon handholds you through the basics, even building some simple applications that use EF Core. When you've become comfortable, it's time to dig more deeply into intermediate-level use, with relationships, migrations, and control beyond EF Core's default behavior. Finally, it's time to use EF Core in real-world applications, tackling important topics such as performance and architecture. Jon's meticulous exploration and his own experience working with large software applications make him a skilled and trustworthy guide.

—JULIE LERMAN

Julie Lerman is known as the foremost expert on Entity Framework and EF Core outside Microsoft. She is the author of the Programming Entity Framework *book series and dozens of courses on Pluralsight.com. Julie coaches businesses on modernizing their software. You can find her presenting and keynoting on EF, Domain-Driven Design, and other topics at software conferences around the world.*

preface

Any software developer should be used to having to learn new libraries or languages, but for me, the learning has been a bit extreme. I stopped coding in 1988, when I went into technical management, and I didn't come back to coding until 2009—a 21-year gap. To say that the landscape had changed is an understatement; I felt like a child on Christmas morning with so many lovely presents, I couldn't take them all in.

I made all the rookie mistakes at the beginning, such as thinking that object-oriented programming is about using inheritance, which it isn't. But I learned the new syntax and new tools (wow!), and reveled in the amount of information I could get online. I chose to focus on Microsoft's stack, mainly because of the wealth of documentation available. That was a good choice at the time, but with the .NET Core initiative with its open source, multiplatform approach, it turns out to be an excellent choice.

The first applications I worked on in 2009 were ones that optimized and displayed healthcare needs geographically, especially around where to locate treatment centers. That task required complex math (which my wife provided) and serious database work. I went through ADO.NET, and LINQ to SQL. In 2013, I swapped to Entity Framework (EF), when EF 5 supported SQL's spatial (geographical) types, and then moved to EF Core when it came out.

Over the intervening years, I have used EF Core a lot, both on client projects and for building open source libraries. In addition to writing this book, I've written extensively on EF Core in my own blog (www.thereformedprogrammer.net). It turns out that I like taking complex software ideas and trying to make them easy for other people to understand, which I hope I manage to do in this book.

Entity Framework Core in Action, Second Edition, covers all the features of EF Core 5.0, with plenty of examples and code you can run. I've also included numerous patterns and practices to help you build robust and refactorable code. Part 3 of the book, called "Using Entity Framework Core in real-world applications," shows my focus on building and shipping real applications. And I have not one, but three chapters on performance-tuning EF Core so you have many performance improvement techniques at your fingertips when your application is not performing as well as you need it to.

Some of the most pleasurable chapters to write were ones that delved into how EF Core works inside (chapters 1, 6, and 11) and performance-tuning an application (chapters 14, 15, and 16). Personally, I learned a lot from using a modular monolith architecture (chapter 13) and building a substantial application with Cosmos DB (chapter 16). Along the way, I try to present the pros and cons of each approach I use, as I don't believe there is a "silver bullet" answer in software—only a range of compromises that we as developers need to consider when choosing how to implement something.

acknowledgments

Although I did most of the work on the book, I had a lot of help along the way, and I want to say "Thank you" to all those who helped.

Thanks to my wife, Dr. Honora Smith, for putting up with my sitting in front on my computer for three-fourths of a year and for getting me back into programming. I love her to bits. Another special mention goes to my great friend JC for his help and support.

Manning Publications has been great, with a robust and comprehensive process that is hard but thorough, which produces an excellent end product. The team is great, and I'm going to list the significant people in chronological order, starting with Brian Sawyer, Breckyn Ely, Marina Michaels, Joel Kotarski, Rejhana Markanovic, Josip Maras, Heather Tucker, Aleksandar Dragosavljević, and many others who helped with production of the book. Marina Michaels was my main contact for the first edition, and I obviously didn't cause her too many problems, as she kindly agreed to help me on the second edition.

I also got a lot of help from the busy EF Core team. As well as answering numerous issues that were raised on the EF Core GitHub repo, they checked a few of the chapters in which their input was valuable. Arthur Vickers and Shay Rojansky get special mention for reviewing some chapters. The other people on the team, in alphabetical order, are Andriy Svyryd, Brice Lambson, Jeremy Likness, Maurycy Markowski, and Smit Patel.

I would also like to thank Julien Pohie, technical proofreader, and the reviewers: Al Pezewski, Anne Epstein, Foster Haines, Hari Khalsa, Janek López, Jeff Neumann, Joel

Clermont, John Rhodes, Mauro Quercioli, Paul G. Brown, Raushan Jha, Ricardo Peres, Shawn Lam, Stephen Byrne, Sumit K Singh, Thomas Gueth, Thomas Overby Hansen, and Wayne Mather. Your suggestions helped make this a better book.

about this book

Entity Framework Core in Action, Second Edition, is about writing EF Core database code quickly, correctly, and ultimately for fast performance. To help with the "quick, correct, fast" aspects, I include a lot of examples with plenty of tips and techniques. And along the way, I throw in quite a bit on how EF Core works on the inside, because that information will help you when things don't work the way you think they should.

The Microsoft documentation is good but doesn't have room for detailed examples. In this book, I try to give you at least one example of each feature I cover, and you'll often find unit tests in the GitHub repo (see the "About the code" section for links) that test a feature in multiple ways. Sometimes, reading a unit test can convey what's happening much more quickly than reading the text in a book can, so consider the unit tests to be a useful resource.

Who should read this book?

Entity Framework Core in Action, Second Edition, is aimed at both software developers who've never used EF before and seasoned EF Core developers, plus anyone else who wants to know what EF Core is capable of doing. I assume that you're familiar with .NET development with C# and that you have at least some idea of what a relational database is. You don't need to be a C# expert, but if you're new to C#, you might find some of the code hard to read, as I don't explain C#. The book starts with basic EF Core commands, which should be accessible to most C# programmers, but from part 2 onward, the topics get more complex as the content goes deeper into EF Core's features.

How this book is organized

I've tried to build a path that starts with the basics (part 1), goes deep into the details (part 2), and ends with useful tools and techniques (part 3). I try not to assume you'll read the book cover to cover, especially the reference section in part 2, but at least skim-reading the first six chapters will help you understand the basics that I use later in the book.

Part 1: Getting started

- Chapter 1 introduces EF Core with a super-simple console application so you can see all the parts of EF Core in action. I also provide an overview of how EF Core works and why you might like to use it.
- Chapter 2 looks at querying (reading data from) the database. I cover the relationships between data stored in the database and how you can load that related data by using EF Core.
- Chapter 3 moves on to changing the data in a database: adding new data, updating existing data, and deleting data from a database.
- Chapter 4 looks at the different ways you can build robust business logic that uses EF Core to access the database. *Business logic* is the name given to code that implements business rules or workflow that's specific to the business problem your application solves.
- Chapter 5 is about building an ASP.NET Core application that uses EF Core. It pulls together the code developed in chapters 2, 3, and 4 to make a web application. I also talk about deploying the web application and accessing the hosted database.
- Chapter 6 covers a wide range of topics. Most topics contain a description of an aspect of EF Core combined with ways to exploit that feature in your code.

Part 2: Entity Framework Core in depth

- Chapter 7 covers the configuration of nonrelational properties—properties that hold a value, such as `int`, `string`, `DateTime`, and so on.
- Chapter 8 covers the configuration of relationships—the links between classes, such as a `Book` class linking to one or more `Author` classes. It also includes special mapping techniques, such as mapping multiple classes to one table.
- Chapter 9 covers all the ways you can change the database structure when using EF Core. It also looks at the issues that arise when you need to change the structure of a database that's being used by a live application.
- Chapter 10 looks at advanced mapping features and the whole area of detecting and handling concurrency conflicts.
- Chapter 11 digs deep into how EF Core's DbContext works, with a blow-by-blow view of what the various methods and properties do inside your application's DbContext.

Part 3: Using Entity Framework Core in real-world applications

- Chapter 12 introduces two approaches that send messages to the enhanced `SaveChanges/SaveChangesAsync` methods. These two approaches provide another way to combine multiple updates into one transactional database update.
- Chapter 13 looks at applying Domain-Driven Design (DDD) approaches to your classes mapped to the database by EF Core. It also describes another architectural approach used in the part 3 Book App.
- Chapter 14 lists all the issues that could affect the performance of your database accesses and discusses what to do about them.
- Chapter 15 is a worked example of performance-tuning an EF Core application. I take the original Book App display query, developed in part 1, and apply three levels of performance tuning.
- Chapter 16 uses Cosmos DB to further performance-tune the Book App, which uncovers the strengths and weakness of Cosmos DB and its EF Core provider. The chapter ends with what you need to do when changing from one database type to another.
- Chapter 17 is all about unit testing applications that use EF Core. I've also created a NuGet package that you can use to help in your own unit testing.

Appendix

- Appendix A introduces the LINQ language that EF Core uses. This appendix is useful for those who are unfamiliar with LINQ or who want a quick refresher on LINQ.

About the code

I feel that I really know something only if I've written code to use that function or feature, so the companion GitHub repo at http://mng.bz/XdlG is available to you.

> **NOTE** I strongly recommend cloning the code from the GitHub URL shown above. The copy of the repo listed on the Manning book page has problems with the `Part3` branch because of long directory names.

This repo contains the code for the applications I show in the book and unit tests that I ran to make sure that what I said in the book is correct. The repo has three branches:

- `master`, which covers part 1 of the book (chapters 1–6)
- `Part2`, which covers part 2 of the book (chapters 7–11)
- `Part3`, which covers part 3 of the book (chapters 12–17)

To run any of the applications, you should first read the Readme file at http://mng.bz/yYjG in the GitHub repo. Each branch's Readme file has three main sections:

- *What you need to install to run the example applications,* which tells you the development applications, .NET version, and database requirements to run any of the applications in the GitHub repo. (This information is the same for every branch.)

- *What you can run in this branch*, which tells you what application(s) you can run in the branch of the GitHub repo you have selected.
- *How to find and run the unit tests*, which tells you where the unit tests are and the various ways you can run them.

As you work through the three parts of the book, you can select each Git branch to access the code specifically for that part. Also, look out for the associated unit tests, grouped by chapter and feature.

> **NOTE** For chapter 17, which is about unit testing, I used a library I created. This library, which you can find at https://github.com/JonPSmith/EfCore.TestSupport, is an updated version of the EfCore.TestSupport library I created for the first edition of this book, now using new features available in EF Core 5. This library is an open source (MIT license) library, so you can use the NuGet package called `EfCore.TestSupport` (version 5 and later) in your own unit tests.

Code conventions

The code samples in this book, and their output, appear in a `fixed-width font` and are often accompanied by annotations. The code samples are deliberately kept as simple as possible because they aren't intended to be reusable parts that can be plugged into your code. Instead, the code samples are stripped down so that you can focus on the principle being illustrated.

This book contains many examples of source code, both in numbered listings and inline with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Sometimes, code is also **`in bold`** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even these changes were not enough, so some listings include line-continuation markers (➥). Additionally, some comments in the source code have been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

Source code for the examples in this book is available for download from the GitHub repo (http://mng.bz/XdlG).

liveBook discussion forum

Purchase of *Entity Framework Core in Action, Second Edition*, includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to https://livebook.manning.com/book/entity-framework-core-in-action-second-edition. You can also learn more about Manning's forums and the rules of conduct at https://livebook.manning.com/#!/discussion.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Online resources

Here are useful links to the Microsoft documentation and code:

- *Microsoft's EF Core documentation*—https://docs.microsoft.com/en-us/ef/core/
- *EF Core code*—https://github.com/dotnet/efcore
- *ASP.NET Core, working with EF Core*—https://docs.microsoft.com/en-us/aspnet/core/data/
- *Stack Overflow EF Core tag* `[entity-framework-core]`—https://stackoverflow.com

about the author

JON P SMITH is a freelance software developer and architect with a special focus on .NET Core and Azure. He works mainly on the backend of client applications, typically using Entity Framework Core (EF Core) and ASP.NET Core web applications. He works remotely for clients around the world, with many of the projects coming from the United States. He typically helps with designing, performance-tuning, and writing sections of the client's application.

Jon is interested in defining patterns and building libraries that improve the speed of development of applications when using EF Core and ASP.NET Core. His libraries were written because he found some repetitive part of a project he was working on that could be turned into a useful library. You can see a summary of his main libraries on his GitHub overview page (https://github.com/JonPSmith).

Jon also writes on his technical blog at http://www.thereformedprogrammer.net, where he has covered topics related to EF Core, ASP.NET Core, and different architectural approaches. The most popular article on his technical blog site is about an improved ASP.NET Core authorization system; see http://mng.bz/ao2z. He has also spoken at a few conferences and at many Meetups in the United Kingdom.

about the cover illustration

The figure on the cover of *Entity Framework Core in Action, Second Edition*, is captioned "The Wife of a Franc Merchant." The illustration is taken from Thomas Jefferys' *A Collection of the Dresses of Different Nations, Ancient and Modern* (four volumes), London, published between 1757 and 1772. The title page states that these illustrations are hand-colored copperplate engravings, heightened with gum arabic.

Thomas Jefferys (1719–71) was called "Geographer to King George III." He was an English cartographer and the leading map supplier of his day. He engraved and printed maps for government and other official entities, and produced a wide range of commercial maps and atlases, especially of North America. His work as a mapmaker sparked an interest in the local dress customs of the lands he surveyed and mapped, which are brilliantly displayed in this collection. Fascination with faraway lands and travel for pleasure were relatively new phenomena in the late 18th century, and collections such as this one were popular, introducing both the tourist and the armchair traveler to the inhabitants of other countries.

The diversity of the drawings in Jefferys' volumes speaks vividly of the uniqueness and individuality of the world's nations some 200 years ago. Dress codes have changed since then, and the diversity by region and country, so rich at the time, has faded away. Now, it's often hard to tell the inhabitants of one continent from another. Perhaps (trying to view things optimistically) we've traded cultural and visual diversity for more varied personal lives—or more varied and interesting intellectual and technical lives.

At a time when it's difficult to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of the regional life of two centuries ago, brought back to life by Jeffreys' pictures.

Part 1

Getting started

Data is everywhere, growing by petabytes per year, and a lot of it is stored in databases. Millions of applications are also out there—at the beginning of 2021, there were 1.2 billion websites—and most of them need to access data in databases. And I haven't started on the Internet of Things yet. So it shouldn't be a surprise that Gartner, a leading research and advisory company, says that global IT spending will reach $3.7 trillion in 2021 (http://mng.bz/gonl).

The good news for you is that your skills will be in demand. But the bad news is that the pressure to develop applications quickly is unrelenting. This book is about one tool that you can use to write database access code quickly: Microsoft's Entity Framework Core (EF Core). EF Core provides an object-oriented way to access relational and nonrelational (NoSQL) databases in the .NET environment. The cool thing about EF Core and the other .NET Core libraries is that they can run on the Windows, Linux, and Apple platforms, and they're fast.

In part 1, I get you into the code straightaway. In chapter 1, you'll build a super-simple console application, and by the end of chapter 5, you'll build a reasonably sophisticated web application that sells books. Chapters 2 and 3 explain the reading and writing of data to a relational database, respectively, and chapter 4 covers writing your business logic. In chapter 5, you'll use Microsoft's ASP.NET Core web framework to build the example book-selling site. Chapter 6 expands your knowledge of how EF Core works inside through a series of useful techniques for solving database problems, such as a quick way to copy data in the database.

You'll have a lot of learning to do in part 1, even though I skip a few topics, mainly by relying on a lot of EF Core's default settings. Nevertheless, part 1 should give you a good understanding of what EF Core can do, with later parts growing your knowledge with extra EF Core features, more details on how you can configure EF Core, and chapters devoted to specific areas such as performance tuning.

1

Introduction to Entity Framework Core

This chapter covers

- Understanding the anatomy of an EF Core application
- Accessing and updating a database with EF Core
- Exploring a real-world EF Core application
- Deciding whether to use EF Core in your application

Entity Framework Core, or *EF Core*, is a library that software developers can use to access databases. There are many ways to build such a library, but EF Core is designed as an *object-relational mapper* (*O/RM*). O/RMs work by mapping between two worlds: the relational database, with its own API, and the object-oriented software world of classes and software code. EF Core's main strength is allowing software developers to write database access code quickly in a language that you may know better than SQL.

EF Core is multiplatform-capable: it can run on Windows, Linux, and Apple. It does this as part of the .NET Core initiative—hence the *Core* part of the EF Core name. .NET 5 covers the whole range of desktop, web, cloud, mobile, gaming, Internet of Things (IoT), and artificial intelligence (AI), but this book is focused on EF Core.

EF Core isn't the first version of Entity Framework; an existing, non-Core, Entity Framework library is known as *EF6.x*. EF Core starts with years of experience built into it via feedback from these previous versions, 4 to 6.x. It has kept the same type of interface as EF6.x but has major changes underneath, such as the ability to handle nonrelational databases, which EF6.x wasn't designed to do. I had used EF5 and EF6 in many applications before EF Core came along, which allowed me to see the significant improvements EF Core made over EF6.x in both features and performance.

This book is for software developers who are already using EF Core, as well as developers who've never used Entity Framework, and seasoned EF6.x developers who want to move over to EF Core. I do assume that you're familiar with .NET development using C# and that you have at least some idea of what relational databases are. I don't assume you know how to write Structured Query Language (SQL), the language used by a majority of relational databases, because EF Core can do most of that for you. But I do show the SQL that EF Core produces because it helps you understand what's going on; using some of the EF Core advanced features requires you to have SQL knowledge, but the book provides plenty of diagrams to help you along the way.

TIP If you don't know a lot about SQL and want to learn more, I suggest the W3Schools online resource: https://www.w3schools.com/sql/sql_intro.asp. The SQL set of commands is vast, and EF Core queries use only a small subset (such as `SELECT`, `WHERE`, and `INNER JOIN`), so that resource is a good place to start.

This chapter introduces you to EF Core through the use of a small application that calls into the EF Core library. You'll look under the hood to see how EF Core interprets software commands and accesses the database. Having an overview of what's happening inside EF Core will help you as you read through the rest of the book.

1.1 What you'll learn from this book

The book gives you an introduction to EF Core, starting with the basics and advancing to some more complex parts of EF Core. To get the best out of this book, you should be comfortable with developing applications using C#, including creating projects and loading NuGet packages. You will learn

- The fundamentals of using EF Core to access a database
- How to use EF Core in an ASP.NET Core web application
- The many ways you can configure EF Core to work exactly as you need
- Some of the deeper database features you might want to use
- How to handle changes in the database layout as your application grows
- How to improve the performance of your database code
- Most important, how to make sure that your code is working correctly

Throughout the book I build simple but fully featured applications so that you can see EF Core working in real situations. All these applications are available via the example repo, which also includes lots of tips and techniques I have picked up while working as a contract developer and on my own projects.

1.2 My "lightbulb moment" with Entity Framework

Before we get into the nitty-gritty, let me tell you about one defining moment I had when using Entity Framework that put me on the road to embracing EF. It was my wife who got me back into programming after a 21-year gap (that's a story in itself!).

My wife, Dr. Honora Smith, is a lecturer in mathematics at the University of Southampton, who specializes in the modeling of healthcare systems, especially focusing on where to locate health facilities. I had worked with her to build several applications to do geographic modeling and visualization for the UK National Health Service and worked for South Africa on optimizing HIV/AIDS testing.

At the start of 2013, I decided to build a web application specifically for healthcare modeling. I used ASP.NET MVC4 and EF5, which had just come out and supported SQL spatial types that handle geographic data. The project went okay, but it was hard work. I knew that the frontend was going to be hard; it was a single-page application using Backbone.js, but I was surprised at how long it took me to do the server-side work.

I applied good software practices and made sure that the database and business logic were matched to the problem space—that of modeling and optimizing the location of health facilities. That was fine, but I spent an inordinate amount of time writing code to convert the database entries and business logic to a form suitable to show to the user. Also, I was using a Repository/Unit of Work pattern to hide EF5 code, and I was continually having to tweak areas to make the repository work properly.

At the end of a project, I always look back and ask, "Could I have done that better?" As a software architect, I'm always looking for parts that (a) worked well, (b) were repetitious and should be automated, or (c) had ongoing problems. This time, the list was as follows:

- *Worked well*—The ServiceLayer, a layer in my application that isolated/adapted the lower layers of the application from the ASP.NET MVC4 frontend, worked well. (I introduce this layered architecture in chapter 2.)
- *Was repetitious*—I used ViewModel classes, also known as *data transfer objects* (DTOs), to represent the data I needed to show to the user. Using a ViewModel/DTO worked well, but writing the code to copy the database tables to the ViewModel/DTO was repetitious and boring. (I also talk about ViewModels/DTOs in chapter 2.)
- *Had ongoing problems*—The Repository/Unit of Work pattern didn't work for me. Ongoing problems occurred throughout the project. (I cover the Repository pattern and alternatives in chapter 13.)

As a result of my review, I built a library called GenericServices (https://github.com/JonPSmith/GenericServices) to use with EF6.x. This library automated the copying of data between database classes and ViewModels/DTOs, and removed the need for a Repository/Unit of Work pattern. It seemed to be working well, but to stress-test GenericServices, I decided to build a frontend over one of Microsoft's example databases: the AdventureWorks 2012 Lite database. I built the whole application with the help of a frontend UI library in 10 days!

Entity Framework + the right libraries + the right approach
= quick development of database access code

The site wasn't too pretty, but appearance wasn't the point. By analyzing my use of the Repository/Unit of Work pattern with EF6.x, I found a better approach. Then, by encapsulating this better approach into my GenericServices library, I automated the process of building Create, Read, Update, and Delete (CRUD) database commands. The result allowed me to build applications really quickly—definitely a "lightbulb moment," and I was hooked on EF.

Since then, I've built new libraries that work with EF Core, which I have found to significantly speed the development of 90% of my database accesses. I work as a contract developer, and these libraries, which are open source and available to you too, automate some of the standard requirements, allowing me to concentrate on the harder topics, such as understanding the client's needs, writing custom business logic, and performance-tuning where necessary. I will be talking about these libraries in later chapters.

1.3 Some words for existing EF6.x developers

TIME-SAVER If you haven't used Entity Framework 6.x, you can skip this section.

If you know EF6.x, much of EF Core will be familiar to you. To help you navigate this book quickly, I've added EF6 notes.

EF6 Watch for notes like this throughout the book. They point out the places where EF Core is different from EF6.x. Also be sure to look at the summaries at the end of each chapter, which point out the biggest changes between EF6 and EF Core in the chapter.

I'll also give you one tip from my journey of learning EF Core. I know EF6.x well, but that knowledge became a bit of a problem when I started using EF Core. I was using an EF6.x approach to problems and didn't notice that EF Core had new ways to solve them. In most cases, the approaches are similar, but in some areas, they aren't.

My advice to you, as an existing EF6.x developer, is to approach EF Core as a new library that someone has written to mimic EF6.x, but understand that it works in a different way. That way, you'll keep your eyes open for the new and different ways of doing things in EF Core.

1.4 An overview of EF Core

You can use EF Core as an O/RM that maps between the relational database and the .NET world of classes and software code. Table 1.1 shows how EF Core maps the two worlds of the relational database and .NET software.

Table 1.1 EF Core mapping between a database and .NET software

Relational database	.NET software
Table	.NET class
Table columns	Class properties/fields
Rows	Elements in .NET collections—for instance, `List`
Primary keys: unique row	A unique class instance
Foreign keys: define a relationship	Reference to another class
SQL—for instance, `WHERE`	.NET LINQ—for instance, `Where(p => …`

1.4.1 The downsides of O/RMs

Making a good O/RM is complex. Although EF6.x or EF Core can seem easy to use, at times the EF Core "magic" can catch you by surprise. Let me mention two issues to be aware of before we dive into how EF Core works.

The first issue is *object-relational impedance mismatch*. Database servers and object-oriented software use different principles; databases use primary keys to define that a row is unique, whereas .NET class instances are, by default, considered unique by their reference. EF Core handles much of the impedance mismatch for you, but your .NET classes gain primary and foreign keys, which is extra data needed only for the database. Your software-only version of the classes doesn't need those extra properties, but the database does.

The second issue is that an O/RM—and especially an O/RM as comprehensive as EF Core—is the opposite of the first issue. EF Core "hides" the database so well that you can sometimes forget about the database underneath. This problem can cause you to write code that would work well in C# but doesn't work for a database. One example is having an expression body property return the full name of a person by combining the `FirstName` and `LastName` properties in the class, such as

```
public string FullName => $"{FirstName} {LastName}";
```

An expression body property such as the one just shown is the right thing to do in C#, but the same property would throw an exception if you tried to filter or order on that property, because EF Core needs a FullName column in the table so that it can apply an SQL `WHERE` or `ORDER` command at the database level.

That's why I spend time in this chapter showing how EF Core works on the inside and the SQL it produces. The more you understand about what EF Core is doing, the better equipped you'll be to write good EF Core code, and—more important—you'll know what to do when your code doesn't work.

> **NOTE** Throughout this book, I use a "Get it working, but be ready to make it faster if I need to" approach to using EF Core. EF Core allows me to develop quickly, but I'm aware that because of EF Core, or my poor use of it, the performance of my database access code might not be good enough for a particular business need. Chapter 5 covers how to isolate your EF Core so you can tune it with minimal side effects, and chapter 15 shows how to find and improve database code that isn't fast enough.

1.5 What about NoSQL?

We can't talk about relational databases without mentioning nonrelational databases, also known colloquially as NoSQL (see http://mng.bz/DW63). Both relational and nonrelational databases have a role in modern applications. I've used both SQL Server (relational database) and Azure Tables (nonrelational database) in the same application to handle two business needs.

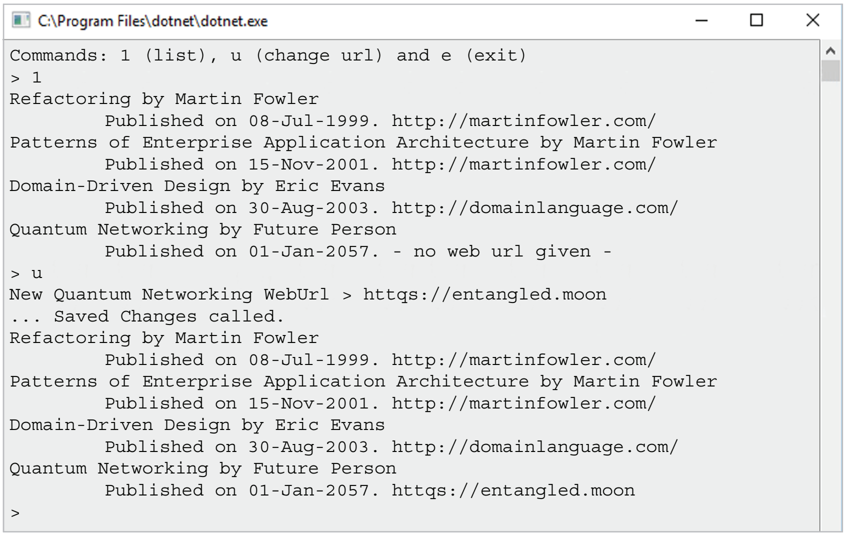
EF Core handles both relational and nonrelational databases—a departure from EF6.x, which was designed around relational databases only. Most of the EF Core commands covered in this book apply to both types of databases, but there are some differences at the database level between relational databases and NoSQL databases, which leave out some of the more complex database commands in favor of scalability and performance.

EF Core 3.0 added a database provider for the Azure NoSQL database called Cosmos DB, which I cover in chapter 16. In that chapter, I point out the differences between a relational database and Cosmos DB; I was surprised by what I found. Now that EF Core has been altered to handle NoSQL databases, I expect that more NoSQL database providers will be written.

> **NOTE** Cosmos DB and other NoSQL databases have many strengths compared with SQL databases. It's much easier, for example, to have multiple copies of NoSQL databases around the world, which gives the user quicker access, and if a data center goes down, other copies can take over the load. But NoSQL databases also have some limitations compared with SQL databases; read chapter 16 for an in-depth analysis of Cosmos DB's benefits and limitations.

1.6 Your first EF Core application

In this chapter, you'll start with a simple example so that we can focus on what EF Core is doing rather than what the code is doing. For this example, you're going to use a small console application called MyFirstEfCoreApp, which accesses a simple database. The MyFirstEfCoreApp application's job is to list and update books in a supplied database. Figure 1.1 shows the console output.



```
Commands: 1 (list), u (change url) and e (exit)
> 1
Refactoring by Martin Fowler
        Published on 08-Jul-1999. http://martinfowler.com/
Patterns of Enterprise Application Architecture by Martin Fowler
        Published on 15-Nov-2001. http://martinfowler.com/
Domain-Driven Design by Eric Evans
        Published on 30-Aug-2003. http://domainlanguage.com/
Quantum Networking by Future Person
        Published on 01-Jan-2057. - no web url given -
> u
New Quantum Networking WebUrl > httqs://entangled.moon
... Saved Changes called.
Refactoring by Martin Fowler
        Published on 08-Jul-1999. http://martinfowler.com/
Patterns of Enterprise Application Architecture by Martin Fowler
        Published on 15-Nov-2001. http://martinfowler.com/
Domain-Driven Design by Eric Evans
        Published on 30-Aug-2003. http://domainlanguage.com/
Quantum Networking by Future Person
        Published on 01-Jan-2057. httqs://entangled.moon
>
```

List all four books.

Update *Quantum Networking* book.

Figure 1.1 The console application provides a command that uses an EF Core query to read and display all the books in your sample database, plus a command to update the database. These two commands show you how EF Core works inside.

This application isn't going to win any prizes for its interface or complexity, but it's a good place to start, especially because I want to show you how EF Core works internally to help you understand what's going on later in this book.

You can download this example application from the Git repo at http://mng.bz/XdlG. You can look at the code and run the application. To do this, you need software development tools.

1.6.1 What you need to install

Microsoft has two development tools for a .NET Core application: Visual Studio and Visual Studio Code (shortened to VS Code). Visual Studio is slightly easier to use, and I suggest that newcomers to .NET use Visual Studio. You can download Visual Studio from www.visualstudio.com. Numerous versions exist, including a free community version, but you need to read the license to make sure that you qualify; see www.visualstudio.com/vs/community.

When you install Visual Studio on Windows, make sure to include the .NET Core Cross-Platform Development feature and Data storage and processing, which are in the Other Toolsets section during the Install Workloads stage. Selecting the .NET Core Cross-Platform Development feature will also install the .NET Core software development kit (SDK) on your system; you need this SDK to create applications with .NET. See http://mng.bz/2x0T for more information.

If you want to use VS Code, which is free, you can download it from https://code.visualstudio.com. You will need to do more setting up on your system, such as installing the latest .NET Core SDK on your computer and the localdb SQL Server. As I said, if you are new to coding in Microsoft's system, I suggest using Visual Studio on Windows, as it sets up a lot of things for you.

One version of Visual Studio runs on an Apple Macintosh machine, and versions of VS Code run in Windows, on a Mac, and in Linux. If you want to run any of the applications or unit tests, you must have an SQL Server instance on your system. You may need to change the server name in the connection strings for the applications and the unit-test project.

You can run your unit tests by using Visual Studio's built-in Test Explorer, available from the Test menu. If you're using VS Code, the test runner is also built in, but you need to set up the build and test tasks in the VS Code tasks.json file, which allows you to run all the tests via the Task > Test command.

1.6.2 Creating your own .NET Core console app with EF Core

I know that many developers like to create their own applications, because building the code yourself means that you know exactly what's involved. This section details how to create the .NET console application MyFirstEfCoreApp by using Visual Studio.

CREATING A .NET CORE CONSOLE APPLICATION

Visual Studio has a great set of tutorials, and you can find an example of creating a C# console application at http://mng.bz/e56z.

> **TIP** You can find out which version of .NET your application is using by choosing Project > MyFirstEfCoreApp Properties from the main menu; the Application tab shows the Target Framework. Some versions of EF Core require a certain version of .NET Core.

ADDING THE EF CORE LIBRARY TO YOUR APPLICATION

You can install the NuGet library in various ways. The more visual way is to use the NuGet Package Manager; you can find a tutorial at http://mng.bz/pVeG. For this application, you need the EF Core package for the database that your application is going to access. In this case, you choose the Microsoft.EntityFrameworkCore.SqlServer NuGet package, because it'll use the development SQL Server that was installed when you installed Visual Studio.

The other thing you need to look at is the version number of the NuGet package you are about to install. EF Core has been built such that each major release has its own number. A version number of 5.1.3, for example, means EF Core major version 5, with minor release 1 and patch (bug fix) version 3. Often, you need to load different EF Core packages in different projects. You might load Microsoft.EntityFrameworkCore in your data layer and Microsoft.EntityFrameworkCore.SqlServer in the web app, for example. If you need to do this, you should try to use NuGet packages with the same Major.Minor.Patch listed in your project's properties. If a match is not found, make sure that the NuGet Major.Minor version matches your project's version.

Downloading and running the example application from the Git repo

You have two options for downloading and running the MyFirstEfCoreApp console application in the Git repo: Visual Studio or VS Code. You can find another Visual Studio tutorial, “Open a project from a repo,” at http://mng.bz/OEOn. The repo associated with this book is http://mng.bz/XdlG.

Be sure to select the right branch. A Git repo has branches that allow you to switch between different versions of the code. For this book, I created three main branches: master, which contains the code for part 1 (chapters 1–6); Part2, which contains the code for part 2 (chapters 7–11); and Part3, which contains the code for part 3 (chapters 12–17).

By default, the repo will be opened in the master branch, so someone who is not used to Git can get started straight away. The Readme file in each branch has more information about what you need to install and what you can run.

1.7 The database that MyFirstEfCoreApp will access

EF Core is about accessing databases, but where does that database come from? EF Core gives you two options: EF Core can create it for you, in what’s known as a *code-first* approach, or you can provide an existing database you built outside EF Core, in what’s known as a *database-first* approach. The first part of the book uses code-first because it’s the approach that many developers use.

EF6 In EF6, you could use an EDMX/database designer to design your database visually, an option known as *design-first*. EF Core doesn’t support this design-first approach in any form, and there are no plans to add it.

In this chapter, we’re not going to learn about how a database is created. To allow the MyFirstEfCoreApp application to work, the code will create the database and add the test data if there isn’t an existing database.

NOTE In my code, I use a basic EF Core command meant for unit testing to create the database, because it’s simple and quick. Chapter 5 covers how to get EF Core to create a database properly, and chapter 9 presents the whole issue of creating and changing the structure of the database, known as the database's *schema*.

For this MyFirstEfCoreApp application example, I created a simple database, shown in figure 1.2, with only two tables:

- A Books table holding the book information
- An Author table holding the author of each book

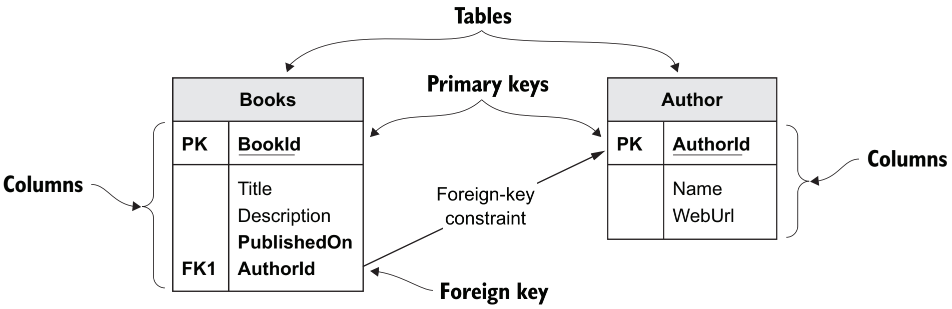
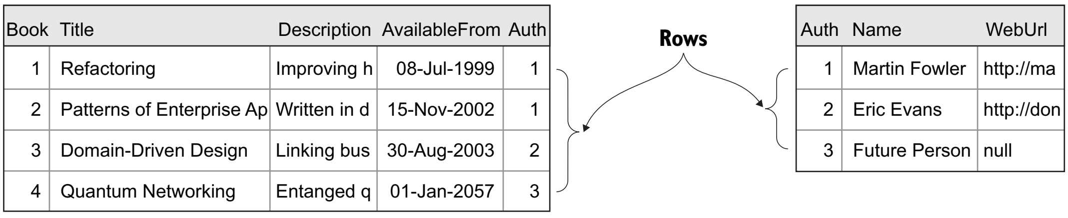


Figure 1.2 Our example relational database with two tables: Books and Author

ADVANCED NOTE In this example, I let EF Core name the tables using its default configuration settings. The Books table name comes from the `DbSet<Book> Books` property shown in figure 1.5. The Author table name hasn't got a `DbSet<T>` property in figure 1.5, so EF Core uses the name of the class.

Figure 1.3 shows the content of the database. It holds only four books, the first two of which have the same author: Martin Fowler.



Book	Title	Description	AvailableFrom	Auth
1	Refactoring	Improving h	08-Jul-1999	1
2	Patterns of Enterprise Ap	Written in d	15-Nov-2002	1
3	Domain-Driven Design	Linking bus	30-Aug-2003	2
4	Quantum Networking	Entanged q	01-Jan-2057	3

Auth	Name	WebUrl
1	Martin Fowler	http://ma
2	Eric Evans	http://don
3	Future Person	null

Figure 1.3 The content of the database, showing four books, two of which have the same author

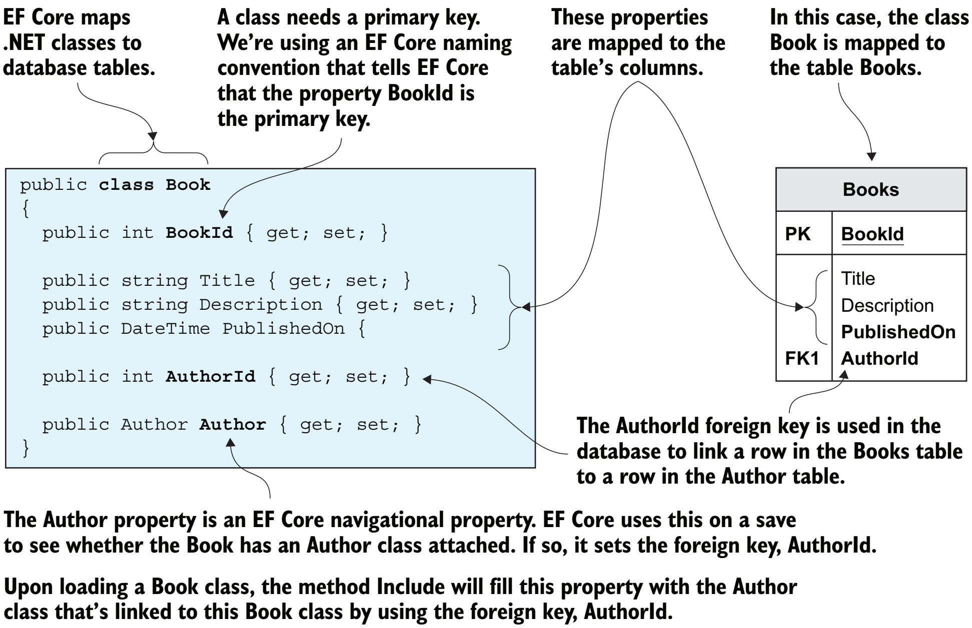
1.8 Setting up the MyFirstEfCoreApp application

Having created and set up a .NET console application, you can now start writing EF Core code. You need to write two fundamental parts before creating any database access code:

- The classes that you want EF Core to map to the tables in your database
- The application's DbContext, which is the primary class that you'll use to configure and access the database

1.8.1 The classes that map to the database: Book and Author

EF Core maps classes to database tables. Therefore, you need to create a class that will define the database table or match a database table if you already have a database. Lots of rules and configurations exist (covered in chapters 7 and 8), but figure 1.4 gives the typical format of a class that's mapped to a database table.



EF Core maps .NET classes to database tables.

A class needs a primary key. We're using an EF Core naming convention that tells EF Core that the property BookId is the primary key.

These properties are mapped to the table's columns.

In this case, the class Book is mapped to the table Books.

```
public class Book
{
  public int BookId { get; set; }

  public string Title { get; set; }
  public string Description { get; set; }
  public DateTime PublishedOn {

  public int AuthorId { get; set; }

  public Author Author { get; set; }
}
```

Books	
PK	BookId
	Title Description PublishedOn
FK1	AuthorId

The AuthorId foreign key is used in the database to link a row in the Books table to a row in the Author table.

The Author property is an EF Core navigational property. EF Core uses this on a save to see whether the Book has an Author class attached. If so, it sets the foreign key, AuthorId.

Upon loading a Book class, the method Include will fill this property with the Author class that's linked to this Book class by using the foreign key, AuthorId.

Figure 1.4 The.NET class `Book`, on the left, maps to a database table called Books, on the right. This is a typical way to build your application, with multiple classes that map to database tables.

Listing 1.1 shows the other class you'll be using: `Author`. This class has the same structure as the `Book` class in figure 1.4, with a primary key that follows the EF Core naming conventions of `<ClassName>Id` (see section 7.3.5). The `Book` class also has a navigational property of type `Author` and an `int` type property called `AuthorId` that matches the `Author`'s primary key. These two properties tell EF Core that you want a link from the `Book` class to the `Author` class and that the `AuthorId` property should be used as the foreign key to link the two tables in the database.

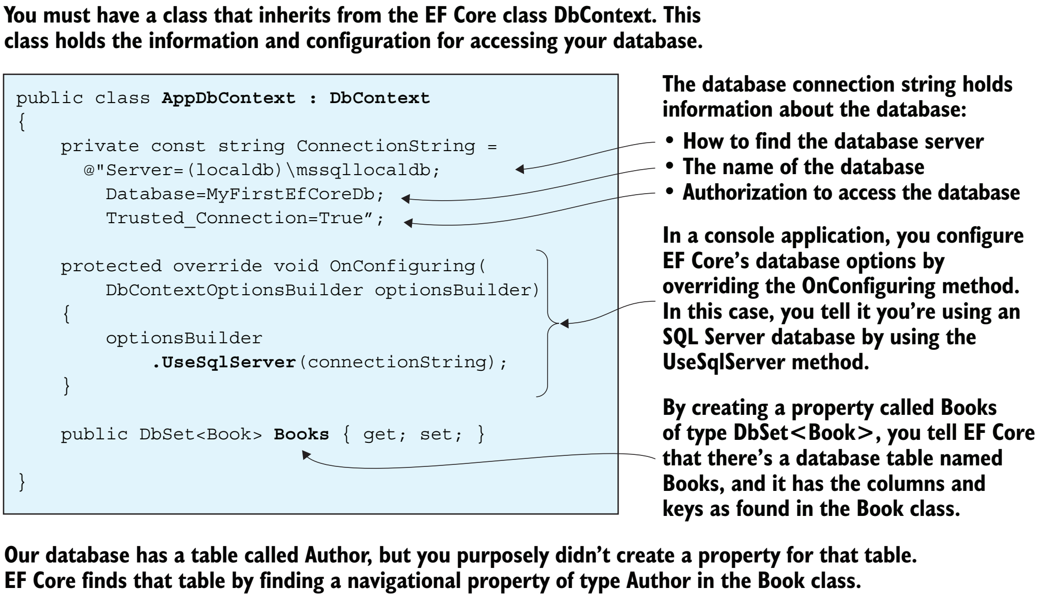
Listing 1.1 The `Author` class from `MyFirstEfCoreApp`

```
public class Author
{
    public int AuthorId { get; set; }
    public string Name { get; set; }
    public string WebUrl { get; set; }
}
```

Holds the primary key of the Author row in the DB. Note that the foreign key in the Book class has the same name.

1.8.2 The application's DbContext

The other important part of the application is DbContext, a class you create that inherits from EF Core's DbContext class. This class holds the information EF Core needs to configure that database mapping and is also the class you use in your code to access the database (see section 1.9.2). Figure 1.5 shows the application's DbContext, called `AppDbContext`, that the MyFirstEfCoreApp console application uses.



You must have a class that inherits from the EF Core class DbContext. This class holds the information and configuration for accessing your database.

```
public class AppDbContext : DbContext
{
    private const string ConnectionString =
      @"Server=(localdb)\mssqllocaldb;
        Database=MyFirstEfCoreDb;
        Trusted_Connection=True";

    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder
            .UseSqlServer(connectionString);
    }

    public DbSet<Book> Books { get; set; }

}
```

The database connection string holds information about the database:

- **How to find the database server**
- **The name of the database**
- **Authorization to access the database**

In a console application, you configure EF Core's database options by overriding the OnConfiguring method. In this case, you tell it you're using an SQL Server database by using the UseSqlServer method.

By creating a property called Books of type DbSet<Book>, you tell EF Core that there's a database table named Books, and it has the columns and keys as found in the Book class.

Our database has a table called Author, but you purposely didn't create a property for that table. EF Core finds that table by finding a navigational property of type Author in the Book class.

Figure 1.5 Two main parts of the application's DbContext created for the MyFirstEfCoreApp console application. First, the setting of the database options defines what type of database to use and where it can be found. Second, the `DbSet<T>` property (or properties) tell(s) EF Core what classes should be mapped to the database.

In our small example application, all the decisions on the modeling are done by EF Core, which works things out by using a set of conventions. You have loads of extra ways to tell EF Core what the database model is, and these commands can get complex. It takes chapter 7, chapter 8, and a bit of chapter 10 to cover all the options available to you as a developer.

Also, you're using a standard approach to define the database access in a console application: overriding the `OnConfiguring` method inside the application's DbContext and providing all the information EF Core needs to define the type and location of the database. The disadvantage of this approach is that it has a fixed connection string, which makes development and unit testing difficult.

For ASP.NET Core web applications, this problem is bigger because you want to access a local database for testing, and a different hosted database when running in production. In chapter 2, as you start building an ASP.NET Core web application, you'll use a different approach that allows you to change the database string (see section 2.2.2).

1.9 Looking under the hood of EF Core

Having run the MyFirstEfCoreApp application, you can now use it to see how an EF Core library works. The focus isn't on the application code, but on what happens inside the EF Core library when you read and write data to the database. My aim is to provide you a mental model of how EF Core accesses a database. This model should help as you dig into the myriad commands described throughout the rest of this book.

Do you really need to know how EF Core works inside to use it?

You can use the EF Core library without bothering to learn how it works. But knowing what's happening inside EF Core will help you understand why the various commands work the way they do. You'll also be better armed when you need to debug your database access code.

The following pages include lots of explanations and diagrams to show you what happens inside EF Core. EF Core "hides" the database so that you, as a developer, can write database access code easily—which does work well in practice. But as I stated earlier, knowing how EF Core works can help you if you want to do something more complex or if things don't work the way you expect.

1.9.1 Modeling the database

Before you can do anything with the database, EF Core must go through a process that I refer to as *modeling the database*. This modeling is EF Core's way of working out what the database looks like by looking at the classes and other EF Core configuration data. Then EF Core uses the resulting model in all database accesses.

The modeling process is kicked off the first time you create the application's DbContext, in this case called `AppDbContext` (shown in figure 1.5). It has one property, `DbSet<Book>`, which is the way that the code accesses the database.

Figure 1.6 provides an overview of the modeling process, which will help you understand the process EF Core uses to model the database. Later chapters introduce you to a range of commands that allow you to configure your database more precisely, but for now, you'll use the default configurations.

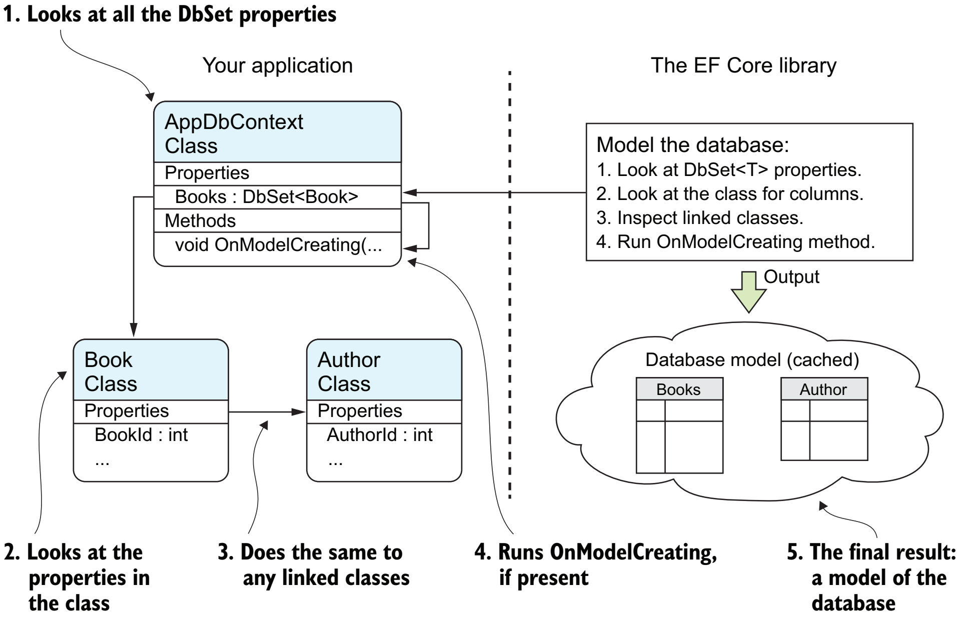


Figure 1.6 The figure shows how EF Core will create a model of the database your classes map to. First, it looks at the classes you have defined via the `DbSet<T>` properties; then it looks down all the references to other classes. Using these classes, EF Core can work out the default model of the database. But then it runs the `OnModelCreating` method in the application's DbContext, which you can override to add your specific commands to configure the database the way you want it.

Figure 1.6 shows the modeling steps that EF Core uses on our `AppDbContext`, which happens the first time you create an instance of the `AppDbContext`. (After that, the model is cached, so that subsequent instances are created quickly.) The following text provides a more detailed description of the process:

- EF Core looks at the application's DbContext and finds all the public `DbSet<T>` properties. From this data, it defines the initial name for the one table it finds: Books.
- EF Core looks through all the classes referred to in `DbSet<T>` and looks at its properties to work out the column names, types, and so forth. It also looks for special attributes on the class and/or properties that provide extra modeling information.
- EF Core looks for any classes that the `DbSet<T>` classes refer to. In our case, the `Book` class has a reference to the `Author` class, so EF Core scans that class too. It

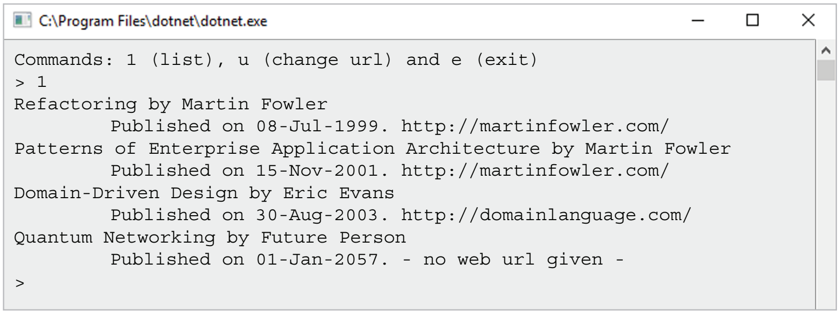
carries out the same search on the properties of the `Author` class as it did on the `Book` class in step 2. It also takes the class name, `Author`, as the table name.
- For the last input to the modeling process, EF Core runs the virtual method `OnModelCreating` inside the application's DbContext. In this simple application, you don't override the `OnModelCreating` method, but if you did, you could provide extra information via a fluent API to do more configuration of the modeling.
- EF Core creates an internal model of the database based on all the information it gathered. This database model is cached so that later accesses will be quicker. Then this model is used for performing all database accesses.

You might have noticed that figure 1.6 shows no database. This is so because when EF Core is building its internal model, it doesn't look at the database. I emphasize that fact to show how important it is to build a good model of the database you want; otherwise, problems could occur if a mismatch exists between what EF Core thinks the database looks like and what the actual database is like.

In your application, you may use EF Core to create the database, in which case there's no chance of a mismatch. Even so, if you want a good and efficient database, it's worth taking care to build a good representation of the database you want in your code so that the created database performs well. The options for creating, updating, and managing the database structure are a big topic, detailed in chapter 9.

1.9.2 Reading data from the database

You're now at the point where you can access the database. The console application has a list (`l`) command, which reads the database and prints the information on the terminal. Figure 1.7 shows the result of running the console application and typing l.



```
Commands: 1 (list), u (change url) and e (exit)
> 1
Refactoring by Martin Fowler
         Published on 08-Jul-1999. http://martinfowler.com/
Patterns of Enterprise Application Architecture by Martin Fowler
         Published on 15-Nov-2001. http://martinfowler.com/
Domain-Driven Design by Eric Evans
         Published on 30-Aug-2003. http://domainlanguage.com/
Quantum Networking by Future Person
         Published on 01-Jan-2057. - no web url given -
>
```

Figure 1.7 Output of the console application when listing the content of the database

The following listing shows the code that's called to list all the books, with each author, out to the console.

Listing 1.2 The code to read all the books and output them to the console

```
public static void ListAll()
{
    using (var db = new AppDbContext())
    {
        foreach (var book in
            db.Books.AsNoTracking()

            .Include(book => book.Author))
        {
            var webUrl = book.Author.WebUrl == null
                ? "- no web URL given -"
                : book.Author.WebUrl;
            Console.WriteLine(
                $"{book.Title} by {book.Author.Name}");
            Console.WriteLine("     " +
                "Published on " +
                $"{book.PublishedOn:dd-MMM-yyyy}" +
                $". {webUrl}");
        }
    }
}
```

You create the application's DbContext through which all database accesses are done.

Reads all the books. AsNoTracking indicates that this access is read-only.

The include causes the author information to be loaded with each book. See chapter 2 for more information.

EF Core uses Microsoft's .NET's Language Integrated Query (LINQ) to carry the commands it wants done, and normal .NET classes to hold the data. Listing 1.2's query doesn't include any LINQ methods, but later in the book, you'll see plenty of LINQ examples.

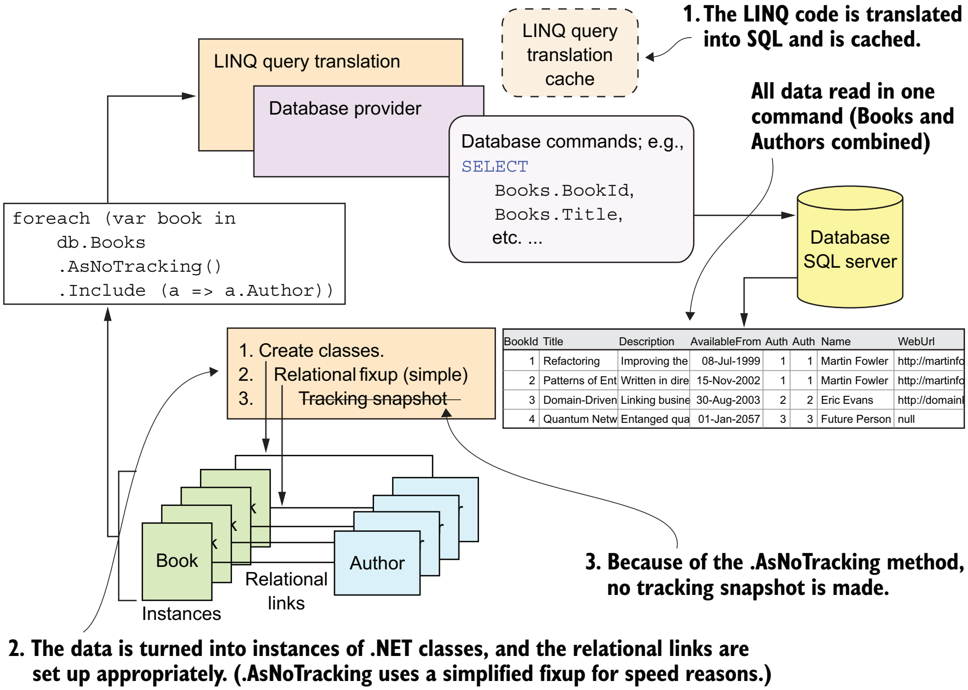
> **NOTE** *Learning LINQ will be essential to you, as EF Core uses LINQ commands for database accesses.* The appendix provides a brief introduction to LINQ. Plenty of online resources are also available; see http://mng.bz/YqBN.

Two lines of code in bold in listing 1.2 cause the database access. Now let's see how EF Core uses that LINQ code to access the database and return the required books with their authors. Figure 1.8 follows those lines of code down into the EF Core library, through the database, and back.

The process to read data from the database is as follows:

- The query `db.Books.AsNoTracking().Include(book => book.Author)` accesses the `DbSet<Book>` property in the application's DbContext and adds a `.Include(book => book.Author)` at the end to ask that the Author parts of the relationship are loaded too. This is converted by the database provider into an SQL command to access the database. The resulting SQL is cached to avoid the cost of retranslation if the same database access is used again.

EF Core tries to be as efficient as possible on database accesses. In this case, it combines the two tables it needs to read, Books and Author, into one big table so that it can do the job in one database access. The following listing shows the SQL created by EF Core and the database provider.



```
foreach (var book in
    db.Books
    .AsNoTracking()
    .Include (a => a.Author))
```

```
SELECT
   Books.BookId,
   Books.Title,
  etc. ...
```

BookId	Title	Description	AvailableFrom	Auth	Auth	Name	WebUrl
1	Refactoring	Improving the	08-Jul-1999	1	1	Martin Fowler	http://martinfo
2	Patterns of Ent	Written in dire	15-Nov-2002	1	1	Martin Fowler	http://martinfo
3	Domain-Driven	Linking busine	30-Aug-2003	2	2	Eric Evans	http://domainl
4	Quantum Netw	Entanged qua	01-Jan-2057	3	3	Future Person	null

1. The LINQ code is translated into SQL and is cached.

All data read in one command (Books and Authors combined)

3. Because of the .AsNoTracking method, no tracking snapshot is made.

2. The data is turned into instances of .NET classes, and the relational links are set up appropriately. (.AsNoTracking uses a simplified fixup for speed reasons.)

Figure 1.8 A look inside EF Core as it executes a database query

Listing 1.3 SQL command produced to read `Books` and `Author`

```
SELECT [b].[BookId],
[b].[AuthorId],
[b].[Description],
[b].[PublishedOn],
[b].[Title],
[a].[AuthorId],
[a].[Name],
[a].[WebUrl]
FROM [Books] AS [b]
INNER JOIN [Author] AS [a] ON
[b].[AuthorId] = [a].[AuthorId]
```

After the database provider has read the data, EF Core puts the data through a process that (a) creates instances of the .NET classes and (b) uses the database relational links, called *foreign keys*, to correctly link the .NET classes by reference—called a *relational fixup*. Because we added the `AsNoTracking` method, the relational fixup uses a simplified fixup for speed reasons.

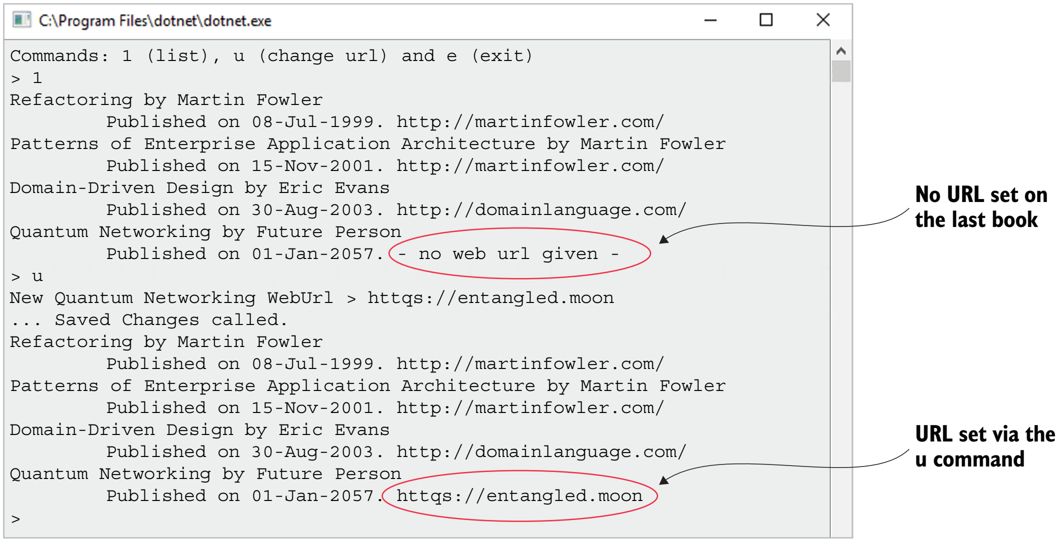
NOTE I discuss the differences between the `AsNoTracking` simplified relational fixup and the normal relational fixup in section 6.1.2.

The result is a set of .NET class instances with the Book's `Author` property linked to an `Author` class containing the author's information. In this example, two books have the same author, Martin Fowler, so there are two instances of the `Author` class, both holding the same information on Martin Fowler.

Because the code includes the command `AsNoTracking`, EF Core knows to suppress the creation of a *tracking snapshot*. Tracking snapshots are used for spotting changes to data, as you'll see in the example of editing the WebUrl database column in section 1.9.3. Because this query is read-only, suppressing the tracking snapshot makes the command faster.

1.9.3 Updating the database

Now you want to use the second command, update (`u`), in MyFirstEfCoreApp to update the WebUrl column in the Author table of the book *Quantum Networking*. As shown in figure 1.9, you first list all the books to show that the last book has no author URL set. Then you run the command `u`, which asks for a new author URL for the last book, *Quantum Networking*. You input a new URL of `https://entangled.moon` (it's a fictitious future book, so why not a fictitious URL!), and after the update, the command lists all the books again, showing that the author's URL has changed (with the two ovals showing you the before and after URLs).



```
Commands: l (list), u (change url) and e (exit)
> l
Refactoring by Martin Fowler
         Published on 08-Jul-1999. http://martinfowler.com/
Patterns of Enterprise Application Architecture by Martin Fowler
         Published on 15-Nov-2001. http://martinfowler.com/
Domain-Driven Design by Eric Evans
         Published on 30-Aug-2003. http://domainlanguage.com/
Quantum Networking by Future Person
         Published on 01-Jan-2057. - no web url given -
> u
New Quantum Networking WebUrl > httqs://entangled.moon
... Saved Changes called.
Refactoring by Martin Fowler
         Published on 08-Jul-1999. http://martinfowler.com/
Patterns of Enterprise Application Architecture by Martin Fowler
         Published on 15-Nov-2001. http://martinfowler.com/
Domain-Driven Design by Eric Evans
         Published on 30-Aug-2003. http://domainlanguage.com/
Quantum Networking by Future Person
         Published on 01-Jan-2057. httqs://entangled.moon
>
```

No URL set on the last book

URL set via the u command

Figure 1.9 This figure shows an update in action. The first command is `l` (list), which shows each book with the author's name and URL on the next line. Then you press `u` (update), which allows you to update the URL of the last book's author. The update command called the list command, so that you can see that your update was successful.

The code for updating the WebUrl column in the Author table linked to the book with the title *Quantum Networking* is shown here.

Listing 1.4 The code to update the author's WebUrl of the book *Quantum Networking*

```
public static void ChangeWebUrl()
{
    Console.Write("New Quantum Networking WebUrl > ");
    var newWebUrl = Console.ReadLine();

    using (var db = new AppDbContext())
    {
        var singleBook = db.Books
            .Include(book => book.Author)
            .Single(book => book.Title == "Quantum Networking");

        singleBook.Author.WebUrl = newWebUrl;
        db.SaveChanges();
        Console.WriteLine("... SavedChanges called.");
    }

    ListAll();
}
```

Reads in from the console the new URL

Loads the author information with the book

Selects only the book with the title Quantum Networking

To update the database, you change the data that was read in.

SaveChanges tells EF Core to check for any changes to the data that has been read in and write out those changes to the database.

Lists all the book information

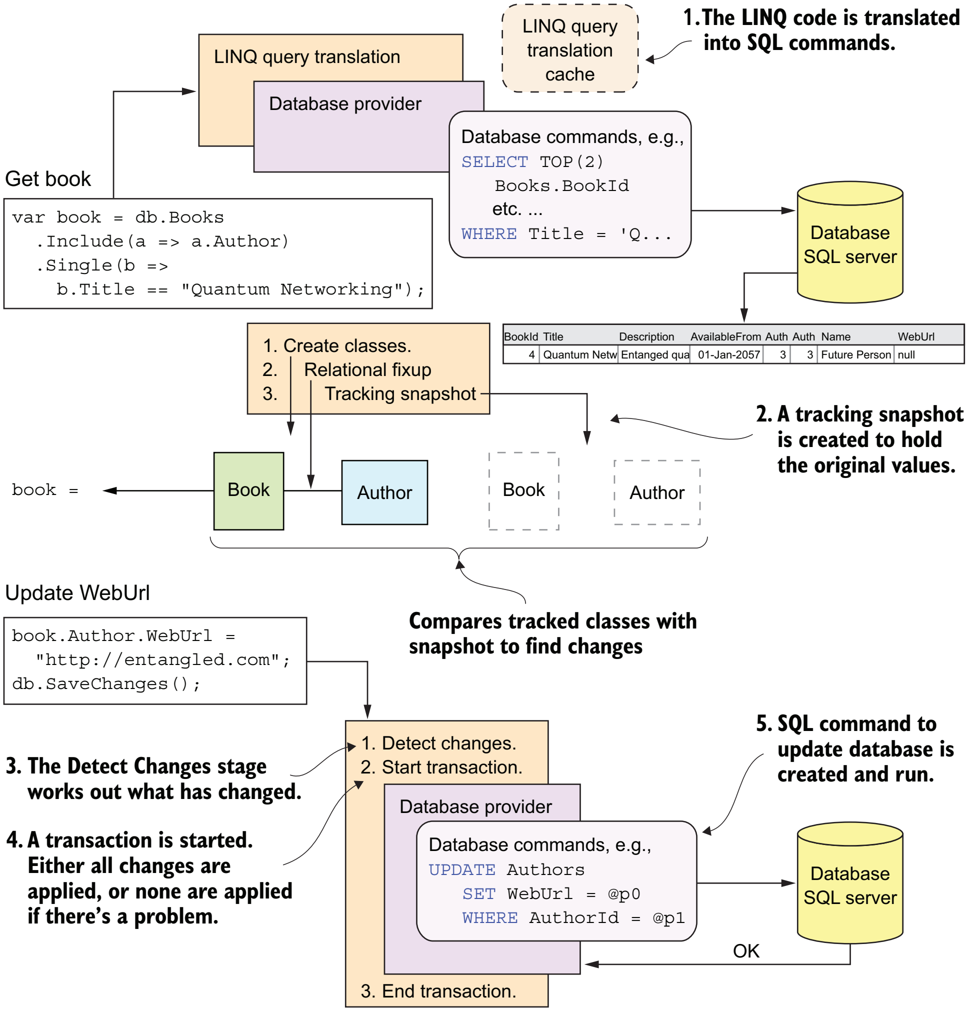
Figure 1.10 shows what is happening inside the EF Core library and follows its progress. This example is a lot more complicated than the previous read example, so let me give you some pointers on what to look for.

First, the read stage, at the top of the diagram, is similar to the read example and so should be familiar. In this case, the query loads a specific book, using the book's title as the filter. The important change is point 2: that a tracking snapshot is taken of the data.

This change occurs in the update stage, in the bottom half of the diagram. Here, you can see how EF Core compares the loaded data with the tracking snapshot to find the changes. From this data, it sees that only the `WebUrl` property has been updated, and EF Core creates an SQL command to update only the WebUrl column in the correct row of the Author table.

I've described most of the steps, but here is a blow-by-blow account of how the author's WebUrl column is updated:

1. The application uses a LINQ query to find a single book with its author information. EF Core turns the LINQ query into an SQL command to read the rows where the Title is *Quantum Networking*, returning an instance of both the `Book` and the `Author` classes, and checks that only one row was found.
2. The LINQ query doesn't include the `.AsNoTracking` method you had in the previous read versions, so the query is considered to be a *tracked query*. Therefore, EF Core creates a tracking snapshot of the data loaded.



1. The LINQ code is translated into SQL commands.

```
var book = db.Books
  .Include(a => a.Author)
  .Single(b =>
    b.Title == "Quantum Networking");
```

```
Database commands, e.g.,
SELECT TOP(2)
    Books.BookId
   etc. ...
WHERE Title = 'Q...
```

BookId	Title	Description	AvailableFrom	Auth	Auth	Name	WebUrl
4	Quantum Netw	Entanged qua	01-Jan-2057	3	3	Future Person	null

1. Create classes.
2. Relational fixup
3. Tracking snapshot

2. A tracking snapshot is created to hold the original values.

Compares tracked classes with snapshot to find changes

```
book.Author.WebUrl =
  "http://entangled.com";
db.SaveChanges();
```

1. Detect changes.
2. Start transaction.
3. End transaction.

3. The Detect Changes stage works out what has changed.

4. A transaction is started. Either all changes are applied, or none are applied if there's a problem.

5. SQL command to update database is created and run.

```
Database commands, e.g.,
UPDATE Authors
   SET WebUrl = @p0
   WHERE AuthorId = @p1
```

Figure 1.10 This figure shows what EF Core does inside when you update an `Author`'s `WebUrl` property and ask EF Core to write it to the database. This figure is quite complex, but if you start at the top and follow the numbered text, it should be easier to understand. It starts with a read to get the required `Book` and `Author`. (Note that in this process, the tracking snapshot is present; see step 2.) Then, when your code updates the `WebUrl` and calls `SaveChanges`, EF Core creates and executes the correct SQL command to update the WebUrl column in the correct row.

3. Then the code changes the `WebUrl` property in the `Author` class of the book. When `SaveChanges` is called, the Detect Changes stage compares all the classes that were returned from a tracked query with the tracking snapshot. From this, it can detect what has changed—in this case, only the `WebUrl` property of the `Author` class, which has a primary key of 3.

4. As a change is detected, EF Core starts a *transaction*. Every database update is done as an *atomic unit*: if multiple changes to the database occur, either they all succeed, or they all fail. This fact is important, because a relational database could get into a bad state if only part of an update were applied.
5. The update request is converted by the database provider to an SQL command that does the update. If the SQL command is successful, the transaction is committed, and the `SaveChanges` method returns; otherwise, an exception is raised.

1.10 The stages of development of EF Core

EF Core and .NET Core have come a long way since the first release. Over time, Microsoft has been working hard to improve the native performance of .NET Core while adding more features, to the point that .NET 5 can take over from the existing .NET Framework 4.8.

Figure 1.11 shows the history of the major releases of EF Core so far. The EF Core version numbers follow the NET Core version number. Note that the releases at the top of the figure are *long-term-support* (LTS) releases, meaning that release is supported for three years after the initial release. Major releases are expected every year, with LTS releases coming every two years.

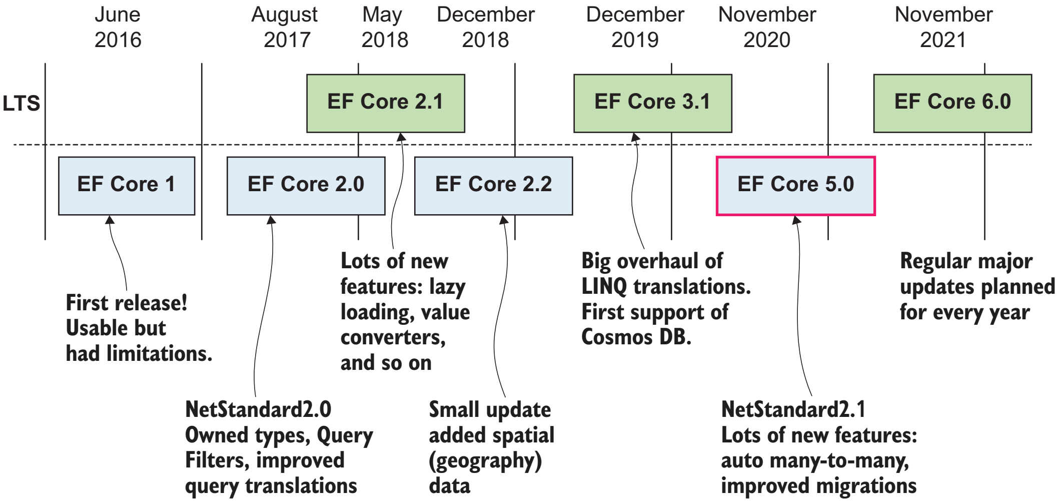


Figure 1.11 This figure depicts the development of EF Core, which runs alongside the development of the NET open source developer platform. The EF Core 5 version is highlighted because this book covers all the EF Core features up to and including EF Core 5.

1.11 Should you use EF Core in your next project?

Now that you have a quick overview of what EF Core is and how it works, the next question is whether you should start using EF Core in your project. For anyone who's planning to switch to EF Core, the key question is "Is EF Core sufficiently superior to the data access library I currently use to make it worth using for my next project?" A cost is associated with learning and adopting any new library, especially complex libraries such as EF Core, so this question is a valid one. Here is my take on EF Core and .NET Core in general.

1.11.1 .NET is the future software platform, and it's fast!

Over time, Microsoft has been working hard to improve the native performance of .NET Core while adding more features. This focus on performance has propelled Microsoft's ASP.NET Core web application from ~250th for ASP.NET MVC to around the 10th to 40th position for ASP.NET Core (depending on workload); see http://mng.bz/Gxaq. Similar but smaller performance gains have been added to EF Core.

Microsoft did say that .NET 5 would take over from the existing .NET Framework 4.8, but the COVID-19 outbreak derailed that plan a bit, and now .NET 6 will replace .NET Framework 4.8. But the writing on the wall is clear: if you're starting a new project, and .NET 5 and EF Core have the features your project needs, moving to EF Core means you aren't going to be left behind.

1.11.2 Open source and open communication

Over many years, Microsoft has transformed itself. All its .NET Core work is open source, with lots of external people getting involved with fixing bugs and adding new features, so you can have direct access to the code if you need it.

Also, the level of open communication about what is happening in .NET Core and other products is impressive. The EF Core team, for example, produces weekly updates on what it is doing, providing lots of early previews of new releases and making nightly builds of EF Core available to all. The team takes feedback seriously, and all work and defects are shown in the issue pages of the EF Core repo.

1.11.3 Multiplatform applications and development

As I said at the start of the chapter, EF Core is multiplatform-capable; you can develop and run EF Core applications on Windows, Linux, and Apple. This fact means that you can run Microsoft-based applications on cheap Linux systems. Also, developing different platforms is quite possible. In fact, Arthur Vickers, who is one of the lead engineers on the EF Core team, decided to move from Windows to Linux as his primary development platform. You can read about his experiences at http://mng.bz/zxWa.

1.11.4 Rapid development and good features

I work as a contract developer as my day job. In a typical data-driven application, I write a lot of database access code, some of it complex. With EF Core, I can write data access code really quickly, and in a way that makes access code easy to understand and refactor if it's too slow. This is the main reason I use EF Core.

At the same time, I need an O/RM that has lots of features so that I can build a database the way I want without hitting too many barriers in EF Core. Sure, some things are ruled out, such as building SQL Common Table Expressions, but a bit of raw SQL gets around things like that if I need it to.

1.11.5 Well supported

EF Core has good documentation (https://docs.microsoft.com/en-us/ef/core/index), and of course, you have this book, which brings together the documentation with deeper explanations and examples, plus patterns and practices to make you a great developer. The internet is full of blogs on EF Core, including mine at https://www.thereformedprogrammer.net. And for questions and bugs, there is always Stack Overflow; see http://mng.bz/0mDx.

The other part of support is the development tools. Microsoft seems to have changed focus by providing support for multiple platforms, but it has also created a free cross-platform development environment called VS Code. Microsoft has also made its main development tool, Visual Studio (Windows and Mac), free to individual developers and small businesses; the Usage section near the bottom of its web page at www.visualstudio.com/vs/community details the terms. That's a compelling offer.

1.11.6 Always high-performance

Ah, the database performance issue. Look, I'm not going to say that EF Core is going to, out of the box, produce blistering database access performance with beautiful SQL and fast data ingest. That's the cost you pay for quick development of your data access code; all that "magic" inside EF Core can't be as good as hand-coded SQL, but you might be surprised how good it can be. See chapter 15, where I tune up an application's performance progressively.

But you have lots of options to improve the performance of your applications. In my applications, I find that only about 5–10% of my queries are the key ones that need hand-tuning. Chapters 14 and 15 are dedicated to performance tuning, as is part of chapter 16. These chapters show that you can do a lot to improve the performance of EF Core database accesses.

But there is no reason you can't drop down to raw SQL for some of the database accesses. That's the great thing: build the application quickly by using EF Core and then convert the (few) places where EF Core isn't delivering good performance to raw SQL commands via ADO.NET or Dapper.

1.12 When should you not use EF Core?

I'm obviously pro-EF Core, but I won't use it on a client project unless using it makes sense. So let's look at a few blockers that might suggest *not* using EF Core.

The first one is obvious: Does it support the database you want to use? You can find a list of supported databases at https://docs.microsoft.com/en-us/ef/core/providers.

The second factor is the level of performance you need. If you're writing, say, a small RESTful service or Serverless system, I'm not sure that pulling in the whole of EF Core is worthwhile; you could use a fast but development-time-hungry library because there aren't many database accesses to write. But if you have a large application, with lots of boring admin accesses and a few important customer-facing accesses, a hybrid approach could work for you. (See chapter 15 for an example of a mixed EF Core/Dapper application.)

Also, EF Core isn't that good at bulk commands. Normally, tasks such as bulk-loading large amounts of data and deleting all the rows in a table can be implemented quicker by raw SQL. But several EF Core bulk CRUD extensions (some open source and some paid) can help; try searching for *EF Core bulk loading* to find possible libraries.

Summary

- EF Core is an object-relational mapper (O/RM) that uses Microsoft's Language Integrated Query (LINQ) to define database queries and return data to linked instances of .NET classes.
- EF Core is designed to make writing code for accessing a database quick and intuitive. This O/RM has plenty of features to match many requirements.
- You've seen various examples of what's happening inside EF Core. These examples will help you understand what the EF Core commands described in later chapters can do.
- There are many good reasons to consider using EF Core: it's built on a lot of experience, is well supported, and runs on multiple platforms.

For readers who are familiar with EF6.x:

- Look for EF6 notes throughout the book. These notes mark differences between the EF Core approach and EF6.x's approach. Also, check the summaries at the end of each chapter, which will point you to the major EF Core changes in that chapter.
- Think of EF Core as a new library that someone has written to mimic EF6.x but that works in a different way. That mindset will help you spot the EF Core improvements that change the way you access a database.
- EF Core no longer supports the EDMX/database designer approach that earlier forms of EF used.

2

Querying the database

This chapter covers

- Modeling three main types of database relationships
- Creating and changing a database via migration
- Defining and creating an application DbContext
- Loading related data
- Splitting complex queries into subqueries

This chapter is all about using EF Core for reading, called *querying*, the database. You'll create a database that contains the three main types of database relationships found in EF Core. Along the way, you'll learn to create and change a database's structure via EF Core.

Next, you'll learn how to access a database via EF Core, reading data from the database tables. You'll explore the basic format of EF Core queries before looking at various approaches to loading related data with the main data, such as loading the author with the book from chapter 1.

After learning the ways to load related data, you'll start to build the more complex queries needed to make a book-selling site work. This task covers sorting, filtering, and paging, plus approaches that combine these separate query commands to create one composite database query.

TIP I use unit tests to ensure that what I write in this book is correct. You might like to look at/run these unit tests, as they may help you understand what is going on. You can find them in the associated GitHub repo at http://mng.bz/XdlG. Look at the Readme file in the repo for information on where to find the unit tests and how to run them.

2.1 Setting the scene: Our book-selling site

In this chapter, you'll start building the example book-selling site, referred to as the *Book App* from now on. This example application provides a good vehicle for looking at relationships in queries. This section introduces the database, the various classes, and EF Core parts that the Book App needs to access the database.

2.1.1 The Book App's relational database

Although we could have created a database with all the data about a book, its author(s), and its reviews in one table, that wouldn't have worked well in a relational database, especially because the reviews are variable in length. The norm for relational databases is to split out any repeated data (such as the authors).

We could have arranged the various parts of the book data in the database in several ways, but for this example, the database has one of each of the main types of relationships you can have in EF Core. These three types are

- *One-to-one relationship*—PriceOffer to a Book
- *One-to-many relationship*—Book with Reviews
- *Many-to-many relationship*—Books linked to Authors and Books linked to Tags

ONE-TO-ONE RELATIONSHIP: PRICEOFFER TO A BOOK

A book can have a promotional price applied to it with an optional row in the PriceOffer, which is an example of a one-to-one relationship. (Technically, the relationship is one-to-zero-or-one, but EF Core handles it the same way.) See figure 2.1.

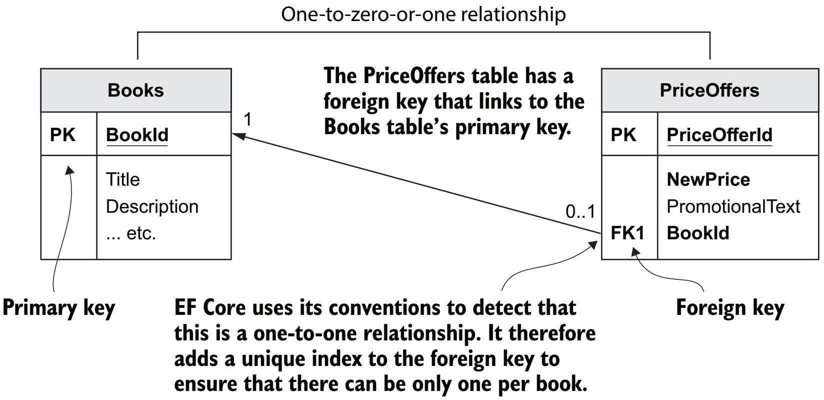


Figure 2.1 The one-to-one relationship between a Book and an optional PriceOffer. If a PriceOffer is linked to a Book, the NewPrice in the PriceOffer overrides the Price in the Book.

To calculate the final price of the book, you need to check for a row in the PriceOffer table that's linked to the Books via a foreign key. If such a row is found, the NewPrice supersedes the price for the original book, and the PromotionalText is shown onscreen, as in this example:

> *~~$40~~ $30 Our summertime price special, for this week only!*

> **ADVANCED FEATURE** In this example, I have a primary key and a foreign key to make the relationship easier to understand. But for one-to-one relationships, you can make the foreign key be the primary key too. In the PriceOffer table shown in figure 2.1, you would have a primary key, called BookId, which would also be the foreign key. As a result, you lose the PriceOfferId column, which makes the table slightly more efficient from the database side. I cover this topic later in the book, in section 8.6.1.

ONE-TO-MANY RELATIONSHIP: REVIEWS TO A BOOK

You want to allow customers to review a book; they can give a book a star rating and optionally leave a comment. Because a book may have no reviews or many (unlimited) reviews, you need to create a table to hold that data. In this example, you'll call the table Review. The Books table has a one-to-many relationship to the Review table, as shown in figure 2.2.

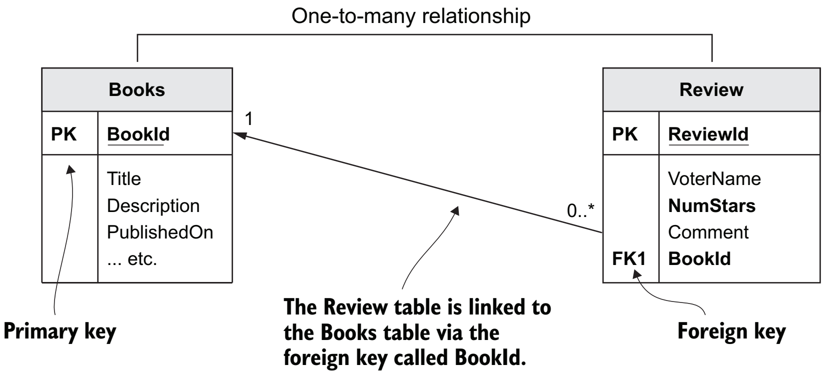


Figure 2.2 The one-to-many relationship between a Book and its zero-to-many Reviews. These Reviews work the same as they do on any e-commerce site, such as Amazon.

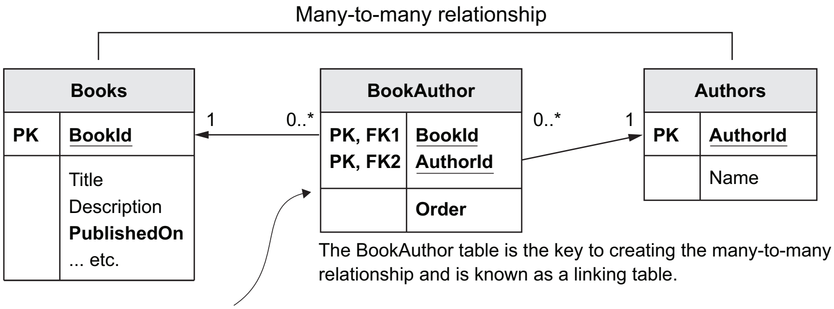
In the Summary display, you need to count the number of reviews and work out the average star rating to show a summary. Here's a typical onscreen display you might produce from this one-to-many relationship:

> *Votes 4.5 by 2 customers*

MANY-TO-MANY RELATIONSHIP: MANUALLY CONFIGURED

Books can be written by one or more authors, and an author may write one or more books. Therefore, you need a table called Books to hold the books data and another table called Authors to hold the authors. The link between the Books and Authors tables is called a *many-to-many relationship*, which in this case needs a linking table to achieve this relationship.

In this case, you create your own linking table with an Order value in it because the names of the authors in a book must be displayed in a specific order (figure 2.3).



The BookAuthor table is the key to creating the many-to-many relationship and is known as a linking table.

This table uses the foreign keys as the primary keys. Because primary keys must be unique, this ensures that only one link can exist between a book and an author.

Figure 2.3 The three tables involved in creating the many-to-many relationship between the Books table and the Authors table. I use a many-to-many relationship because books can have many authors, and authors may have written many books. The extra feature needed here is the Order value, because the order in which authors are listed in a book matters, so I use the Order value to display the authors in the correct sequence.

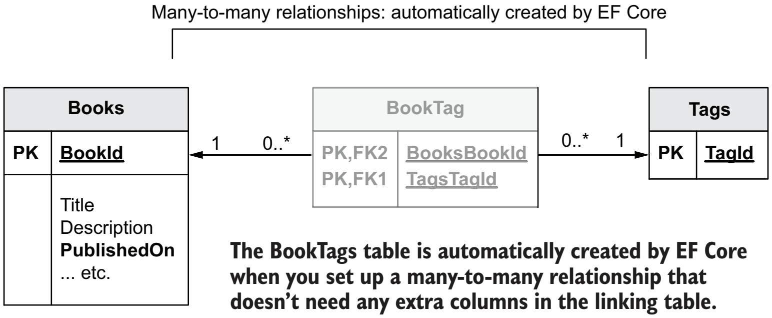
A typical onscreen display from the many-to-many relationship would look like this:

by Dino Esposito, Andrea Saltarello

MANY-TO-MANY RELATIONSHIP: AUTOCONFIGURED BY EF CORE

Books can be tagged with different categories—such as Microsoft .NET, Linux, Web, and so on—to help the customer to find a book on the topic they are interested in. A category might be applied to multiple books, and a book might have one or more categories, so a many-to-many linking table is needed. But unlike in the previous BookAuthor linking table, the tags don't have to be ordered, which makes the linking table simpler.

EF Core 5 and later can automatically create the many-to-many linking table for you. Figure 2.4 shows your database with the automatic BookTag table that provides a many-to-many link between the Books table and the Tags table. The BookTag table is grayed out to represent the fact that EF Core creates it automatically and that it isn't mapped to any of the classes you have created.



The BookTags table is automatically created by EF Core when you set up a many-to-many relationship that doesn't need any extra columns in the linking table.

This table has two primary keys, which are both unique, so that there is only one many-to-many link between a Books row and a Tags row.

Figure 2.4 The Books and Tags tables are created by you, and EF Core detects the many-to-many relationship between the Books table and the Tags table. EF Core automatically creates the linking table needed to set up the many-to-many relationships.

NOTE Chapter 8 covers the different ways to set up many-to-many relationships.

A typical onscreen display from a many-to-many relationship would look like this:

Categories: Microsoft .NET, Web

2.1.2 Other relationship types not covered in this chapter

The three types of relationships I covered in section 2.1.1 are the main relationships you will be using: one-to-one, one-to-many, and many-to-many. But EF Core does have some other variations. Here is a quick rundown on what is coming later in chapter 8:

- *Owned Type class*—Useful for adding grouped data, such as an `Address` class, to an entity class. The `Address` class is linked to the main entity, but your code can copy around the `Address` class rather than copying individual `Street`, `City`, `State`, and related properties.
- *Table splitting*—Maps multiple classes to one table. You could have a summary class with the basic properties in it and a detailed class containing all the data, for example, which would give you a quicker load of the summary data.
- *Table per hierarchy (TPH)*—Useful for groups of data that are similar. If you have a lot of data with only a few differences, such as a list of animals, you can have a base `Animal` class that `Dog`, `Cat`, and `Snake` classes can inherit, with per-type properties such as `LengthOfTail` for `Dog` and `Cat` and a `Venomous` flag for the `Snake`. EF Core maps all the classes to one table, which can be more efficient.
- *Table per type (TPT)*—Useful for groups of data that have dissimilar data. TPT, introduced in EF Core 5, is the opposite of TPH, in which each class has its own table. Following the `Animal` example for TPH, the TPT version would map the `Dog`, `Cat`, and `Snake` classes to three different tables in the database.

These four relationship patterns are built into EF Core to allow you to optimize the way you handle or store data in the database. But another relationship type doesn't need specific EF Core commands to implement: *hierarchical* data. A typical example of hierarchical data is an `Employee` class that has a relationship pointing to the employee's manager, who in turn is an employee. EF Core uses the same approaches as one-to-one and one-to-many to provide hierarchical relationships; I talk more about this type of relationship in chapters 6 and 8.

2.1.3 The database showing all the tables

Figure 2.5 shows the Book App's database that you'll be using for the examples in this chapter and in chapter 3. It contains all the tables I've described so far, including all the columns and relationships in the Books table.

NOTE The database diagram uses the same layout and terms as in chapter 1: *PK* means *primary key*, and *FK* means *foreign key*.

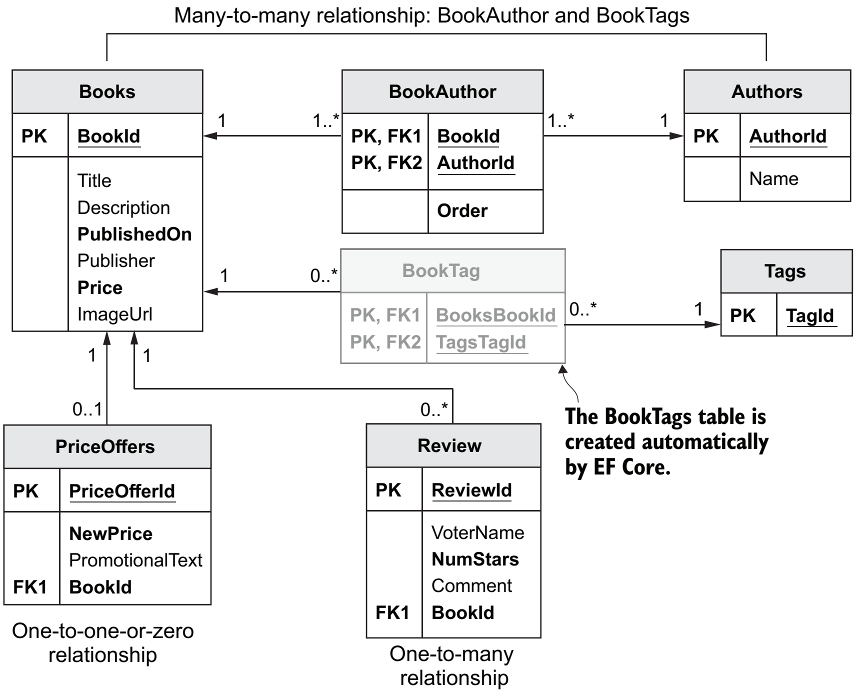
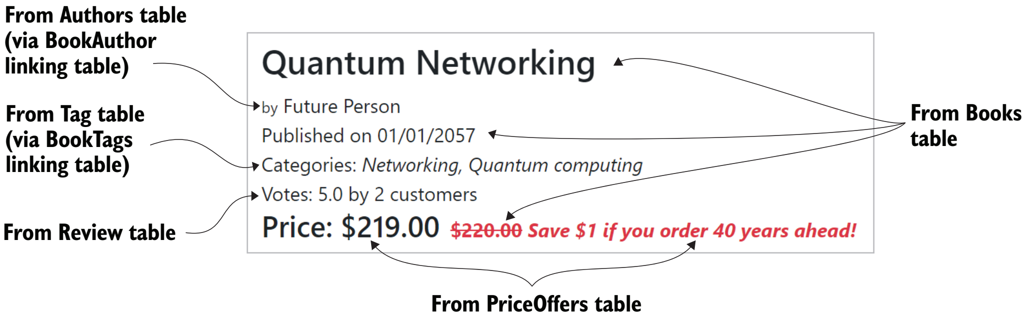


Figure 2.5 The complete relational database schema for the Book App, showing all the tables and their columns used for holding the book information. You create classes to map to all the tables you see in this figure, apart from the BookTags table (shown as grayed out). EF Core created the BookTags table automatically when it found the direct many-to-many relationship between the Books and Tags tables.



Quantum Networking

by Future Person

Published on 01/01/2057

Categories: *Networking, Quantum computing*

Votes: 5.0 by 2 customers

Price: $219.00 ~~$220.00~~ *Save $1 if you order 40 years ahead!*

Figure 2.6 A listing of a single book showing which database table provides each part of the information. As you can see, the listing requires information from all five of the database tables to create this view. In this chapter, you will build the code to produce this display, with various ordering, filtering, and paging features to make a proper e-commerce application.

To help you make sense of this database, figure 2.6 shows the onscreen output of the list of books but focuses on only one book. As you can see, the Book App needs to access every table in the database to build the book list (figure 2.10 in section 2.6). Later, I show you this same book display, but with the query that supplies each element).

> **Downloading and running the example application from the Git repo**
>
> If you want to download the Book App code and run it locally, follow the steps defined in the sidebar with the same name in section 1.6.2. The master branch contains all the code for part 1 of the book, which includes the BookApp ASP.NET Core project.

2.1.4 The classes that EF Core maps to the database

I've created five .NET classes to map to the six tables in the database. These classes are called `Book`, `PriceOffer`, `Review`, `Tag`, `Author`, and `BookAuthor` for the many-to-many-linking table, and they are referred to as *entity classes* to show that they're mapped by EF Core to the database. From the software point of view, there's nothing special about entity classes. They're normal .NET classes, sometimes referred to as *plain old CLR objects* (POCOs). The term *entity class* identifies the class as one that EF Core has mapped to the database.

The primary entity class is the `Book` class, shown in the following listing. You can see that it refers to a single `PriceOffer` class, a collection of `Review` class instances, a collection of `Tag` class instances, and finally a collection of `BookAuthor` classes that links the book data to one or more `Author` classes containing the author's name.

Listing 2.1 The `Book` class, mapped to the Books table in the database

```
public class Book
{
```

The Book class contains the main book information.

```
    public int BookId { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public DateTime PublishedOn { get; set; }
    public string Publisher { get; set; }
    public decimal Price { get; set; }
    public string ImageUrl { get; set; }

    //----------------------------------------------
    //relationships

    public PriceOffer Promotion { get; set; }
    public ICollection<Review> Reviews { get; set; }
    public ICollection<Tag> Tags { get; set; }
    public ICollection<BookAuthor>
        AuthorsLink { get; set; }
}
```

We use EF Core's By Convention configuration to define the primary key of this entity class, so we use <ClassName>Id, and because the property is of type int, EF Core assumes that the database will use the SQL IDENTITY command to create a unique key when a new row is added.

Link to the optional one-to-one PriceOffer relationship.

There can be zero to many reviews of the book.

EF Core 5's automatic many-to-many relationship to the Tag entity class

Provides a link to the many-to-many linking table that links to the Authors of this book

> **NOTE** In part 1, the entity classes use the default (empty) constructor. If you want to create specific constructors for any of your entity classes, you should be aware that EF Core may use your constructor when reading and creating an instance of an entity class. I cover this topic in section 6.1.11.

For simplicity, we use EF Core's By Convention configuration approach to model the database. We use By Convention naming for the properties that hold the primary key and foreign keys in each of the entity classes. In addition, the .NET type of the navigational properties, such as `ICollection<Review> Reviews`, defines what sort of relationship we want. Because the `Reviews` property is of the .NET type `ICollection<Review>`, for example, the relationship is a one-to-many relationship. Chapters 7 and 8 describe the other approaches for configuring the EF Core database model.

> **ADVANCED NOTE** In the Book App, when I have navigational properties that are collections, I use the type `ICollection<T>`. I do so because the new eager loading sort capability (see section 2.4.1) can return a sorted collection, and the default `HashSet` definition says it holds only a collection "whose elements are in no particular order." But there is a performance cost to not using `HashSet` when your navigational properties contain a large collection. I cover this issue in chapter 14.

What happens if you want to access an existing database?

The examples in this book show how to define and create a database via EF Core because the most complex situation is when you need to understand all the configuration options. But accessing an existing database is much easier, because EF Core can build your application's DbContext class and all your entity classes for you, using a feature called *reverse engineering,* which is covered in section 9.7.

The other possibility is that you don't want EF Core to change the database structure, but want to look after that task yourself, such as via an SQL change script or a database deployment tool. I cover that approach in section 9.6.2.

2.2 Creating the application's DbContext

To access the database, you need to do the following:

1. Define your application's DbContext, which you do by creating a class and inheriting from EF Core's `DbContext` class.
2. Create an instance of that class every time you want to access the database.

All the database queries you'll see later in this chapter use these steps, which I describe in detail in the following sections.

2.2.1 Defining the application's DbContext: EfCoreContext

The key class you need to use EF Core is the application's DbContext. You define this class by inheriting EF Core's `DbContext` class and adding various properties to allow your software to access the database tables. It also contains methods you can override to access other features in EF Core, such as configuring the database modeling. Figure 2.7 gives you an overview of the Book App's DbContext, pointing out all the important parts.

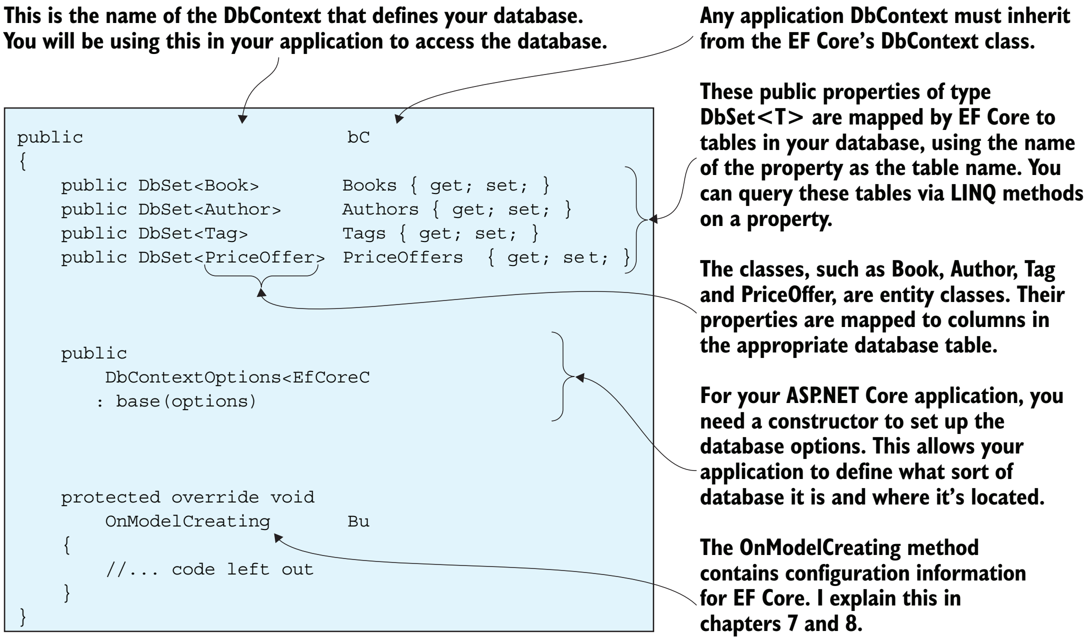
One point to note about figure 2.7 is that the Book App's DbContext doesn't include `DbSet<T>` properties for your `Review` entity class and the `BookAuthor` linking entity class. In the book app, both entity classes are accessed not directly, but via the `Book` class navigational properties, as you'll see in section 2.4.

NOTE I skip configuring the database modeling, which is done in the `OnModelCreating` method of the application's DbContext. Chapters 7 and 8 cover how to model the database in detail.

2.2.2 Creating an instance of the application's DbContext

Chapter 1 showed you how to set up the application's DbContext by overriding its `OnConfiguring` method. The downside of that approach is that the connection string is fixed. In this chapter, you'll use another approach, because you'd want to use a different database for development and unit testing. You'll use a method that provides that database via the application's DbContext constructor.

Listing 2.2 provides the options for the database at the time you create the application DbContext, called `EfCoreContext`. To be honest, this listing is based on what I use in the unit-testing chapter (chapter 17), because it has the benefit of showing you each step of creating an instance of the application's DbContext. Chapter 5, which is about using EF Core in an ASP.NET Core application, presents a more powerful way to create the application's DbContext, using a feature called dependency injection.



```
public                         bC
{
    public DbSet<Book>         Books { get; set; }
    public DbSet<Author>       Authors { get; set; }
    public DbSet<Tag>          Tags { get; set; }
    public DbSet<PriceOffer>   PriceOffers  { get; set; }



    public
        DbContextOptions<EfCoreC
       : base(options)



    protected override void
        OnModelCreating        Bu
    {
        //... code left out
    }
}
```

This is the name of the DbContext that defines your database. You will be using this in your application to access the database.

Any application DbContext must inherit from the EF Core's DbContext class.

These public properties of type DbSet<T> are mapped by EF Core to tables in your database, using the name of the property as the table name. You can query these tables via LINQ methods on a property.

The classes, such as Book, Author, Tag and PriceOffer, are entity classes. Their properties are mapped to columns in the appropriate database table.

For your ASP.NET Core application, you need a constructor to set up the database options. This allows your application to define what sort of database it is and where it's located.

The OnModelCreating method contains configuration information for EF Core. I explain this in chapters 7 and 8.

Figure 2.7 The application's DbContext is the key class in accessing the database. This figure shows the main parts of an application's DbContext, starting with its inheriting EF Core's DbContext, which brings in lots of code and features. You have to add some properties with the class `DbSet<T>` that map your classes to a database table with the same name as the property name you use. The other parts are the constructor, which handles setting up the database options, and the `OnModelCreating` method, which you can override to add your own configuration commands and set up the database the way you want.

Listing 2.2 Creating an instance of the application's DbContext to access the database

```
const string connection =
     "Data Source=(localdb)\\mssqllocaldb;"+
     "Database=EfCoreInActionDb.Chapter02;"+
     "Integrated Security=True;";
var optionsBuilder =
    new DbContextOptionsBuilder
           <EfCoreContext>();

optionsBuilder.UseSqlServer(connection);
var options = optionsBuilder.Options;

using (var context = new EfCoreContext(options))
{

    var bookCount = context.Books.Count();
    //... etc.
```

The connection string, with its format dictated by the sort of database provider and hosting you're using

You need an EF Core DbContextOptionsBuilder<> instance to set the options you need.

You're accessing an SQL Server database and using the UseSqlServer method from the Microsoft.EntityFrameworkCore.SqlServer library, and this method needs the database connection string.

Creates the all-important EfCoreContext, using the options you've set up. You use a using statement because the DbContext is disposable.

Uses the DbContext to find out the number of books in the database

At the end of this listing, you create an instance of `EfCoreContext` inside a `using` statement because DbContext has an `IDisposable` interface and therefore should be disposed after you've used it. So from now on, if you see a variable called `context`, it was created by using the code in listing 2.2 or a similar approach.

2.2.3 Creating a database for your own application

You have a few ways to create a database using EF Core, but the normal way is to use EF Core's migration feature. This feature uses your application's DbContext and the entity classes, like the ones I've described, as the model for the database structure. The `Add-Migration` command first models your database and then, using that model, builds commands to create a database that fits that model.

> **TIP** If you have cloned the Git repo that goes with this book (http://mng.bz/XdlG), you can see what a migration looks like by looking at the Migration folder in the DataLayer project. Also, all the correct NuGet packages are added to the DataLayer and BookApp projects to allow migrations to be created and applied to an SQL Server database.

The great thing about migrations, besides handling database creation, is the fact that they can update the database with any changes you make in the software. If you change your entity classes or any of your application's DbContext configuration, the `Add-Migration` command will build a set of commands to update the existing database. Here are the steps you need to go through to add a migration and create or migrate a database. This process is based on a ASP.NET Core application (see chapter 5 for more on ASP.NET Core) with your DbContext in a separate project and on developing with Visual Studio. (I cover other options in chapter 9.)

1. The project that contains your DbContext needs the NuGet package Microsoft.EntityFrameworkCore.SqlServer or another database provider if you are using a different database.
2. The ASP.NET Core project needs the following NuGet packages:
   a. Microsoft.EntityFrameworkCore.SqlServer (or same database provider as in step 1)
   b. Microsoft.EntityFrameworkCore.Tools
3. The ASP.NET Core's `Startup` class contains the commands to add an EF Core database provider, and the appsettings.json file contains the connection string for the database you want to create/migrate. (EF Core uses the ASP.NET Core's `CreateHostBuilder(args).Build()` methods to obtain a valid instance of your DbContext.)
4. In Visual Studio, open the Package Manager Console (PMC) by choosing Tools > NuGet Package Manager > Package Manager Console.
5. In the PMC window, make sure that the default project is your ASP.NET Core project.

6. In PMC, run the command `Add-Migration MyMigrationName -Project DataLayer`. This command creates a set of classes that migrate the database from its current state to a state that matches your application's DbContext and the entity classes at the time that you run your command. (The `MyMigrationName` shown in the command is the name that will be used for the migration.)
7. Run the command `Update-Database` to apply the commands created by the `Add-Migration` command to your database. If no database exists, `Update-Database` will create one. If a database exists, the command checks whether that database has this database migration applied to it, and if any database migrations are missing, this command applies them to the database. (See chapter 9 for more on migration commands.)

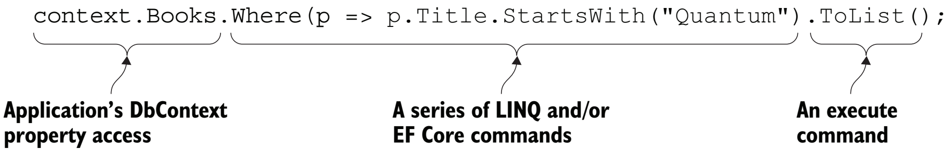
NOTE You can also use EF Core's .NET Core command-line interface (CLI) to run these commands (see http://mng.bz/454w). Chapter 9 lists both the Visual Studio and CLI versions of the migration commands.

An alternative to using the `Update-Database` command is to call the `context.Database.Migrate` method in the startup code of your application. This approach is especially useful for an ASP.NET Core web application that's hosted; chapter 5 covers this option, including some of its limitations.

NOTE Chapter 9 provides a detailed look at EF Core's migrations feature as well as other ways to alter the structure of your database (referred to as the database's *schema*).

2.3 Understanding database queries

Now you can start looking at how to query a database by using EF Core. Figure 2.8 shows an example EF Core database query, with the three main parts of the query highlighted.



```
context.Books.Where(p => p.Title.StartsWith("Quantum")).ToList();
```

Figure 2.8 The three parts of an EF Core database query, with example code. You will become familiar with this type of LINQ statement, which is the basic building block of all queries.

TIME-SAVER If you're familiar with EF and/or LINQ, you can skip this section.

The command shown in figure 2.8 consists of several methods, one after the other. This structure is known as a *fluent interface*. Fluent interfaces like this one flow logically and intuitively, which makes them easy to read. The three parts of this command are described in the following sections.

NOTE The LINQ command in figure 2.8 is known as the LINQ method, or lambda syntax. You can use another format for writing LINQ commands with EF Core: the query syntax. I describe the two LINQ syntaxes in appendix A.

2.3.1 Application's DbContext property access

The first part of the command is connected to the database via EF Corea. The most common way to refer to a database table is via a `DbSet<T>` property in the application's DbContext, as shown in figure 2.7.

You'll use this DbContext property access throughout this chapter, but later chapters introduce other ways to get to a class or property. The basic idea is the same: you need to start with something that's connected to the database via EF Core.

2.3.2 A series of LINQ/EF Core commands

The major part of a command is a set of LINQ and/or EF Core methods that create the type of query you need. The LINQ query can range from being super-simple to quite complicated. This chapter starts with simple examples of queries, but by the end of this chapter, you'll learn how to build complex queries.

NOTE *Learning LINQ will be essential to you, as EF Core uses LINQ commands for database accesses.* The appendix gives you a brief overview of LINQ. Plenty of online resources are available too; see http://mng.bz/j4Qx.

2.3.3 The execute command

The last part of the command reveals something about LINQ. Until a final execute command is applied at the end of the sequence of LINQ commands, the LINQ is held as a series of commands in what is called an *expression tree* (see section A.2.2), which means that it hasn't been executed on the data yet. EF Core can translate an expression tree into the correct commands for the database you're using. In EF Core, a query is executed against the database when

- It's enumerated by a `foreach` statement.
- It's enumerated by a collection operation such as `ToArray`, `ToDictionary`, `ToList`, `ToListAsync`, and so forth.
- LINQ operators such as `First` or `Any` are specified in the outermost part of the query.

You'll use certain EF Core commands, such as `Load`, in the explicit loading of a relationship later in this chapter.

At this point, your LINQ query will be converted to database commands and sent to the database. If you want to build high-performance database queries, you want all your LINQ commands for filtering, sorting, paging, and so on to come before you call an execute command. Therefore, your filter, sort, and other LINQ commands will be run inside the database, which improves the performance of your query. You will see this approach in action in section 2.8, when you build a query to filter, sort, and page the books in the database to display to your user.

2.3.4 The two types of database queries

The database query in figure 2.8 is what I call a *normal* query, also known as a *read-write* query. This query reads in data from the database in such a way that you can update that data (see chapter 3) or use it as an existing relationship for a new entry, such as creating a new book with an existing Author (see section 6.2.2).

The other type of query is an AsNoTracking query, also known as a read-only query. This query has the EF Core's `AsNoTracking` method added to the LINQ query (see the following code snippet). As well as making the query read-only, the `AsNoTracking` method improves the performance of the query by turning off certain EF Core features; see section 6.12 for more information:

```
context.Books.AsNoTracking()
       .Where(p => p.Title.StartsWith("Quantum")).ToList();
```

NOTE Section 6.1.2 provides a detailed list of the differences between the normal, read-write query and the AsNoTracking, read-only query.

2.4 Loading related data

I've shown you the `Book` entity class, which has links to three other entity classes: `PriceOffer`, `Review`, and `BookAuthor`. Now I want to explain how you, as a developer, can access the data behind these relationships. You can load data in four ways: eager loading, explicit loading, select loading, and lazy loading. Before I cover these approaches, however, you need to be aware that EF Core won't load any relationships in an entity class unless you ask it to. If you load a `Book` class, each of the relationship properties in the `Book` entity class (`Promotion`, `Reviews`, and `AuthorsLink`) will be `null` by default.

This default behavior of not loading relationships is correct, because it means that EF Core minimizes the database accesses. If you want to load a relationship, you need to add code to tell EF Core to do that. The following sections describe the four approaches that get EF Core to load a relationship.

2.4.1 Eager loading: Loading relationships with the primary entity class

The first approach to loading related data is *eager loading*, which entails telling EF Core to load the relationship in the same query that loads the primary entity class. Eager loading is specified via two fluent methods, `Include` and `ThenInclude`. The next listing shows the loading of the first row of the Books table as an instance of the `Book` entity class and the eager loading of the single relationship, `Reviews`.

Listing 2.3 Eager loading of first book with the corresponding `Reviews` relationship

```
var firstBook = context.Books
    .Include(book => book.Reviews)
    .FirstOrDefault();
```

Gets a collection of Review class instances, which may be an empty collection

Takes the first book or null if there are no books in the database

If you look at the SQL command that this EF Core query creates, shown in the following snippet, you'll see two SQL commands. The first command loads the first row in the Books table. The second loads the reviews, where the foreign key, `BookId`, has the same value as the first `Books` row primary key:

```
SELECT "t"."BookId", "t"."Description", "t"."ImageUrl",
    "t"."Price", "t"."PublishedOn", "t"."Publisher",
       "t"."Title", "r"."ReviewId", "r"."BookId",
            "r"."Comment", "r"."NumStars", "r"."VoterName"
FROM (
    SELECT "b"."BookId", "b"."Description", "b"."ImageUrl",
        "b"."Price", "b"."PublishedOn", "b"."Publisher", "b"."Title"
    FROM "Books" AS "b"
    LIMIT 1
) AS "t"
LEFT JOIN "Review" AS "r" ON "t"."BookId" = "r"."BookId"
ORDER BY "t"."BookId", "r"."ReviewId"
```

Now let's look at a more complex example. The following listing shows a query to get the first `Book`, with eager loading of all its relationships—in this case, `AuthorsLink` and the second-level `Author` table, the `Reviews`, and the optional `Promotion` class.

Listing 2.4 Eager loading of the `Book` class and all the related data

The first Include gets a collection of BookAuthor.

Gets the next link—in this case, the link to the author

Gets a collection of Review class instances, which may be an empty collection

```
var firstBook = context.Books
    .Include(book => book.AuthorsLink)
        .ThenInclude(bookAuthor => bookAuthor.Author)
    .Include(book => book.Reviews)
    .Include(book => book.Tags)
    .Include(book => book.Promotion)
    .FirstOrDefault();
```

Loads and directly accesses the Tags

Takes the first book, or null if there are no books in the database

Loads any optional PriceOffer class, if one is assigned

The listing shows the use of the eager-loading method `Include` to get the `AuthorsLink` relationship. This relationship is a first-level relationship, referred to directly from the entity class you're loading. That `Include` is followed by `ThenInclude` to load the second-level relationship—in this case, the `Author` table at the other end of the linking table, `BookAuthor`. This pattern, `Include` followed by `ThenInclude`, is the standard way of accessing relationships that go deeper than a first-level relationship. You can go to any depth with multiple `ThenIncludes`, one after the other.

If you use the direct linking of many-to-many relationships introduced in EF Core 5, you don't need `ThenInclude` to load the second-level relationship because the property directly accesses the other end of the many-to-many relationship via the `Tags` property, which is of type `ICollection<Tag>`. This approach can simplify the use of a many-to-many relationship as long you don't need some data in the linking table, such as the `Order` property in the `BookAuthor` linking entity class used to order the `Book`'s `Authors` correctly.

> **EF6** Eager loading in EF Core is similar to that in EF6.x, but EF6.x doesn't have a `ThenInclude` method. As a result, the `Include`/`ThenInclude` code used in listing 2.4 would be written in EF6.x as `context.Books.Include(book => book.AuthorLink.Select(bookAuthor => bookAuthor.Author))`.

If the relationship doesn't exist (such as the optional `PriceOffer` class pointed to by the Promotion property in the `Book` class), `Include` doesn't fail; it simply doesn't load anything, or in the case of collections, it returns an empty collection (a valid collection with zero entries). The same rule applies to `ThenInclude`: if the previous `Include` or `ThenInclude` was empty, subsequent `ThenIncludes` are ignored. If you don't `Include` a collection, it is null by default.

The advantage of eager loading is that EF Core will load all the data referred to by the `Include` and `ThenInclude` in an efficient manner, using a minimum of database accesses, or *database round-trips*. I find this type of loading to be useful in relational updates in which I need to update an existing relationship; chapter 3 covers this topic. I also find eager loading to be useful in business logic; chapter 4 covers this topic in much more detail.

The downside is that eager loading loads *all* the data, even when you don't need part of it. The book list display, for example, doesn't need the book description, which could be quite large.

SORTING AND FILTERING WHEN USING INCLUDE AND/OR THENINCLUDE

EF Core 5 added the ability to sort or filter the related entities when you use the `Include` or `ThenInclude` methods. This capability is helpful if you want to load only a subset of the related data (such as only `Reviews` with five stars) and/or to order the included entities (such as ordering the `AuthorsLink` collection against the `Order` property). The only LINQ commands you can use in the `Include` or `ThenInclude` methods are `Where`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `Skip`, and `Take`, but those commands are all you need for sorting and filtering.

The next listing shows the same code as listing 2.4, but with the `AuthorsLink` collection being sorted on the `Order` property and with the `Reviews` collection being filtered to load only `Reviews` in which `NumStars` is 5.

Listing 2.5 Sorting and filtering when using `Include` or `ThenInclude`

```
var firstBook = context.Books
    .Include(book => book.AuthorsLink
        .OrderBy(bookAuthor => bookAuthor.Order))
        .ThenInclude(bookAuthor => bookAuthor.Author)
    .Include(book => book.Reviews
        .Where(review => review.NumStars == 5))
    .Include(book => book.Promotion)
    .First();
```

Sort example: On the eager loading of the AuthorsLink collection, you sort the BookAuthors so that the Authors will be in the correct order to display.

Filter example. Here, you load only the Reviews with a star rating of 5.

2.4.2 *Explicit loading: Loading relationships after the primary entity class*

The second approach to loading data is *explicit loading*. After you've loaded the primary entity class, you can explicitly load any other relationships you want. Listing 2.6 does the same job as listing 2.4 with explicit loading. First, it loads the `Book`; then it uses explicit-loading commands to read all the relationships.

Listing 2.6 Explicit loading of the `Book` class and related data

```
var firstBook = context.Books.First();
context.Entry(firstBook)
    .Collection(book => book.AuthorsLink).Load();
foreach (var authorLink in firstBook.AuthorsLink)
{
    context.Entry(authorLink)
        .Reference(bookAuthor =>
            bookAuthor.Author).Load();
}

context.Entry(firstBook)
    .Collection(book => book.Tags).Load();
context.Entry(firstBook)
    .Reference(book => book.Promotion).Load();
```

Reads in the first book on its own

Explicity loads the linking table, BookAuthor

To load all the possible authors, the code has to loop through all the BookAuthor entries…

…and load each linked Author class.

Loads all the reviews

Loads the Tags

Loads the optional PriceOffer class

Alternatively, explicit loading can be used to apply a query to the relationship instead of loading the relationship. Listing 2.7 shows the use of the explicit-loading method `Query` to obtain the count of reviews and to load the star ratings of each review. You can use any standard LINQ command after the `Query` method, such as `Where` or `OrderBy`.

Listing 2.7 Explicit loading of the `Book` class with a refined set of related data

```
var firstBook = context.Books.First();
var numReviews = context.Entry(firstBook)
    .Collection(book => book.Reviews)
    .Query().Count();
```

Reads in the first book on its own

Executes a query to count reviews for this book

```
var starRatings = context.Entry(firstBook)
    .Collection(book => book.Reviews)
    .Query().Select(review => review.NumStars)
    .ToList();
```

Executes a query to get all the star ratings for the book

The advantage of explicit loading is that you can load a relationship of an entity class later. I've found this technique useful when I'm using a library that loads only the primary entity class, and need one of its relationships. Explicit loading can also be useful when you need that related data in only some circumstances. You might also find explicit loading to be useful in complex business logic because you can leave the job of loading the specific relationships to the parts of the business logic that need it.

The downside of explicit loading is more database round trips, which can be inefficient. If you know up front the data you need, eager loading the data is usually more efficient because it takes fewer database round trips to load the relationships.

2.4.3 Select loading: Loading specific parts of primary entity class and any relationships

The third approach to loading data is using the LINQ `Select` method to pick out the data you want, which I call *select loading*. The next listing shows the use of the `Select` method to select a few standard properties from the `Book` class and execute specific code inside the query to get the count of customer reviews for this book.

Listing 2.8 `Select` of the `Book` class picking specific properties and one calculation

```
var books = context.Books
    .Select(book => new
        {
            book.Title,
            book.Price,
            NumReviews
                = book.Reviews.Count,
        }
    ).ToList();
```

Uses the LINQ Select keyword and creates an anonymous type to hold the results

Simple copies of a couple of properties

Runs a query that counts the number of reviews

The advantage of this approach is that only the data you need is loaded, which can be more efficient if you don't need all the data. For listing 2.8, only one SQL `SELECT` command is required to get all that data, which is also efficient in terms of database round trips. EF Core turns the `p.Reviews.Count` part of the query into an SQL command, so that count is done inside the database, as you can see in the following snippet of the SQL created by EF Core:

```
SELECT "b"."Title", "b"."Price", (
    SELECT COUNT(*)
    FROM "Review" AS "r"
    WHERE "b"."BookId" = "r"."BookId") AS "NumReviews"
FROM "Books" AS "b"
```

The downside to the select-loading approach is that you need to write code for each property/calculation you want. In section 7.15.4, I show a way to automate this process.

NOTE Section 2.6 contains a much more complex select-loading example, which you'll use to build the high-performance book list query for the Book App.

2.4.4 Lazy loading: Loading relationships as required

Lazy loading makes writing queries easy, but it has a bad effect on database performance. Lazy loading does require some changes to your DbContext or your entity classes, but after you make those changes, reading is easy; if you access a navigational property that isn't loaded, EF Core will execute a database query to load that navigational property.

You can set up lazy loading in either of two ways:

- Adding the Microsoft.EntityFrameworkCore.Proxies library when configuring your DbContext
- Injecting a lazy loading method into the entity class via its constructor

The first option is simple but locks you into setting up lazy loading for all the relationships. The second option requires you to write more code but allows you to pick which relationships use lazy loading. I'm going to explain only the first option in this chapter because it is simple, and leave the second option for chapter 6 (section 6.1.10) because it uses concepts that I haven't covered yet, such as dependency injection.

NOTE If you want to see all the lazy-loading options now, access Microsoft's EF Core documentation at https://docs.microsoft.com/en-us/ef/core/querying/related-data/lazy.

To configure the simple lazy loading approach, you must do two things:

- Add the keyword `virtual` before *every* property that is a relationship.
- Add the method `UseLazyLoadingProxies` when setting up your DbContext.

So the converted Book entity type to the simple lazy loading approach would look like the following code snippet, with the virtual keyword added to the navigational properties:

```
public class BookLazy
{
    public int BookLazyId { get; set; }
    //… Other properties left out for clarity

    public virtual PriceOffer Promotion { get; set; }
    public virtual ICollection<Review> Reviews { get; set; }
    public virtual ICollection<BookAuthor> AuthorsLink { get; set; }
}
```

Using the EF Core's Proxy library has a limitation: you must make every relational property virtual; otherwise, EF Core will throw an exception when you use the DbContext.

The second part is adding the EF Core's Proxy library to the application that sets up the DbContext and then adding the `UseLazyLoadingProxies` to the configuring of the DbContext. The following code snippet shows the added method to the DbContext shown in listing 2.2 (`UseLazyLoadingProxies`):

```
var optionsBuilder =
    new DbContextOptionsBuilder<EfCoreContext>();
optionsBuilder
    .UseLazyLoadingProxies()
    .UseSqlServer(connection);
var options = optionsBuilder.Options;

using (var context = new EfCoreContext(options))
```

When you have configured lazy loading in your entity classes and in the way you create the DbContext, reading relationships is simple; you don't need extra `Include` methods in your query because the data is loaded from the database when your code accesses that relationship property. Listing 2.9 shows the lazy loading of the `Book`'s `Reviews` property.

Listing 2.9 Lazy loading of `BookLazy`'s `Reviews` navigational property

Gets an instance of the BookLazy entity class that has configured its Reviews property to use lazy loading

```
var book = context.BookLazy.Single();
var reviews = book.Reviews.ToList();
```

When the Reviews property is accessed, EF Core will read in the reviews from the database.

Listing 2.9 creates two database accesses. The first access loads the `BookLazy` data without any properties, and the second happens when you access `BookLazy`'s `Reviews` property.

Many developers find lazy loading to be useful, but I avoid it because of its performance issues. There is time overhead for every access to the database server, so the best approach is to minimize the number of calls to the database server. But lazy loading (and explicit loading) can create lots of database accesses, making the query slow and causing the database server to work harder. See section 14.5.1 for a side-by-side comparison of the four types of loading of related data.

TIP Even if you have set up a relational property for lazy loading, you can get better performance by adding an `Include` on a virtual relational property. The lazy loading will see that the property has been loaded and not load it again. Changing the first line of listing 2.9 to `context.BookLazy.Include(book => book.Reviews).Single()`, for example, would reduce the two database accesses to one access.

2.5 Using client vs. server evaluation: Adapting data at the last stage of a query

All the queries you've seen so far are ones that EF Core can convert to commands that can be run on the database server. But EF Core has a feature called *client vs. server evaluation*, which allows you to run code at the last stage of the query (that is, the final `Select` part in your query) that can't be converted to database commands. EF Core runs these non-server-runnable commands after the data has come back from the database.

EF6 Client vs. server evaluation is a new feature in EF Core, and a useful one too.

The client vs. server evaluation feature gives you the opportunity to adapt/change the data within the last part of the query, which can save you from having to apply an extra step after the query. In section 2.6, you use client vs. server evaluation to create a comma-delimited list of the authors of a book. If you didn't use client vs. server evaluation for that task, you would need to (a) send back a list of all the Author names and (b) add an extra step after the query, using a `foreach` section to apply a `string.Join` to each book's authors.

Warning: EF Core will throw an exception if it cannot convert your LINQ

Before EF Core 3, any LINQ that couldn't be translated to a database command would be run in software using the client vs. server evaluation. In some cases, this approach would produce extremely poor-performing queries. (I wrote about this subject in the first edition of this book.) EF Core 3 changed this situation so that the client vs. server evaluation is used in only the final stage of your LINQ queries, stopping client vs. server evaluation from producing poor-performing queries.

But that change creates a different problem: if your LINQ queries can't be converted to database commands, EF Core will throw an `InvalidOperationException`, with a message containing the words `could not be translated`. The trouble is that you get that error only when you try that query—and you don't want that error to happen in production!

Throughout this book, I will guide you to write queries that will work, but with complex queries, it's easy to get something not quite right in your LINQ, causing the `InvalidOperationException` to be thrown. This still happens to me, even though I know EF Core well, which is why I recommend in chapter 17 that you unit-test your database accesses with a real database and/or have a set of integration tests.

For the list display of the books in the Book App, you need to (a) extract all the authors' names, in order, from the Authors table and (b) turn them into one string with commas between names. Here's an example that loads two properties, `BookId` and `Title`, in the normal manner, and a third property, `AuthorsString`, that uses client vs. server evaluation.

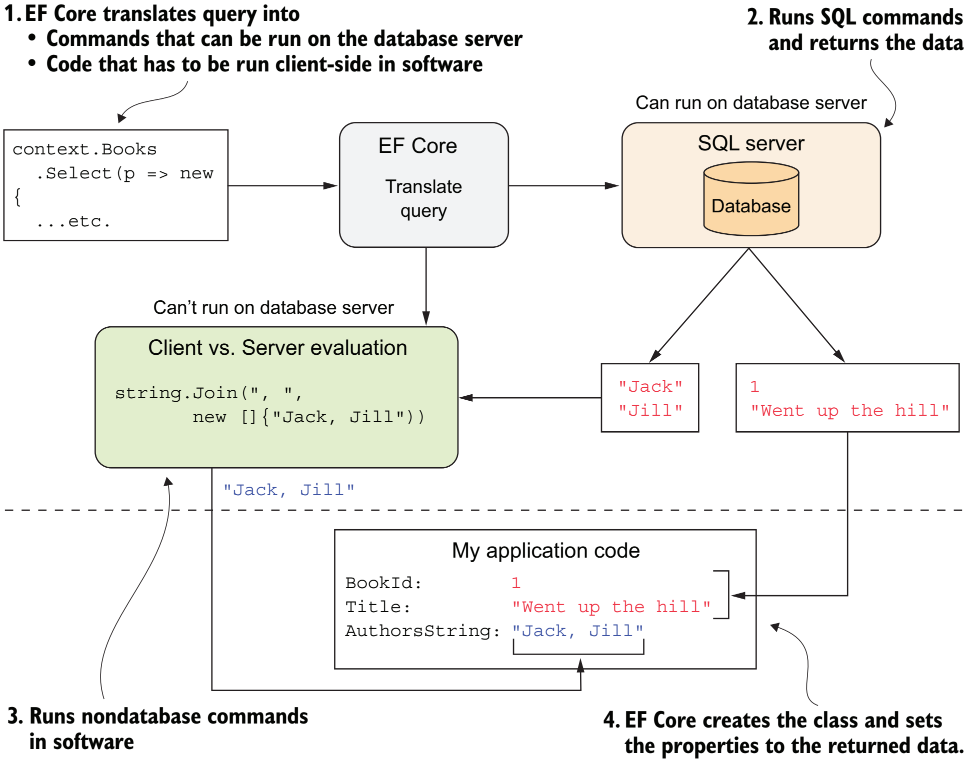
Listing 2.10 `Select` query that includes a non-SQL command, `string.Join`

```
var firstBook = context.Books
    .Select(book => new
    {
        book.BookId,
        book.Title,
        AuthorsString = string.Join(", ",
            book.AuthorsLink
            .OrderBy(ba => ba.Order)
            .Select(ba => ba.Author.Name))
    }
    ).First();
```

string.Join is executed on the client in software.

These parts of the select can be converted to SQL and run on the server.

Running this code on a book that has two authors, Jack and Jill, would cause `AuthorsString` to contain Jack, Jill, and the `BookId`, and `Title` would be set to the value of the corresponding columns in the Books table. Figure 2.9 shows how listing 2.10 would be processed through four stages. I want to focus on stage 3, where EF Core runs the client-side code that it couldn't convert to SQL.



```
context.Books
  .Select(p => new
{
  ...etc.
```

```
string.Join(", ",
        new []{"Jack, Jill"))
```

```
"Jack"
"Jill"
```

```
1
"Went up the hill"
```

```
"Jack, Jill"
```

```
BookId:         1
Title:          "Went up the hill"
AuthorsString:  "Jack, Jill"
```

Figure 2.9 Some parts of the query are converted to SQL and run in SQL Server; another part, `string.Join`, has to be done client-side by EF Core before the combined result is handed back to the application code.

The example in listing 2.10 is fairly simple, but you need to be careful how you use a property created by client vs. server evaluation. Using client vs. server evaluation on a property means that you cannot use that property in any LINQ command that would produce database commands, such as any commands that sort or filter that property. If you do, you will get an `InvalidOperationException`, with a message that contains the words `could not be translated`. In figure 2.9, for example, if you tried to sort or filter on the `AuthorsString`, you would get the `could not be translated` exception.

2.6 Building complex queries

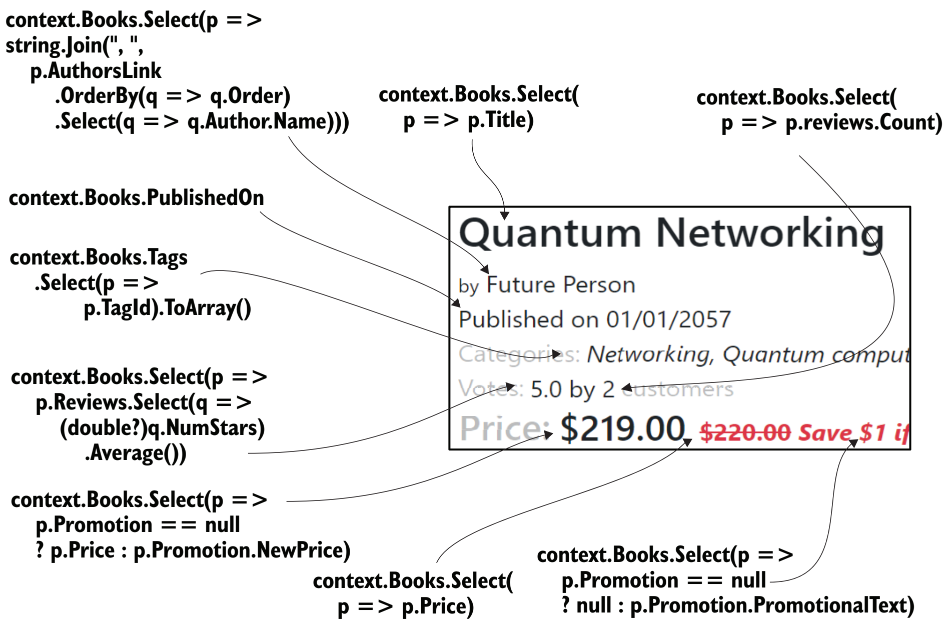
Having covered the basics of querying the database, let's look at examples that are more common in real applications. You're going to build a query to list all the books in the Book App, with a range of features including sorting, filtering, and paging.

You could build the book display by using eager loading. First, you'd load all the data; then, in the code, you'd combine the authors, calculate the price, calculate the average votes, and so on. The problem with that approach is that (a) you are loading data you don't need and (b) sorting and filtering have to be done in software. For this chapter's Book App, which has approximately 50 books, you could eager-load *all* the books and relationships into memory and then sort or filter them in software, but that approach wouldn't work for Amazon!

The better solution is to calculate the values inside SQL Server so that sorting and filtering can be done before the data is returned to the application. In the rest of this chapter, you'll use a select-loading approach that combines the select, sort, filter, and paging parts into one big query. You start in this section with the select part. Before I show you the select query that loads the book data, however, let's go back to the book list display of *Quantum Networking* from the beginning of this chapter. This time, figure 2.10 shows each individual LINQ query needed to get each piece of data.

This figure is complicated because the queries needed to get all the data are complicated. With this diagram in mind, let's look at how to build the book select query. You start with the class you're going to put the data in. This type of class, which exists only to bring together the exact data you want, is referred to in various ways. In ASP.NET, it is referred to as a ViewModel, but that term also has other connotations and uses; therefore, I refer to this type of class as a *Data Transfer Object* (*DTO*). Listing 2.11 shows you the DTO class `BookListDto`.

DEFINITION There are lots of definitions of a Data Transfer Object (DTO), but the one that fits my use of DTOs is "object that is used to encapsulate data, and send it from one subsystem of an application to another" (Stack Overflow, https://stackoverflow.com/a/1058186/1434764).



```
context.Books.Select(p =>
string.Join(", ",
   p.AuthorsLink
      .OrderBy(q => q.Order)
      .Select(q => q.Author.Name)))
```

```
context.Books.Select(
   p => p.Title)
```

```
context.Books.Select(
   p => p.reviews.Count)
```

```
context.Books.PublishedOn
```

```
context.Books.Tags
   .Select(p =>
         p.TagId).ToArray()
```

```
context.Books.Select(p =>
   p.Reviews.Select(q =>
      (double?)q.NumStars)
         .Average())
```

```
context.Books.Select(p =>
   p.Promotion == null
   ? p.Price : p.Promotion.NewPrice)
```

```
context.Books.Select(
   p => p.Price)
```

```
context.Books.Select(p =>
   p.Promotion == null
   ? null : p.Promotion.PromotionalText)
```

Quantum Networking

by Future Person

Published on 01/01/2057

Categories: *Networking, Quantum comput*

Votes: 5.0 by 2 customers

Price: $219.00 ~~$220.00~~ *Save $1 if*

Figure 2.10 Each individual query needed to build the book list display, with each part of the query that's used to provide the value needed for that part of the book display. Some queries are easy, such as getting the title of the book, but others aren't so obvious, such as working out the average votes from the reviews.

Listing 2.11 The DTO `BookListDto`

```
public class BookListDto
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public DateTime PublishedOn { get; set; }
    public decimal Price { get; set; }
    public decimal
        ActualPrice { get; set; }
    public string
        PromotionPromotionalText { get; set; }
    public string AuthorsOrdered { get; set; }

    public int ReviewsCount { get; set; }
    public double?
        ReviewsAverageVotes { get; set; }
    public string[] TagStrings { get; set; }
}
```

You need the primary key if the customer clicks the entry to buy the book.

Although the publication date isn't shown, you'll want to sort by it, so you have to include it.

The normal selling price of the book

Selling price—either the normal price or the promotional.NewPrice if present

Promotional text to show whether there's a new price

String to hold the comma-delimited list of authors' names

Number of people who reviewed the book

Average of all the votes, null if no votes

The Tag names (that is the categories) for this book

To work with EF Core's select loading, the class that's going to receive the data must have a default constructor (which you can create without providing any properties to the constructor), the class must not be static, and the properties must have public setters.

Next, you'll build a select query that fills in every property in `BookListDto`. Because you want to use this query with other query parts, such as sort, filter, and paging, you'll use the `IQueryable<T>` type to create a method called `MapBookToDto` that takes in `IQueryable<Book>` and returns `IQueryable<BookListDto>`. The following listing shows this method. As you can see, the LINQ `Select` pulls together all the individual queries you saw in figure 2.10.

Listing 2.12 The `Select` query to fill `BookListDto`

```
public static IQueryable<BookListDto>
    MapBookToDto(this IQueryable<Book> books)
{
    return books.Select(book => new BookListDto
    {
        BookId = book.BookId,
        Title = book.Title,
        Price = book.Price,
        PublishedOn = book.PublishedOn,
        ActualPrice = book.Promotion == null
            ? book.Price
            : book.Promotion.NewPrice,
        PromotionPromotionalText =
            book.Promotion == null
                ? null
                : book.Promotion.PromotionalText,
        AuthorsOrdered = string.Join(", ",
            book.AuthorsLink
                .OrderBy(ba => ba.Order)
                .Select(ba => ba.Author.Name)),
        ReviewsCount = book.Reviews.Count,
        ReviewsAverageVotes =
            book.Reviews.Select(review =>
                (double?)review.NumStars).Average(),
        TagStrings = book.Tags
            .Select(x => x.TagId).ToArray(),
    });
}
```

Takes in IQueryable<Book> and returns IQueryable<BookListDto>

Simple copies of existing columns in the Books table

Calculates the selling price, which is the normal price, or the promotion price if that relationship exists

PromotionalText depends on whether a PriceOffer exists for this book

Obtains an array of authors' names, in the right order. You're using client vs. server evaluation because you want the author names combined into one string.

You need to calculate the number of reviews.

To get EF Core to turn the LINQ average into the SQL AVG command, you need to cast the NumStars to (double?).

Array of Tag names (categories) for this book

NOTE The individual parts of the `Select` query in listing 2.12 are the repetitive code I mention in my lightbulb moment in chapter 1. Chapter 6 introduces mappers that automate much of this coding, but in part 1 of this book, I list all the code in full so that you see the whole picture. Be assured that there's a way to automate the select-loading approach of querying that will improve your productivity.

The `MapBookToDto` method uses the Query Object pattern; the method takes in `IQueryable<T>` and outputs `IQueryable<T>`, which allows you to encapsulate a query, or part of a query, in a method. That way, the query is isolated in one place, which makes it easier to find, debug, and performance-tune. You'll use the Query Object pattern for the sort, filter, and paging parts of the query too.

> **NOTE** Query Objects are useful for building queries such as the book list in this example, but alternative approaches exist, such as the Repository pattern.

The `MapBookToDto` method is also what .NET calls an *extension method*. Extension methods allow you to chain Query Objects together. You'll see this chaining used in section 2.9, when you combine each part of the book list query to create the final, composite query.

> **NOTE** A method can become an extension method if (a) it's declared in a static class, (b) the method is static, and (c) the first parameter has the keyword `this` in front of it.

Query Objects take in a `IQueryable<T1>` input and return `IQueryable<T2>`, so you're adding LINQ commands to the original `IQueryable<T1>` input. You can add another Query Object to the end, or if you want to execute the query, add an execute command (see figure 2.8) such as `ToList` to execute the query. You'll see this approach in action in section 2.9, when you combine the Book's Select, Sort, Filter, and Paging Query Objects, which EF Core turns into a fairly efficient database query. In chapter 15, you'll work through a series of performance tunes to make the book list query even faster.

> **NOTE** You can see the results of this query by cloning the code from the Git repo and then running the Book App web application locally. A Logs menu feature will show you the SQL used to load the book list with the specific sorting, filtering, and paging setting you've selected.

2.7 Introducing the architecture of the Book App

I've waited until this point to talk about the design of the Book App, because it should make more sense now that you've created the `BookListDto` class. At this stage, you have the entity classes (`Book`, `Author`, and so on) that map to the database via EF Core. You also have a `BookListDto` class, which holds the data in the form that the presentation side needs—in this case, an ASP.NET Core web server.

In a simple example application, you might put the entity classes in one folder, the DTOs in another, and so on. But even in a small application, such as the Book App, this practice can be confusing, because the approach you use with the database is different from the approach you use when displaying data to the customer. The Separation of Concerns (SoC) principle (see http://mng.bz/7Vom) says that your software should be broken down into separate parts. The book display database query, for example, shouldn't contain the code that creates the HTML to show to the books to the user.

You could split the parts of the Book App in numerous ways, but we'll use a common design called *layered architecture*. This approach works well for small to medium-size .NET applications. Figure 2.11 shows the architecture of the Book App for this chapter.

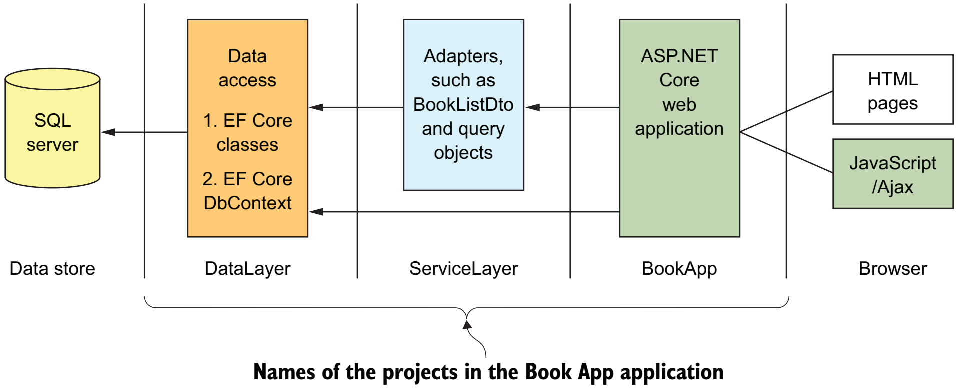


Figure 2.11 The layered architectural approach for the Book App. Placing the parts of the code in discrete projects separates what the code in each project does. The DataLayer, for example, has to worry only about the database and doesn't need to know how the data is going to be used; this is the SoC principle in action. The arrows always point to the left because the lower (left) projects can't access the higher (right) projects.

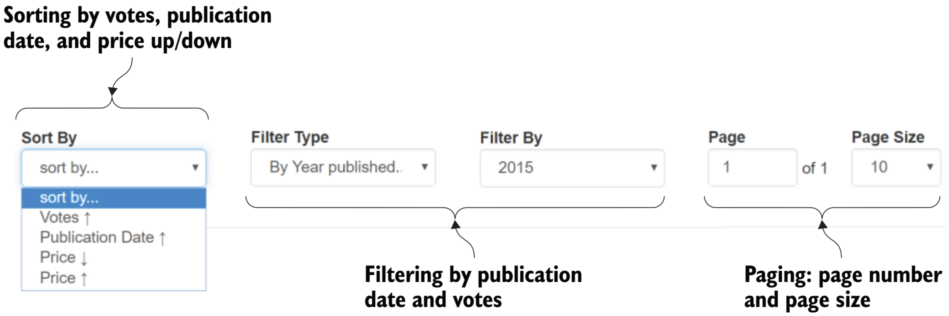
The three large rectangles are .NET projects, with their names at the bottom of the figure. The classes and code of these three projects are split in the following way:

- *DataLayer*—This layer's focus is the database access. The entity classes and the application's DbContext are in this project. This layer doesn't know anything about the layers above it.
- *ServiceLayer*—This layer acts as an adapter between the DataLayer and the ASP.NET Core web application by using DTOs, Query Objects, and various classes to run the commands. The idea is that the frontend ASP.NET Core layer has so much to do that the ServiceLayer hands its premade data for display.
- *BookApp*—The focus of this layer, called the *presentation layer*, is on presenting data in a way that's convenient and applicable to the user. The presentation layer should focus only on the interaction with the user, which is why we move as much as possible of the database and data adapting out of the presentation layer. In the Book App, you'll use an ASP.NET Core web application serving mainly HTML pages, with a small amount of JavaScript running in the browser.

Using a layered architecture makes the Book App a little more complex to understand, but it's one way to build real applications. Using layers also enables you to know more easily what each bit of the code is supposed to be doing in the associated Git repo, because the code isn't all tangled up.

2.8 Adding sorting, filtering, and paging

With the project structure out of the way, you can push on more quickly and build the remaining Query Objects to create the final book list display. I'll start by showing you a screenshot (figure 2.12) of the Book App's sort, filter, and page controls to give you an idea of what you're implementing.



Filter Type: By Year published..

Filter By: 2015

Page: 1 of 1

Page Size: 10

Sorting by votes, publication date, and price up/down

Filtering by publication date and votes

Paging: page number and page size

Figure 2.12 The three commands—sorting, filtering, and paging—as shown on the Book App's home page. You can see this page in action if you run the Book App in the accompanying Git repo.

2.8.1 Sorting books by price, publication date, and customer ratings

Sorting in LINQ is done by the methods `OrderBy` and `OrderByDescending`. You create a Query Object called `OrderBooksBy` as an extension method, as shown in the next listing. You'll see that in addition to the `IQueryable<BookListDto>` parameter, this method takes in an `enum` parameter that defines the type of sort the user wants.

Listing 2.13 The `OrderBooksBy` Query Object method

```
public static IQueryable<BookListDto> OrderBooksBy
    (this IQueryable<BookListDto> books,
     OrderByOptions orderByOptions)
{
    switch (orderByOptions)
    {
        case OrderByOptions.SimpleOrder:
            return books.OrderByDescending(
                x => x.BookId);
        case OrderByOptions.ByVotes:
            return books.OrderByDescending(x =>
                x.ReviewsAverageVotes);
        case OrderByOptions.ByPublicationDate:
            return books.OrderByDescending(
                x => x.PublishedOn);
        case OrderByOptions.ByPriceLowestFirst:
            return books.OrderBy(x => x.ActualPrice);
        case OrderByOptions.ByPriceHighestFirst:
            return books.OrderByDescending(
                x => x.ActualPrice);
```

Because of paging, you always need to sort. You default-sort on the primary key, which is fast.

Orders the book by votes. Books without any votes (null return) go at the bottom.

Orders by publication date, with the latest books at the top

Orders by actual price, which takes into account any promotional price—both lowest first and highest first

```
        default:
            throw new ArgumentOutOfRangeException(
                nameof(orderByOptions), orderByOptions, null);
    }
}
```

Calling the `OrderBooksBy` method returns the original query with the appropriate LINQ sort command added to the end. You pass this query on to the next Query Object, or if you've finished, you call a command to execute the code, such as `ToList`.

> **NOTE** Even if the user doesn't select a sort, you'll still sort (see the `SimpleOrder` switch statement) because you'll be using paging, providing only a page at a time rather than all the data, and SQL requires the data to be sorted to handle paging. The most efficient sort is on the primary key, so you sort on that key.

2.8.2 Filtering books by publication year, categories, and customer ratings

The filtering created for the Book App is a bit more complex than the sorting covered in section 2.8.1 because you get the customer to first select the type of filter they want and then select the actual filter value. The filter value for `Votes` is easy: it's a set of fixed values (4 or higher, 3 or higher, and so on), and the categories are the `Tag's` `TagId`. But to filter by Date, you need to find the dates of the publications to put in the drop-down list.

It's instructive to look at the code for working out the years that have books, because that code is a nice example of combining LINQ commands to create the final drop-down list. Here's a snippet of code taken from the `GetFilterDropDownValues` method.

Listing 2.14 The code to produce a list of the years when books are published

```
var result = _db.Books
    .Where(x => x.PublishedOn <= DateTime.UtcNow.Date)
    .Select(x => x.PublishedOn.Year)
    .Distinct()
    .OrderByDescending(x => x.PublishedOn)
    .Select(x => new DropdownTuple
    {
        Value = x.ToString(),
        Text = x.ToString()
    }).ToList();
var comingSoon = _db.Books.
    Any(x => x.PublishedOn > DateTime.Today);
if (comingSoon)
    result.Insert(0, new DropdownTuple
    {
        Value = BookListDtoFilter.AllBooksNotPublishedString,
```

Loads books while filtering out the future books; then selects the years when the books were published

The Distinct method returns a list of each year a book was published.

Orders the years, with newest year at the top

I finally use two client/server evaluations to turn the values into strings.

Returns true if a book in the list is not yet published

Adds a "coming soon" filter for all the future books

```
        Text = BookListDtoFilter.AllBooksNotPublishedString
    });

return result;
```

The result of this code is a list of `Value`/`Text` pairs holding each year that books are published, plus a Coming Soon section for books yet to be published. This data is turned into an HTML drop-down list by ASP.NET Core and sent to the browser.

The following listing shows the filter Query Object called `FilterBooksBy`, which takes as an input the `Value` part of the drop-down list created in listing 2.14, plus whatever type of filtering the customer has asked for.

Listing 2.15 The `FilterBooksBy` Query Object method

```
public static IQueryable<BookListDto> FilterBooksBy(
    this IQueryable<BookListDto> books,
    BooksFilterBy filterBy, string filterValue)
{
    if (string.IsNullOrEmpty(filterValue))
        return books;

    switch (filterBy)
    {
        case BooksFilterBy.NoFilter:
            return books;
        case BooksFilterBy.ByVotes:
            var filterVote = int.Parse(filterValue);
            return books.Where(x =>
                x.ReviewsAverageVotes > filterVote);
        case BooksFilterBy.ByTags:
            return books.Where(x => x.TagStrings
                .Any(y => y == filterValue));
        case BooksFilterBy.ByPublicationYear:
            if (filterValue == AllBooksNotPublishedString)
                return books.Where(
                    x => x.PublishedOn > DateTime.UtcNow);

            var filterYear = int.Parse(filterValue);
            return books.Where(
                x => x.PublishedOn.Year == filterYear
                 && x.PublishedOn <= DateTime.UtcNow);
        default:
            throw new ArgumentOutOfRangeException
                (nameof(filterBy), filterBy, null);
    }
}
```

The method is given both the type of filter and the user-selected filter value.

If the filter value isn't set, returns IQueryable with no change

For no filter selected, returns IQueryable with no change

The filter by votes returns only books with an average vote above the filterVote value. If there are no reviews for a book, the ReviewsAverageVotes property will be null, and the test always returns false.

Selects any books with a Tag category that matches the filterValue

If Coming Soon was picked, returns only books not yet published

If we have a specific year, we filter on that. Note that we also remove future books (in case the user chose this year's date).

2.8.3 *Other filtering options: Searching text for a specific string*

We could've created loads of other types of filters/searches of books, and searching by title is an obvious one. But you want to make sure that the LINQ commands you use to search a string are executed in the database, because they'll perform much better than loading all the data and filtering in software. EF Core converts the following C# code in a LINQ query to a database command: `==`, `Equal`, `StartsWith`, `EndsWith`, `Contains`, and `IndexOf`. Table 2.1 shows some of these commands in action.

Table 2.1 Example .NET string commands in an SQL Server database

String command — **Example (finds a title with the string `"The Cat sat on the mat."`)**

`StartsWith`

```
var books = context.Books
   .Where(p => p.Title.StartsWith("The"))
   .ToList();
```

`EndsWith`

```
var books = context.Books
   .Where(p => p.Title.EndsWith("MAT."))
   .ToList();
```

`Contains`

```
var books = context.Books
   .Where(p => p.Title.Contains("cat"))
```

The other important thing to know is that the case sensitivity of a string search executed by SQL commands depends on the type of database, and in some databases, the rule is called *collation*. A default SQL Server database default collation uses case-insensitive searches, so searching for `Cat` would find `cat` and `Cat`. Many SQL databases are case-insensitive by default, but Sqlite has a mix of case-sensitive/case-insensitive (see unit test `Ch02_StringSearch` class in the repo for more details), and Cosmos DB is by default case-sensitive.

EF Core 5 provides various ways to set the collation in a database. Typically, you configure the collation for the database or a specific column (covered in section 7.7), but you can also define the collation in a query by using the `EF.Functions.Collate` method. The following code snippet sets an SQL Server collation, which means that this query will compare the string using the `Latin1_General_CS_AS` (case-sensitive) collation for this query:

```
context.Books.Where( x =>
      EF.Functions.Collate(x.Title, "Latin1_General_CS_AS")
          == “HELP” //This does not match “help”
```

> **NOTE** Defining what is uppercase and what is lowercase over many languages with many scripts is a complex issue! Fortunately, relational databases have been performing this task for many years, and SQL Server has more than 200 collations.

Another string command is the SQL command `LIKE`, which you can access through the `EF.Function.Like` method. This command provides a simple pattern-matching approach using _ (underscore) to match any letter and % to match zero-to-many characters. The following code snippet would match `"The Cat sat on the mat."` and `"The` `dog sat on the step."` but not `"The rabbit sat on the hutch."` because `rabbit` isn't three letters long:

```
var books = context.Books
    .Where(p => EF.Functions.Like(p.Title, "The ___ sat on the %."))
    .ToList();
```

OTHER QUERY OPTIONS: COMPLEX QUERIES (GROUPBY, SUM, MAX, AND SO ON)

This chapter has covered a wide range of query commands, but EF Core can translate many more commands to most databases. Section 6.1.8 covers the commands that need a bit more explanation or special coding.

2.8.4 Paging the books in the list

If you've used Google search, you've used paging. Google presents the first dozen or so results, and you can *page* through the rest. Our Book App uses paging, which is simple to implement by using the LINQ commands' `Skip` and `Take` methods.

Although the other Query Objects were tied to the `BookListDto` class because the LINQ paging commands are so simple, you can create a generic paging Query Object that will work with any `IQueryable<T>` query. This Query Object is shown in the following listing. The object does rely on getting a page number in the right range, but another part of the application has to do that anyway to show the correct paging information onscreen.

Listing 2.16 A generic `Page` Query Object method

```
public static IQueryable<T> Page<T>(
    this IQueryable<T> query,
    int pageNumZeroStart, int pageSize)
{
    if (pageSize == 0)
        throw new ArgumentOutOfRangeException
            (nameof(pageSize), "pageSize cannot be zero.");

    if (pageNumZeroStart != 0)
        query = query
            .Skip(pageNumZeroStart * pageSize);

    return query.Take(pageSize);
}
```

Skips the correct number of pages

Takes the number for this page size

As I said earlier, paging works only if the data is ordered. Otherwise, SQL Server will throw an exception because relational databases don't guarantee the order in which data is handed back; there's no default row order in a relational database.

2.9 Putting it all together: Combining Query Objects

We've covered each Query Object you need to build a book list for the Book App. Now it's time to see how to combine these Query Objects to create a composite query to work with the website. The benefit of building a complex query in separate parts is that this approach makes writing and testing the overall query simpler, because you can test each part on its own.

Listing 2.17 shows a class called `ListBooksService`, which has one method, `SortFilterPage`, which uses all the Query Objects (select, sort, filter, and page) to build the composite query. It also needs the application's DbContext to access the `Books` property, which you provide via the constructor.

> **TIP** Listing 2.17 highlights in bold the `AsNoTracking` method. This method stops EF Core from taking a tracking snapshot (see figure 1.6) on read-only queries, which makes the query slightly quicker. You should use the `AsNoTracking` method in any read-only queries (queries in which you read the data, but don't ever update it). In this case, we are not loading any entity classes, so it's redundant, but I put it there to remind us that the query is read-only.

Listing 2.17 The `ListBookService` class providing a sorted, filtered, and paged list

```
public class ListBooksService
{
    private readonly EfCoreContext _context;

    public ListBooksService(EfCoreContext context)
    {
        _context = context;
    }

    public IQueryable<BookListDto> SortFilterPage
        (SortFilterPageOptions options)
    {
        var booksQuery = _context.Books
            .AsNoTracking()
            .MapBookToDto()
            .OrderBooksBy(options.OrderByOptions)
            .FilterBooksBy(options.FilterBy,
                           options.FilterValue);

        options.SetupRestOfDto(booksQuery);

        return booksQuery.Page(options.PageNum-1,
                               options.PageSize);
    }
}
```

Starts by selecting the Books property in the Application's DbContext

Because this query is read-only, you add .AsNoTracking.

Uses the Select Query Object, which picks out/calculates the data it needs

Adds the commands to order the data by using the given options

Adds the commands to filter the data

This stage sets up the number of pages and makes sure that PageNum is in the right range.

Applies the paging commands

As you can see, the four Query Objects—select, sort, filter, and page—are added in turn (called *chaining*) to form the final composite query. Note that the `options.SetupRestOfDto(booksQuery)` code before the Page Query Object sorts out things such as how many pages there are, ensures that the `PageNum` is in the right range, and performs a few other housekeeping items. Chapter 5 shows how the `ListBooksService` is called in our ASP.NET Core web application.

Summary

- To access a database in any way via EF Core, you need to define an application DbContext.
- An EF Core query consists of three parts: the application's DbContext property, a series of LINQ/EF Core commands, and a command to execute the query.
- Using EF Core, you can model three primary database relationships: one-to-one, one-to-many, and many-to-many. Other relationships are covered in chapter 8.
- The classes that EF Core maps to the database are referred to as *entity classes*. I use this term to highlight the fact that the class I'm referring to is mapped by EF Core to the database.
- If you load an entity class, it won't load any of its relationships by default. Querying the `Book` entity class, for example, won't load its relationship properties (`Reviews`, `AuthorsLink`, and `Promotion`); it leaves them as `null`.
- You can load related data that's attached to an entity class in four ways: eager loading, explicit loading, select loading, and lazy loading.
- EF Core's client vs. server evaluation feature allows the last stage of a query to contain commands, such as `string.Join`, that can't be converted to SQL commands.
- I use the term *Query Object* to refer to an encapsulated query or a section of a query. These Query Objects are often built as .NET extension methods, which means that they can easily be chained together, similar to the way LINQ is written.
- Selecting, sorting, filtering, and paging are common query uses that can be encapsulated in a Query Object.
- If you write your LINQ queries carefully, you can move the aggregate calculations, such as `Count`, `Sum`, and `Average`, into the relational database, improving performance.

For readers who are familiar with EF6.x:

- Many of the concepts in this chapter are the same as in EF6.x. In some cases (such as eager loading), the EF Core commands and/or configuration have changed slightly, but often for the better.

3

Changing the database content

This chapter covers

- Creating a new row in a database table
- Updating existing rows in a database table for two types of applications
- Updating entities with one-to-one, one-to-many, and many-to-many relationships
- Deleting single entities, and entities with relationships, from a database

Chapter 2 covered querying a database. This chapter moves on to changing the content of a database. Changing data has three distinct parts—creating new rows in a database table, updating existing rows in a database table, and deleting rows in a database table—and I cover them in that order. *Create*, *update*, and *delete*, along with *read* (which is *query* in EF Core terms) are database terms for what's happening, and the foursome is often shortened to *CRUD*.

You'll use the same database as in chapter 2, which has the `Book`, `PriceOffer`, `Review`, `BookAuthor`, and `Author` entity classes. These classes provide a good selection of property types and relationships that you can use to learn the various issues and approaches to changing data in a database via EF Core.

3.1 Introducing EF Core's entity State

Before I start describing the methods to add, update, or delete entities, I want to introduce you to EF Core's entity property, called `State`. This property provides another look under the hood at the way EF Core does things, which helps you understand what's going on when you add, update, or delete entities.

Any entity class instance has a `State`, which can be accessed via the following EF Core command: `context.Entry(someEntityInstance).State`. The `State` tells EF Core what to do with this instance when `SaveChanges` is called. Here's a list of the possible states and what happens if `SaveChanges` is called:

- `Added`—The entity needs to be created in the database. `SaveChanges` inserts it.
- `Unchanged`—The entity exists in the database and hasn't been modified on the client. `SaveChanges` ignores it.
- `Modified`—The entity exists in the database and has been modified on the client. `SaveChanges` updates it.
- `Deleted`—The entity exists in the database but should be deleted. `SaveChanges` deletes it.
- `Detached`—The entity you provided isn't tracked. `SaveChanges` doesn't see it.

Normally, you don't look at or alter the `State` directly. You use the various commands listed in this chapter to add, update, or delete entities. These commands make sure the `State` is set in a *tracked entity* (see definition below). When `SaveChanges` is called, it looks at all the tracked entities and their `State` to decide what type of database changes it needs to apply to the database. I refer to the entity's `State` in the rest of the chapter to show you how EF Core decides what type of change to apply to the database.

DEFINITION *Tracked entities* are entity instances that have been read in from the database by using a query that didn't include the `AsNoTracking` method. Alternatively, after an entity instance has been used as a parameter to EF Core methods (such as `Add`, `Update`, or `Delete`), it becomes tracked.

3.2 Creating new rows in a table

Creating new data in a database is about adding (via the `SQL` command `INSERT` in a relational database) a new row to a table. If you want to add a new author to our Book App, for example, that addition would be referred to as a create operation on the database.

In EF Core terms, creating new data in a database is the simplest of the update operations because EF Core can take a set of linked entity classes, save them to the database, and sort out the foreign keys needed to link things. In this section, you'll start with a simple example and then build up to more complex creates.

3.2.1 Creating a single entity on its own

Let’s start with an entity class that has no navigational properties—that is, relationships to other tables in your database. This example is rare but shows the two steps in a create operation:

1. Add the entity to the application’s DbContext.
2. Call the application’s DbContext’s `SaveChanges` method.

This listing creates an `ExampleEntity` entity class and adds a new row to the table that the entity is mapped to—in this case, the ExampleEntities table.

Listing 3.1 An example of creating a single entity

```
var itemToAdd = new ExampleEntity
{
    MyMessage = "Hello World"
};
context.Add(itemToAdd);
context.SaveChanges();
```

Uses the Add method to add SingleEntity to the application’s DbContext. The DbContext determines the table to add it to, based on its parameter type.

Calls the SaveChanges method from the application’s DbContext to update the database

Because you add the entity instance `itemToAdd` that wasn’t originally tracked, EF Core starts to track it and sets its `State` to `Added`. After `SaveChanges` is called, EF Core finds a tracked entity of type `ExampleEntity` with a `State` of `Added`, so it’s added as a new row in the database table associated with the `ExampleEntity` class.

> **EF6** In EF6.x, you’d need to add the `itemToAdd` to a `DbSet<ExampleEntity>` property in the application’s DbContext—for example, `context.ExampleEntities.Add(itemToAdd)`. That approach is still valid, but EF Core has introduced the shorthand shown in listing 3.1, which applies to the `Add`, `Remove`, `Update`, and `Attach` methods. (See chapter 11 for more on the last two commands.) EF Core works out which entity you're altering by looking at the type of the instance you provide.

EF Core creates the SQL command to update an SQL Server–based database.

Listing 3.2 SQL commands created to insert a new row into the `SingleEntities` table

```
SET NOCOUNT ON;
INSERT INTO ExampleEntities]
   ([MyMessage]) VALUES (@p0);

SELECT [ExampleEntityId]
FROM [ExampleEntities]
WHERE @@ROWCOUNT = 1 AND
     [ExampleEntityId] = scope_identity();
```

Inserts (creates) a new row into the ExampleEntities table

Reads back the primary key in the newly created row

The second SQL command produced by EF Core reads back the primary key of the row that was created by the database server. This command ensures that the original `ExampleEntity` instance is updated with the primary key so that the in-memory version of the entity is the same as the version in the database. Reading back the primary key is important, as you might update the entity later, and the update will need the primary key.

> **EF6** In EF6.x, when you call `SaveChanges`, EF6.x by default validates the data by using the standard .NET validation approach; it looks for data validation attributes and, if they are present, runs `IValidatableObject.Validate` on entity classes. EF Core doesn't include this feature because a lot of validation is done in the frontend, but it's not hard to add a validation feature if you need it. Chapter 4 shows you how.

3.2.2 Creating a book with a review

Next, you'll look at a create that includes relationships—in this case, adding a new book with a review. Although the setup of the entity classes is a bit more complex, the process has the same steps as our earlier, nonrelational create:

- It adds the entity class(es) in some way to EF Core's tracked entities with the `State` of `Add`.
- It calls `SaveChanges`, which looks at the `State` of all the tracked entities and runs the SQL `INSERT` command for all entities with the `State` set to `Added`.

This example uses the Book App's database with its Books and Review tables. Figure 3.1 shows a partial database diagram of these tables.

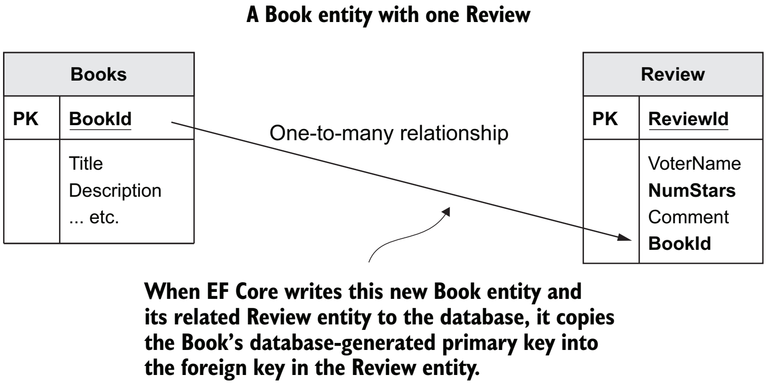


Figure 3.1 The Books and Review tables. The Review row has a foreign key that EF Core fills with the primary key value from the new Books row that's created.

In the next listing, you create a new `Book` entity and fill the `Reviews` collection property with a single `Review` entity. Then you call the `context.Add` method, followed by the `SaveChanges` method, which writes both entities to the database.

Listing 3.3 Adding a `Book` entity class also adds any linked entity classes

```
var book = new Book
{
    Title = "Test Book",
    PublishedOn = DateTime.Today,
    Reviews = new List<Review>()
    {
        new Review
        {
            NumStars = 5,
            Comment = "Great test book!",
            VoterName = "Mr U Test"
        }
    }
};

context.Add(book);
context.SaveChanges();
```

Creates the book with the title "Test Book"

Creates a new collection of reviews

Adds one review with its content

Uses the Add method to add the book to the application's DbContext property, Books

Calls the SaveChanges method from the application's DbContext to update the database. It finds a new Book, which has a collection containing one new Review, and then adds both to the database.

The thing to note from this listing is that you add only the `Book` entity class, but the related `Review` entity class is also written to the database. This happens because EF Core follows all the relational links and finds the new `Review` instance, and because that `Review` isn't tracked, EF Core knows that the `Review` needs to be added to the database.

As you saw in the simple example in listing 3.1, EF Core works out what to do with the linked entity classes by accessing their EF Core `State` values. If the linked instances are new (not already known to EF Core), EF Core will start tracking them and set their `State` to `Added`. In all other cases, EF Core will obey the `State` linked to the entity instance. In listing 3.3, the `Review` entity instance isn't already known to EF Core, which means that its `State` is `Detached`, but when the `Add` call is made, its `State` is set to `Added`. That instance will be `INSERT`ed into the database as a new row.

WHAT HAPPENS AFTER THE SAVECHANGES RETURNS SUCCESSFULLY?

When the `Add` and `SaveChanges` have finished successfully, a few things happen: the entity instances that have been inserted into the database are now tracked by EF Core, and their `State` is set to `Unchanged`. Because we are using a relational database, and because the two entity classes, `Book` and `Review`, have primary keys that are of type `int`, EF Core by default will expect the database to create the primary keys by using the SQL `IDENTITY` keyword. Therefore, the SQL commands created by EF Core read back the primary keys into the appropriate primary keys in the entity class instances to make sure that the entity classes match the database.

NOTE The Cosmos DB database doesn't have an equivalent to SQL's `IDENTITY`, so you need to provide a unique key, such as GUIDs (globally unique identifiers).

> Unique GUIDs are generated by what EF Core calls a `ValueGenerator` (see chapter 10). GUIDs are also useful for primary keys in relational databases when you need a unique key that won't change when you copy/duplicate the data to another database.

Also, EF Core knows about the relationships by the navigational properties in the entity classes. In listing 3.3, the `Book` entity's `Reviews` collection property has a new `Review` entity instance in it. As part of the `SaveChanges` process, any foreign key will be set by copying the primary keys into the foreign keys in each of the new relationships. Then the entity instance matches the database. That's useful in case you want to read the primary or foreign keys, and EF Core can detect any subsequent changes you make to the primary or foreign keys if you call `SaveChanges` again.

Why you should call `SaveChanges` only once at the end of your changes

In listing 3.3. you see that the `SaveChanges` method is called at the end of create, and you see the same pattern—the `SaveChanges` method is called at the end—in the update and delete examples too. In fact, even for complex database change containing a mixture of creates, updates, and deletes, you should still call the `SaveChanges` method only once at the end. You do that because EF Core will save all your changes (creates, updates and deletes) and apply them to the database together, and if the database rejects any of your changes, all your changes are rejected (by means of a database feature called a *transaction*; see section 4.7.2).

This pattern is called a *Unit Of Work* and means that your database changes can't be half-applied to the database. If you created a new `Book` with a `BookAuthor` reference to an `Author` that wasn't in the database, for example, you wouldn't want the `Book` instance to be saved. Saving it might break the book display, which expects every `Book` to have at least one `Author`.

Sometimes, you may think that you need to call `SaveChanges` twice—say, when you need the primary key of a new entity class to fill in the foreign key of entity class, but there is always a way around that situation with EF Core. In fact, listing 3.3 gets around it by creating a new `Book` and a new `Review` at the same time. Have a read through section 6.2.1 and 6.2.2 to get an "under the hood" look at how EF Core achieves this task.

EXAMPLE THAT HAS ONE INSTANCE ALREADY IN THE DATABASE

The other situation you may need to deal with is creating a new entity containing a navigational property that uses another entity already in the database. If you want to create a new `Book` entity that has an `Author` that already exists in the database, you need to obtain a tracked instance of the `Author` entity that you want to add to your new `Book` entity. The following listing gives you one example. Note that the database already contains an author called "Mr. A."

Listing 3.4 Adding a Book with an existing Author

```
var foundAuthor = context.Authors
    .SingleOrDefault(author => author.Name == "Mr. A");
if (foundAuthor == null)
    throw new Exception("Author not found");

var book = new Book
{
    Title = "Test Book",
    PublishedOn = DateTime.Today
};
book.AuthorsLink = new List<BookAuthor>
{
    new BookAuthor
    {
        Book = book,
        Author = foundAuthor
    }
};

context.Add(book);
context.SaveChanges();
```

Reads in the Author with a check that the Author was found

Creates a Book in the same way as the previous example

Adds an AuthorBook linking entry, but uses the Author that is already in the database

Adds the new Book to the DbContext Books property and calls SaveChanges

The first four lines load an `Author` entity with some checks to make sure that it was found; this `Author` class instance is tracked, so EF Core knows that it is already in the database. You create a new `Book` entity and add a new `BookAuthor` linking entity, but instead of creating a new `Author` entity instance, you use the `Author` entity that you read in from the database. Because EF Core is tracking the `Author` instance and knows that it's in the database, EF Core won't try to add it again to the database when `SaveChanges` is called at the end of listing 3.4.

3.3 *Updating database rows*

Updating a database row is achieved in three stages:

1. Read the data (database row), possibly with some relationships.
2. Change one or more properties (database columns).
3. Write the changes back to the database (update the row).

In this section, you'll ignore any relationships and focus on the three stages. In the next section, you'll learn how to update relationships by adding more commands to each stage.

Listing 3.5 changes the publication date of an existing book. Through this code, you can see the standard flow of an update:

1. You load the entity class(es) you want to change as a tracked entity.
2. You change the property/properties in your entity class(es).
3. You call `SaveChanges` to update the database.

Listing 3.5 Updating *Quantum Networking*'s publication date

```
var book = context.Books
    .SingleOrDefault(p =>
         p.Title == "Quantum Networking");
if (book == null)
    throw new Exception("Book not found");

book.PublishedOn = new DateTime(2058, 1, 1);
context.SaveChanges();
```

Finds the specific book you want to update—in this case, our special book Quantum Networking

Throws an exception if the book isn't found

Changes the expected publication date to year 2058 (it was 2057)

Calls SaveChanges, which includes running a method called DetectChanges. This method spots that the PublishedOn property has been changed.

When the `SaveChanges` method is called, it runs a method called `DetectChanges`, which compares the tracking snapshot against the entity class instance that it handed to the application when the query was originally executed. From this example, EF Core decides that only the `PublishedOn` property has been changed, and EF Core builds the SQL to update that property.

> **NOTE** Using the tracking snapshot is the normal way that `DetectChanges` finds the changed properties. But chapter 11 describes an alternative to the tracking snapshot, such as `INotifyPropertyChanging`. This topic is advanced, so I use the tracked-entities approach throughout part 1 of this book.

The following listing shows the two SQL commands that EF Core produces for the code in listing 3.5. One SQL command finds and loads the `Book` entity class, and a second command updates the PublishedOn column.

Listing 3.6 SQL generated by EF Core for the query and update in listing 3.5

```
SELECT TOP(2)
   [p].[BookId],
   [p].[Description],
   [p].[ImageUrl],
   [p].[Price],
   [p].[PublishedOn],
   [p].[Publisher],
   [p].[Title]
FROM [Books] AS [p]
WHERE [p].[Title] = N'Quantum Networking'

SET NOCOUNT ON;
UPDATE [Books]
   SET [PublishedOn] = @p0

WHERE [BookId] = @p1;
SELECT @@ROWCOUNT;
```

Reads up to two rows from the Books table. You asked for a single item, but this code makes sure that it fails if more than one row fits.

The read loads all the columns in the table.

Your LINQ Where method, which picks out the correct row by its title

SQL UPDATE command—in this case, on the Books table

Because EF Core's DetectChanges method finds that only the PublishedOn property has changed, it can target that column in the table.

Sends back the number of rows that were inserted into this transaction. SaveChanges returns this integer, but normally, you can ignore it.

EF Core uses the primary key from the original book to uniquely select the row it wants to update.

3.3.1 Handling disconnected updates in a web application

As you learned in section 3.3, an update is a three-stage process, needing a read, an update, and a `SaveChanges` call to be executed by the same instance of the application's DbContext. The problem is that for certain applications, such as websites and RESTful APIs, using the same instance of the application's DbContext isn't possible because in web applications, each HTTP request typically is a new request, with no data held over from the last HTTP request. In these types of applications, an update consists of two stages:

- The first stage is an initial read, done in one instance of the application's DbContext.
- The second stage applies the update by using a new instance of the application's DbContext.

In EF Core, this type of update is called a *disconnected* update because the first stage and the second stage use two different instances of the application's DbContext (see the preceding list). You can handle a disconnected update in several ways. The method you should use depends a lot on your application. Here are the two main ways of handling disconnected updates:

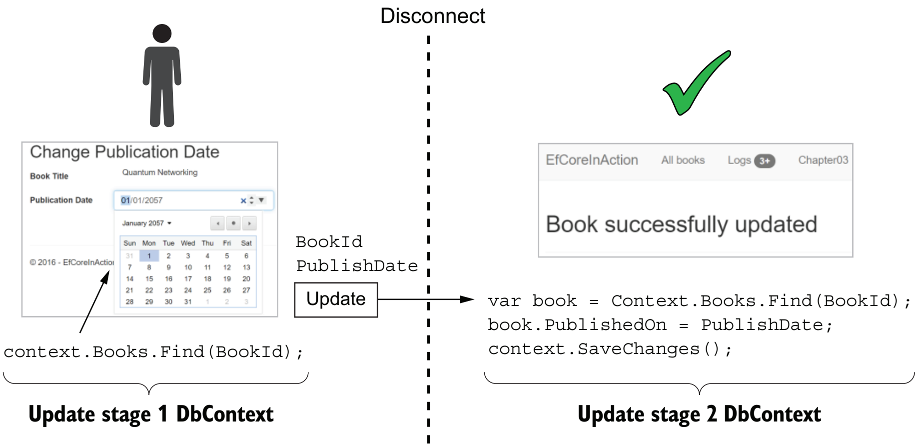
- *You send only the data you need to update back from the first stage.* If you were updating the published date for a book, you would send back only the `BookId` and the `PublishedOn` properties. In the second stage, you use the primary key to reload the original entity with tracking and update the specific properties you want to change. In this example, the primary key is the `BookId`, and the property to update is the `PublishedOn` property of the `Book` entity (see figure 3.2). When you call `SaveChanges`, EF Core can work out which properties you've changed and update only those columns in the database.
- *You send all the data needed to re-create the entity class back from the first stage.* In the second stage, you rebuild the entity class, and maybe relationships, by using the data from the first stage and tell EF Core to update the whole entity (see figure 3.3). When you call `SaveChanges`, EF Core will know, because you told it, that it must update all the columns in the table row(s) affected with the substitute data that the first stage provided.

NOTE Another way of handling the partial update of an entity described in option 1 is to create a new entity instance and manipulate the `State` of each property. Chapter 11 covers this option, when we look at how to alter the entity's `State` in more detail.

That's a lot of words! Now I'll give you an example of each approach for handling disconnected updates.

DISCONNECTED UPDATE, WITH RELOAD

Figure 3.2 shows an example of a disconnected update in a web application. In this case, you're providing a feature to allow an admin user to update the publication date of a book. The figure shows that you send only the `BookId` and the `PublicationDate` data back from the first stage.



```
context.Books.Find(BookId);
```

```
var book = Context.Books.Find(BookId);
book.PublishedOn = PublishDate;
context.SaveChanges();
```

Figure 3.2 The two stages in a disconnected update on a website using EF Core. The thick, dashed line in the middle represents the point where the data held in the application in the first stage is lost, and the second stage starts with no knowledge of what stage 1 did. Only the `BookId` and `PublishDate` information is returned when the user clicks the Update button that bridges the gap.

For web applications, the approach of returning only a limited amount of data to the web server is a common way of handling EF Core updates. This approach makes the request faster, but a big reason for it is security. You wouldn't want the `Price` of a `Book` to be returned, for example, as that information would allow hackers to alter the price of the book they want to buy.

There are several ways of controlling what data is returned/accepted by the web server. In ASP.NET Core, for example, you have the attribute `BindNever`, which allows you to define named properties that won't be returned to the second stage. But a more general approach, and one I prefer, is to use a special class that contains only properties that should be sent/received. This class is referred to as a DTO or ViewModel. It's similar in nature to the DTO used in the select-loading query in chapter 2, but in this case it's used not only in the query, but also to receive the specific data you need back from the user, via a browser. For our example that updates the publication date, you need three parts. The first part, a DTO to send/receive the data to/from the user, is shown here.

Listing 3.7 `ChangePubDateDto` sends data to and receives it from the user

```
public class ChangePubDateDto
{
    public int BookId { get; set; }

    public string Title { get; set; }

    [DataType(DataType.Date)]
    public DateTime PublishedOn { get; set; }
}
```

Holds the primary key of the row you want to update, which makes finding the right row quick and accurate

You send over the title to show the user so that they can be sure they are altering the right book.

The property you want to alter. You send out the current publication date and get back the changed publication date.

The quickest way to read an entity class using its primary key(s)

When you want to update a specific entity and need to read it in using its primary key, you have a few options. I used to use the `Find` command, but after some digging, I now recommend `SingleOrDefault` because it's quicker than the `Find` command. But I should point out two useful things about the `Find` method:

- The `Find` method checks the current application's DbContext to see whether the required entity instance has already been loaded, which can save an access to the database. But if the entity isn't in the application's DbContext, the load will be slower because of this extra check.
- The `Find` method is simpler and quicker to type because it's shorter than the `SingleOrDefault` version, such as `context.Find<Book>(key)` versus `context.SingleOrDefault(p => p.Bookid == key)`.

The upside of using the `SingleOrDefault` method is that you can add it to the end of a query with methods such as `Include`, which you can't do with `Find`.

Second, you need a method to get the initial data for stage 1. Third, you need a method to receive the data back from the browser and then reload/update the book. This listing shows the `ChangePubDateService` class that contains two methods to handle these two stages.

Listing 3.8 The `ChangePubDateService` class to handle the disconnected update

This interface is needed when registering this class in DI. You use DI in chapter 5 when building the ASP.NET Core BookApp.

```
public class ChangePubDateService : IChangePubDateService
{
    private readonly EfCoreContext _context;

    public ChangePubDateService(EfCoreContext context)
    {
        _context = context;
    }
```

The application's DbContext is provided via a class constructor—the normal way of building classes that you will use as a service in ASP.NET Core.

```
    public ChangePubDateDto GetOriginal(int id)
    {
        return _context.Books
            .Select(p => new ChangePubDateDto
            {
                BookId = p.BookId,
                Title = p.Title,
                PublishedOn = p.PublishedOn
            })
            .Single(k => k.BookId == id);
    }

    public Book UpdateBook(ChangePubDateDto dto)
    {
        var book = _context.Books.SingleOrDefault(
            x => x.BookId == dto.BookId);
        if (book == null)
            throw new ArgumentException(
                "Book not found");
        book.PublishedOn = dto.PublishedOn;
        _context.SaveChanges();
        return book;
    }
}
```

This method handles the first part of the update, such as by getting the data from the chosen book to show to the user.

A select load query that returns only three properties

Uses the primary key to select the exact row we want to update

This method handles the second part of the update, such as performing a selective update of the chosen book.

Loads the book. I use SingleOrDefault because it's slightly quicker than the Find method.

I catch the case where a book wasn't found and throw an exception.

Selective update of the PublishedOn property of the loaded book

SaveChanges uses its DetectChanges method to find out what has changed and then updates the database.

Returns the updated book

The advantages of this reload-then-update approach is that it's more secure (in our example, sending/returning the price of the book over HTTP would allow someone to alter it) and faster because of less data. The downside is that you have to write code to copy over the specific properties you want to update. Chapter 6 covers a few tricks to automate this process.

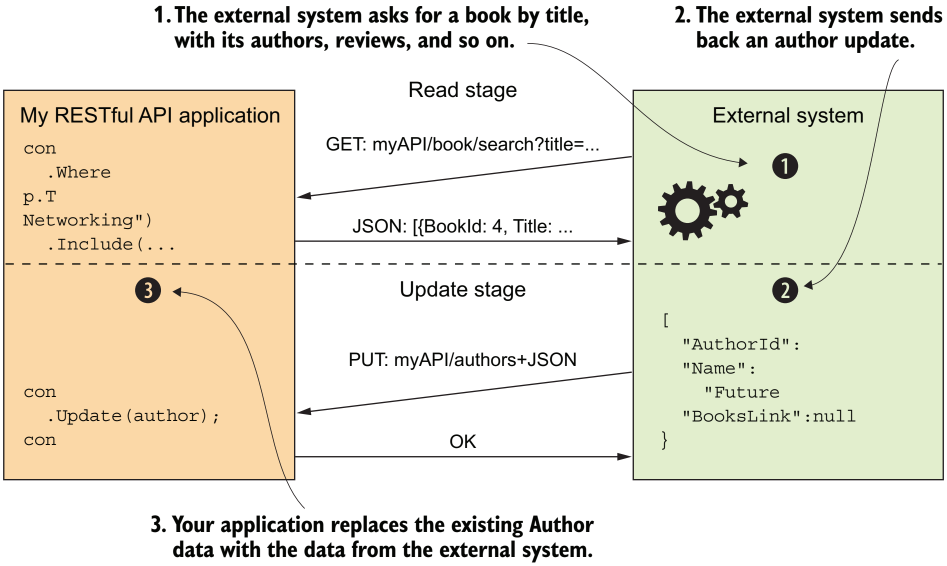
> **NOTE** You can see this code and try updating the publication date on the example Book App. If you download the code from the Git repo and run it locally, you'll see an Admin button for each book. This button contains a link called Change Pub Date, which will step you through this process. You can also see the SQL commands that EF Core uses to carry out this update via the Logs menu item.

DISCONNECTED UPDATE, SENDING ALL THE DATA

In some cases, all the data may be sent back, so there's no reason to reload the original data. This can happen for simple entity classes, in some RESTful APIs, or process-to-process communication. A lot depends on how closely the given API format matches the database format and how much you trust the other system.

Figure 3.3 shows an example of a RESTful API in which an external system first queries the system for books with a given title. In the update stage, the external system sends back an update on the author of the book it received.

Listing 3.9 simulates the RESTful API by having a first stage that reads in the `Author` entity class you want to update and then serializes it into a JSON string.



1. The external system asks for a book by title, with its authors, reviews, and so on.

2. The external system sends back an author update.

3. Your application replaces the existing Author data with the data from the external system.

Figure 3.3 An example of a disconnected update, in which you replace all the database information with the new data. Unlike the one in the previous example, this process doesn't need to reload the original data before performing the update.

(Figure 3.3, step 2 shows what that JSON looks like.) Then you decode that JSON and use the EF Core `Update` command, which replaces all the information in the row defined by the primary key—in this case, the `AuthorId`.

Listing 3.9 Simulating an update/replace request from an external system

```
string json;
using (var context = new EfCoreContext(options))
{
    var author = context.Books
        .Where(p => p.Title == "Quantum Networking")
        .Select(p => p.AuthorsLink.First().Author)
        .Single();
    author.Name = "Future Person 2";
    json = JsonConvert.SerializeObject(author);
}
using (var context = new EfCoreContext(options))
{
    var author = JsonConvert
        .DeserializeObject<Author>(json);


    context.Authors.Update(author);
    context.SaveChanges();
}
```

Simulates an external system returning a modified Author entity class as a JSON string

Provides a link to the many-to-many linking table that links to the authors of this book

Simulates receiving a JSON string from an external system and decoding it into an Author class

Update command, which replaces all the row data for the given primary key—in this case, AuthorId

You call the EF Core `Update` command with the `Author` entity instance as a parameter, which marks as modified all the properties of the `Author` entity. When the `SaveChanges` command is called, it'll update all the columns in the row that have the same primary key as the entity class.

> **EF6** The `Update` command is new in EF Core. In EF6.x, you need to manipulate the entity object state directly, such as by using the command `DbContext.Entry(object).State = EntityState.Modified`. Subtle changes in the way that EF Core sets the entity state are covered in chapter 11.

The plus side of this approach is that the database update is quicker, because you don't have the extra read of the original data. You also don't have to write code to copy over the specific properties you want to update, which you did need to do in the previous approach.

The downsides are that more data can be transferred and that unless the API is carefully designed, it can be difficult to reconcile the data you receive with the data already in the database. Also, you're trusting the external system to remember all the data correctly, especially the primary keys of your system.

> **NOTE** Listing 3.9 covers only a single class with no relationship, but in many RESTful APIs and process-to-process communications, a lot of linked data might be sent over. In the example, the API might expect the whole book with all its relationships to be sent back only for an update of the author's name. This process gets complicated, so I cover it in chapter 11, which shows how to manage the state of each property and introduces EF Core's `TrackGraph` method, which helps handle partial updates of classes with relationships.

3.4 Handling relationships in updates

Now that we've established the three basic steps for updating the database, it's time to look at updating relationships between entity classes—adding a new review to a book, for example. Updating relationships adds another level of complexity to the code, especially in the disconnected state, which is why I put this content in a separate section.

This section covers updates for the three types of relational linking that EF Core uses and gives examples of both connected and disconnected updates. In all cases, you'll use the `Book` entity class, which has three relationship links. The following listing shows the `Book` entity class, but with the focus on the relationships at the end. (I've removed some nonrelational properties to keep the focus on the relationships.)

Listing 3.10 The `Book` entity class, showing the relationships to update

```
public class Book
{
    public int BookId { get; set; }
    //… other nonrelational properties removed for clarity
```

Book class contains the main book information.

```
    //-----------------------------------------------
    //relationships

    public PriceOffer Promotion { get; set; }
    public ICollection<Review> Reviews { get; set; }
    public ICollection<Tag> Tags { get; set; }
    public ICollection<BookAuthor>
        AuthorsLink { get; set; }
}
```

Links to the optional PriceOffer

Can be zero to many reviews of the book

EF Core 5's automatic many-to-many relationship to the Tag entity class

Provides a link to the many-to-many linking table that links to the authors of this book

3.4.1 Principal and dependent relationships

The terms *principal* and *dependent* are used in EF to define parts of a relationship:

- *Principal entity*—Contains a primary key that the dependent relationship refer to via a foreign key
- *Dependent entity*—Contains the foreign key that refers to the principal entity's primary key

In the Book App example, the `Book` entity class is the principal entity. The `PriceOffer`, `Review`, and `BookAuthor` entity classes are the dependent entities. I find the terms *principal* and *dependent* to be helpful, because they define what's in charge: the principal entity. I use these terms throughout this book where applicable.

> **NOTE** An entity class can be both a principal and a dependent entity at the same time. In a hierarchical relationship of, say, libraries with books that have reviews, the book would be a dependent relationship of the library entity class.

CAN THE DEPENDENT PART OF A RELATIONSHIP EXIST WITHOUT THE PRINCIPAL?

The other aspect of a dependent relationship is whether it can exist on its own. If the principal relationship is deleted, is there a business case for the dependent relationship to still exist? In many cases, the dependent part of a relationship doesn't make sense without the principal relationship. A book review has no meaning if the book it links to is deleted from the database, for example.

In a few cases, a dependent relationship should exist even if the principal part is deleted. Suppose that you want to have a log of all the changes that happen to a book in its lifetime. If you delete a book, you wouldn't want that set of logs to be deleted too.

This task is handled in databases by handling the nullability of the foreign key. If the foreign key in the dependent relationship is non-nullable, the dependent relationship can't exist without the principal. In the example Book App database, the `PriceOffer`, `Review`, and `BookAuthor` entities are all dependent on the principal, `Book` entity, so their foreign keys are of type `int`. If the book is deleted or the link to the book is removed, the dependent entities will be deleted.

But if you define a class for logging—let's call it `BookLog`—you want this class to exist even if the book is deleted. To make this happen, you'd make its `BookId` foreign key of type `Nullable<int>`. Then, if you delete the book that the `BookLog` entity is linked to, you could configure that the `BookLog`'s `BookId` foreign key would be set to `null`.

> **NOTE** In the preceding `BookLog` example, if you delete a `Book` entity that a `BookLog` is linked to, the default action is to set the `BookLog's` foreign key to `null` because EF Core defaults to a `ClientSetNull` setting for the `OnDelete` property of optional relationships. Section 8.8.1 covers this topic in more detail.

I mention this situation now because as we go through updating the relationships, in some cases, a dependent relationship is removed from its principal. I'll give an example of replacing all the dependent relationships with new ones. What happens to the old relationships we remove depends on the nullability of the foreign key: if the foreign key is non-nullable, the dependent relationships are deleted, and if the foreign key is nullable, it's set to `null`. I talk more about this topic and how EF Core handles deletion in section 3.5.

3.4.2 Updating one-to-one relationships: Adding a PriceOffer to a book

In our example Book App database, we have an optional, dependent relationship property called `Promotion` from the `Book` entity class to the `PriceOffer` entity class. This subsection covers how to add a `PriceOffer` class to an existing book. This listing shows you the content of the `PriceOffer` entity class, which links to the Books table via the foreign key called `BookId`.

Listing 3.11 `PriceOffer` entity class, showing the foreign key back to the `Book` entity

```
public class PriceOffer
{
    public int PriceOfferId { get; set; }
    public decimal NewPrice { get; set; }
    public string PromotionalText { get; set; }

    //----------------------------------------------
    //Relationships

    public int BookId { get; set; }
}
```

PriceOffer, if present, is designed to override the normal price.

Foreign key back to the book it should be applied to

Connected state update

The connected state update assumes that you're using the same context for both the read and the update. Listing 3.12 shows an example of the code, which has three stages:

1. Load the `Book` entity with any existing `PriceOffer` relationship.
2. Set the relationship to the new `PriceOffer` entity you want to apply to this book.
3. Call `SaveChanges` to update the database.

Listing 3.12 Adding a new promotional price to an existing book that doesn't have one

Finds a book. In this example, the book doesn't have an existing promotion, but it would also work if there were an existing promotion.

Although the include isn't needed because you're loading something without a Promotion, using the include is good practice, as you should load any relationships if you're going to change a relationship.

```
var book = context.Books
    .Include(p => p.Promotion)
    .First(p => p.Promotion == null);

book.Promotion = new PriceOffer
{
    NewPrice = book.Price / 2,
    PromotionalText = "Half price today!"
};
context.SaveChanges();
```

Adds a new PriceOffer to this book

The SaveChanges method calls DetectChanges, which finds that the Promotion property has changed, so it adds that entity to the PriceOffers table.

As you can see, the update of the relationship is like the basic update you made to change the book's published date. In this case, EF Core has to do extra work because of the relationship. EF Core creates a new row in the PriceOffers table, which you can see in the SQL snippet that EF Core produces for the code in listing 3.12:

```
INSERT INTO [PriceOffers]
    ([BookId], [NewPrice], [PromotionalText])
     VALUES (@p0, @p1, @p2);
```

Now, what happens if there's an existing promotion on the book (that is, the `Promotion` property in the `Book` entity class isn't `null`)? That case is why the `Include(p => p.Promotion)` command in the query that loaded the `Book` entity class is so important. Because of that `Include` method, EF Core will know that an existing `PriceOffer` is assigned to this book and will delete it before adding the new version.

To be clear, in this case you must use some form of loading of the relationship—*eager*, *explicit*, *select*, or *lazy* loading of the relationship—so that EF Core knows about it before the update. If you don't, and if there's an existing relationship, EF Core will throw an exception on a duplicate foreign key `BookId`, which EF Core has placed a unique index on, and another row in the `PriceOffers` table will have the same value.

DISCONNECTED STATE UPDATE

In the disconnected state, the information to define which book to update and what to put in the `PriceOffer` entity class would be passed back from stage 1 to stage 2. That situation happened in the update of the book's publication date (figure 3.2), where the `BookId` and the `PublishedOn` values were fed back.

In the case of adding a promotion to a book, you need to pass in the `BookId`, which uniquely defines the book you want, plus the `NewPrice` and the `PromotionalText` values that make up the `PriceOffer` entity class. The next listing shows you the `ChangePriceOfferService` class, which contains the two methods to show the data to the user and update the promotion on the `Book` entity class when the user submits a request.

Listing 3.13 `ChangePriceOfferService` class with a method to handle each stage

```
public class ChangePriceOfferService : IChangePriceOfferService
{
    private readonly EfCoreContext _context;

    public Book OrgBook { get; private set; }

    public ChangePriceOfferService(EfCoreContext context)
    {
        _context = context;
    }

    public PriceOffer GetOriginal(int id)
    {
        OrgBook = _context.Books
            .Include(r => r.Promotion)
            .Single(k => k.BookId == id);

        return OrgBook?.Promotion
            ?? new PriceOffer
               {
                   BookId = id,
                   NewPrice = OrgBook.Price
               };
    }

    public Book AddUpdatePriceOffer(PriceOffer promotion)
    {
        var book = _context.Books
            .Include(r => r.Promotion)
            .Single(k => k.BookId
                  == promotion.BookId);
        if (book.Promotion == null)
        {
            book.Promotion = promotion;
        }
        else
        {
            book.Promotion.NewPrice
                = promotion.NewPrice;
            book.Promotion.PromotionalText
                = promotion.PromotionalText;
        }
        _context.SaveChanges();
        return book;
    }
}
```

Gets a PriceOffer class to send to the user to update

Loads the book with any existing Promotion

You return either the existing Promotion for editing or create a new one. The important point is to set the BookId, as you need to pass it through to the second stage.

Handles the second part of the update, performing a selective add/update of the Promotion property of the selected book

Loads the book with any existing promotion, which is important because otherwise, your new PriceOffer will clash and throw an error

Checks whether the code should create a new PriceOffer or update the existing PriceOffer

You need to add a new PriceOffer, so you assign the promotion to the relational link. EF Core will see it and add a new row in the PriceOffer table.

You need to do an update, so you copy over only the parts that you want to change. EF Core will see this update and produce code to update only these two columns.

SaveChanges uses its DetectChanges method, which sees what changes—either adding a new PriceOffer or updating an existing one.

Returns the updated book

This code either updates an existing `PriceOffer` or adds a new `PriceOffer` if none exists. When `SaveChanges` is called, it can work out, via EF Core's `DetectChanges` method, what type of update is needed and create the correct SQL to update the database. This is different from the connected version shown in listing 3.12, where you replaced any `PriceOffer` with a new version. Both versions work, but if you are logging who last created/updated an entity (see section 11.4.3), updating an existing entity gives you a bit more information about what changed.

Alternative way of updating the relationship: Creating a new row directly

We've approached this update as changing a relationship in the `Book` entity class, but you can also approach it as creating/deleting a row in the PriceOffers table. This listing finds the first `Book` in the database that doesn't have a `Promotion` linked to it and then adds a new `PriceOffer` entity to that book.

Listing 3.14 Creating a `PriceOffer` row to go with an existing book

```
var book = context.Books
    .First(p => p.Promotion == null);

context.Add( new PriceOffer
{
    BookId = book.BookId,
    NewPrice = book.Price / 2,
    PromotionalText = "Half price today!"
});
context.SaveChanges();
```

You find the book that you want to add the new PriceOffer to, which must not be an existing PriceOffer.

Adds the new PriceOffer to the PriceOffers table

Defines the PriceOffer. You must include the BookId (which EF Core filled in previously).

SaveChanges adds the PriceOffer to the PriceOffers table.

You should note that previously, you didn't have to set the `BookId` property in the `PriceOffer` entity class, because EF Core did that for you. But when you're creating a relationship this way, you do need to set the foreign key. Having done so, if you load the `Book` entity class with its `Promotion` relationship after the previous create code, you'll find that the `Book` has gained a `Promotion` relationship.

> **NOTE** The `PriceOffer` entity class doesn't have a relational property link back to the `Book` class (`public Book BookLink {get; set;}`). If it did, you could set the `BookLink` to the `Book` entity class instead of setting the foreign key. Either setting the foreign key(s) or setting a relational link back to the principal entity will tell EF Core to set up the relationship.

The advantage of creating the dependent entity class is that it saves you from needing to reload the principal entity class (in this case, `Book`) in a disconnected state. The downside is that EF Core doesn't help you with the relationships. In this case, if there were an existing `PriceOffer` on the book and you added another, `SaveChanges` would fail because you'd have two `PriceOffer` rows with the same foreign key.

When EF Core can't help you with the relationships, you need to use the create/delete approach with care. Sometimes, this approach can make handling a complex relationship easier, so it's worth keeping in mind, but I prefer updating the principal entity class's relationship in most one-to-one cases.

NOTE Later, in section 3.4.5, you'll learn another way of updating relationships by changing foreign keys.

3.4.3 Updating one-to-many relationships: Adding a review to a book

You've learned the basic steps in updating a relationship by looking at a one-to-one relationship. I'll move a bit quicker with the remaining relationships, as you've seen the basic pattern. But I'll also point out some differences in the *many* side of a relationship.

The one-to-many relationship in the Book App database is represented by `Book`'s `Reviews`; a user of the site can add a review to a book. There can be any number of reviews, from none to a lot. This listing shows the `Review`-dependent entity class, which links to the Books table via the foreign key called `BookId`.

Listing 3.15 The `Review` class, showing the foreign key back to the `Book` entity class

```
public class Review
{
    public int ReviewId { get; set; }
    public string VoterName { get; set; }
    public int NumStars { get; set; }
    public string Comment { get; set; }

    //----------------------------------------------
    //Relationships

    public int BookId { get; set; }
}
```

Holds customer reviews with their ratings

Foreign key holds the key of the book this review belongs to.

CONNECTED STATE UPDATE

Listing 3.16 adds a new `Review` to a `Book`. This code follows the same pattern as the one-to-one connected update: load the `Book` entity class and the `Reviews` relationship via the `Include` method. But in this case, you add the `Review` entity to the `Book's Reviews` collection. Because you used the `Include` method, the `Reviews` property will be an empty collection if there are no reviews or a collection of the reviews linked to this book. In this example, the database already contains some `Book` entities, and I take the first.

Listing 3.16 Adding a review to a book in the connected state

```
var book = context.Books
    .Include(p => p.Reviews)
    .First();

book.Reviews.Add(new Review
{
    VoterName = "Unit Test",
    NumStars = 5,
    Comment = "Great book!"
});
context.SaveChanges();
```

Finds the first book and loads it with any reviews it might have

Adds a new review to this book

SaveChanges calls DetectChanges, which finds that the Reviews property has changed, and from there finds the new Review, which it adds to the Review table.

As with the `PriceOffer` example, you don't fill in the foreign key (the `BookId` property) in the `Review`, because EF Core knows that the `Review` is being added to a `Book` entity class and sets up the foreign key to the right value.

ALTERING/REPLACING ALL THE ONE-TO-MANY RELATIONSHIPS

Before moving on to the disconnected state update, I want to consider the case in which you want to alter or replace the whole collection, rather than add to the collection, as you did with the review.

If the books had categories (say, Software Design, Software Languages, and so on), you might allow an admin user to change the categories. One way to implement this change would be to show the current categories in a multiselect list, allow the admin user to change them, and then replace *all* the categories on the book with the new selection.

EF Core makes replacing the whole collection easy. If you assign a new collection to a one-to-many relationship that has been loaded with tracking (such as by using the `Include` method), EF Core will replace the existing collection with the new collection. If the items in the collection can be linked to only the principal class (the dependent class has a non-nullable foreign key), by default, EF Core will delete the items that were in the collection that have been removed.

Next is an example of replacing the whole collection of existing book reviews with a new collection. The effect is to remove the original reviews and replace them with the one new review.

Listing 3.17 Replacing a whole collection of reviews with another collection

```
var book = context.Books
    .Include(p => p.Reviews)
    .Single(p => p.BookId == twoReviewBookId);

book.Reviews = new List<Review>
{
    new Review
    {
        VoterName = "Unit Test",
        NumStars = 5,
    }
};
context.SaveChanges();
```

This include is important; it creates a collection with any existing reviews in it or an empty collection if there are no existing reviews.

This book you're loading has two reviews.

You replace the whole collection.

SaveChanges, via DetectChanges, knows that the old collection should be deleted and that the new collection should be written to the database.

Because you're using test data in the example, you know that the book with the primary key `twoReviewBookId` has two reviews and that the book is the only one with reviews; hence, there are only two reviews in the whole database. After the `SaveChanges` method is called, the book has only one review, and the two old reviews have been deleted, so now the database has only one review in it.

Removing a single row is as simple as removing the entity from the list. EF Core will see the change and delete the row that's linked to that entity. Similarly, if you add a new `Review` to the `Book`'s `Reviews` collection property, EF Core will see that change to that collection and add the new `Review` to the database.

The loading of the existing collection is important for these changes: if you don't load them, EF Core can't remove, update, or replace them. The old versions will still be in the database after the update because EF Core didn't know about them at the time of the update. You haven't replaced the existing two `Reviews` with your single `Review`. In fact, you now have three `Reviews`—the two that were originally in the database and your new one—which is not what you intended to do.

DISCONNECTED-STATE UPDATE

In the disconnected state, you create an empty `Review` entity class but fill in its foreign key, `BookId`, with the book the user wants to provide a review for. Then the user votes on the book, and you add that review to the book that they referred to. The following listing shows the `AddReviewService` class, which has methods for the setup and update of the book, to add a new review from a user.

Listing 3.18 Adding a new review to a book in the example Book App

```
public class AddReviewService
{
    private readonly EfCoreContext _context;

    public string BookTitle { get; private set; }

    public AddReviewService(EfCoreContext context)
    {
        _context = context;
    }

    public Review GetBlankReview(int id)
    {
        BookTitle = _context.Books
            .Where(p => p.BookId == id)
            .Select(p => p.Title)
            .Single();
        return new Review
        {
            BookId = id
        };
    }

    public Book AddReviewToBook(Review review)
    {
        var book = _context.Books
            .Include(r => r.Reviews)
            .Single(k => k.BookId
                  == review.BookId);
        book.Reviews.Add(review);
        _context.SaveChanges();
        return book;
    }
}
```

Forms a review to be filled in by the user

You read the book title to show to the user when they're filling in their review.

Creates a review with the BookId foreign key filled in

Updates the book with the new review

Loads the correct book by using the value in the review's foreign key, and includes any existing reviews (or an empty collection if there are no reviews yet)

Adds the new review to the Reviews collection

SaveChanges uses its DetectChanges method, which sees that the Book Review property has changed, and creates a new row in the Review table.

Returns the updated book

This code has a simpler first part than the previous disconnected-state examples because you're adding a new review, so you don't have to load the existing data for the user. But overall, the code takes the same approach that the `ChangePriceOfferService` class used.

ALTERNATIVE WAY OF UPDATING THE RELATIONSHIP: CREATING A NEW ROW DIRECTLY

As with the `PriceOffer`, you can add a one-to-many relationship directly to the database. But again, you take on the role of managing the relationship. If you want to replace the entire reviews collection, for example, you'd have to delete all the rows that the reviews linked to the book in question before adding your new collection.

Adding a row directly to the database has some advantages, because loading all the one-to-many relationships might turn out to be a lot of data if you have lots of items and/or they're big. Therefore, keep this approach in mind if you have performance issues.

> **NOTE** My experiments show that not loading the relationship and then assigning a new collection to a one-to-many relationship is equivalent to creating a new row directly. But I don't recommend doing this because it's not the normal update pattern; someone else (or even you) might come back later and misread your intentions.

3.4.4 Updating a many-to-many relationship

In EF Core, we talk about many-to-many relationships, but a relational database doesn't directly implement many-to-many relationships. Instead, we're dealing with two one-to-many relationships, as shown in figure 3.4.

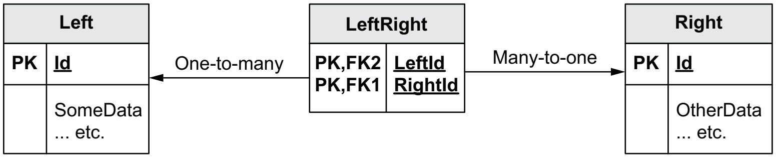


Figure 3.4 A many-to-many relationship in the database is created by a linking table that contains the primary keys of the two tables that need a many-to-many relationship.

In EF Core, you have two ways to create many-to-many relationships between two entity classes:

- You link to a linking table in each entity—that is, you have an `ICollection<LeftRight>` property in your `Left` entity class. You need to create an entity class to act as the linking table (such as `LeftRight` in figure 3.4), but that entity class lets you add extra data in the linking table so that you can sort/filter the many-to-many relationships.

- You link directly between the two entity classes you want to have a many-to-many relationship—that is, you have an `ICollection<Right>` property in your `Left` entity class. This link is much easier to code because EF Core handles the creation of the linking table, but then you can't access the linking table in a normal `Include` method to sort/filter.

NOTE This chapter uses EF Core default settings for a many-to-many relationship. Chapter 8 covers the configuration options for many-to-many relationships.

UPDATING A MANY-TO-MANY RELATIONSHIP VIA A LINKING ENTITY CLASS

In the `Book` entity class, you need a many-to-many link to the `Authors` of the book. But in a book, the order of the authors' names matters. Therefore, you create a linking table with an `Order (byte)` property that allows you to display the `Author's Name` properties in the correct order, which means that you

- Create an entity class called `BookAuthor`, which contains both the primary key of the `Book` entity class (`BookId`) and the primary key of the `Author` entity class (`AuthorId`). You also add an `Order` property, which contains a number setting the order in which the `Authors` should be displayed for this book. The `BookAuthor` linking entity class also contains two one-to-one relationships to the `Author` and the `Book`.
- You add a navigational property called `AuthorsLink` of type `ICollection<BookAuthor>` to your `Book` entity class.
- You also add a navigational property called `BooksLink` of type `ICollection<BookAuthor>` to your `Author` entity class.

These three entity classes are shown in figure 3.5, with only the `Book` to `BookAuthor` and `BookAuthor` to `Author` links shown.

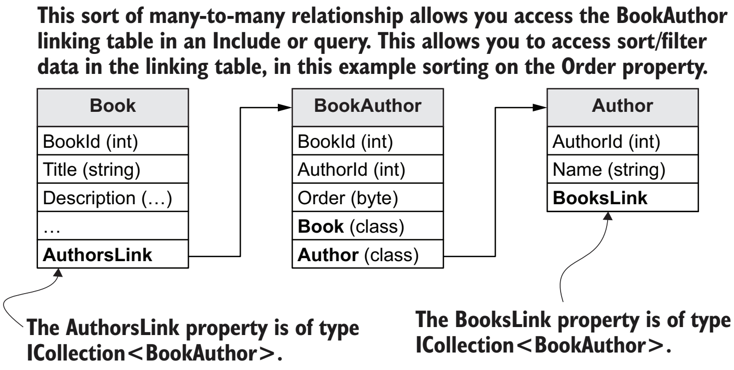


Figure 3.5 The `Book` to its `Authors` many-to-many relationship, which uses a `BookAuthor` linking table. Because you create a one-to-many link to the `BookAuthor` entity class, you can access the `Order` property to sort the order in which the `Author` names should be shown to the customer.

The `BookAuthor` entity class, shown in figure 3.5, has two properties: `BookId` and `AuthorId`. These properties are foreign keys to the Books table and the Authors table, respectively. Together, they also form the primary key (known as a *composite key*, because it has more than one part) for the `BookAuthor` row. The composite key has the effect of ensuring that there's only one link between the `Book` and the `Author`. Chapter 7 covers composite keys in more detail. In addition, the `BookAuthor` entity class also has an `Order` property, which allows you to define the order of the `Author` entity classes so that the `Author's Name` property will be shown in the Book App book list.

As an example, you will add the author Martin Fowler as an extra author to the *Quantum Networking book* via the `BookAuthor` linking entity class. (I'm sure that Martin Fowler would love to collaborate on this book if he's around when quantum networking is perfected.) You set the `Order` property to `1` to make Martin Fowler the second author. (The existing `BookAuthor` entity for the current `Author` has the `Order` property set to `0`.) The next listing shows the resulting code.

Listing 3.19 Adding a new `Author` to the book *Quantum Networking*

```
var book = context.Books
    .Include(p => p.AuthorsLink)
    .Single(p => p.Title == "Quantum Networking");

var existingAuthor = context.Authors
    .Single(p => p.Name == "Martin Fowler");

book.AuthorsLink.Add(new BookAuthor
{
    Book = book,
    Author = existingAuthor,
    Order = (byte) book.AuthorsLink.Count
});
context.SaveChanges();
```

This code finds the book with the title "Quantum Networking," whose current author is "Future Person."

You find an existing author—in this case, "Martin Fowler."

You add a new BookAuthor linking entity to the Book's AuthorsLink collection.

You fill in the two navigational properties that are in the many-to-many relationship.

You set the Order to the old count of AuthorsLink—in this case, 1 (because the first author has a value of 0).

The SaveChanges will create a new row in the BookAuthor table.

The thing to understand is that the `BookAuthor` entity class is the *many* side of the relationship. This listing, which adds another author to one of the books, should look familiar because it's similar to the one-to-many update methods I've already explained.

One thing to note is that when you load the Book's `AuthorsLink`, you don't need to load the corresponding `BooksLink` in the `Author` entity class. The reason is that when you update the `AuthorsLink` collection, EF Core knows that there is a link to the `Book`, and during the update, EF Core will fill in that link automatically. The next time someone loads the `Author` entity class and its `BooksLink` relationship, they'll see a link to the *Quantum Networking* book in that collection. (See section 6.2.2 for a detailed review of what links are filled in when.)

Also be aware that deleting an `AuthorsLink` entry won't delete the `Book` or `Author` entities they link to because that entry is the *one* end of a one-to-many relationship, which isn't dependent on the `Book` or `Author`. In fact, the `Book` and `Author` entity classes are *principal entities*, with the `BookAuthor` classes being dependent on both of the principal entity classes.

UPDATING A MANY-TO-MANY RELATIONSHIP WITH DIRECT ACCESS TO THE OTHER ENTITY

EF Core 5 added the ability to access another entity class directly in a many-to-many relationship. This ability makes it much easier to set up and use the many-to-many relationship, but you won't be able to access the linking table in an `Include` method.

> **EF6** In EF6.x, you can define a many-to-many relationship, and EF6.x will create a hidden linking table for you and handle all the creation/deletion of the rows in that table. EF Core 5 adds that ability, but now you have much better control of configuration of the linking table.

In the Book App, a book can have zero to many categories, such as Linux, Databases, and Microsoft .NET, to help a customer find the right book. These categories are held in a `Tag` entity (the `TagId` holds the category name) with a direct many-to-many relationship to a `Book`. This allows the `Book` to show its categories in the Book App's book list display and also allows the Book App to provide a feature to filter the book list display by a category. Figure 3.6 shows the `Book` and `Tag` entity classes with their properties that link directly to one another.

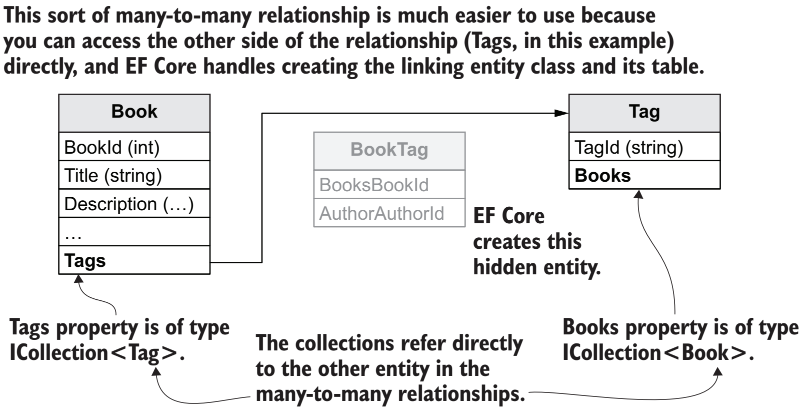


Figure 3.6 A direct many-to-many relationship between the `Book` entity class and the `Tag` entity class. You can access each end of the many-to-many relationship. EF Core builds a hidden entity class when it sees this sort of many-to-many relationship and creates the correct database code to use the associated linking table.

This direct-access many-to-many feature makes adding/deleting links between the `Book` entity and the `Tag` entities simple. The following listing shows how you would add another `Tag` to the *Quantum Networking* `Book`.

Listing 3.20 Adding a `Tag` to a `Book` via a direct many-to-many relationship

```
var book = context.Books
    .Include(p => p.Tags)
    .Single(p => p.Title == "Quantum Networking");

var existingTag = context.Tags
    .Single(p => p.TagId == "Editor's Choice");

book.Tags.Add(existingTag);
context.SaveChanges();
```

Finds the book with the title "Quantum Networking" and loads it with its Tags

You find the Tag called "Editor's Choice" to add this book.

You add the Tag to the Books Tags collection.

When SaveChanges is called, EF Core creates a new row in the hidden BookTags table.

If you compare the previous listing (listing 3.20) with adding another `Author` to a `Book` in listing 3.19, you'll see that it's much easier to add a new entry to a direct many-to-many relationship. EF Core takes on the work of creating the necessary row in the BooksTag table. And if you removed an entry in the `Tags` collection, you would delete the corresponding row in the BooksTag table.

Alternative way of updating the relationship: Creating a new row directly

Having described how to update the two types of many-to-many relationships, now I'll discuss another approach: creating the linking table row directly. The benefit of this approach is better performance when you have lots of entries in the collection.

Rather than having to read in the collection, you can create a new entry in the linking table. You could create a `BookAuthor` entity class and fill in the `Book` and `Author` one-to-one relationships in that class, for example. Then you `Add` that new `BookAuthor` entity instance to the database and call `SaveChanges`. For the `AuthorsLink` collection, which is likely to be small, this technique is most likely not worth the extra effort, but for many-to-many relationships that contain lots of linking entries, it can significantly improve performance.

3.4.5 *Advanced feature: Updating relationships via foreign keys*

Up to this point, I've shown you how to update relationships by using the entity classes themselves. When you added a review to a book, for example, you loaded the `Book` entity with all its `Reviews`. That's fine, but in a disconnected state, you have to load the `Book` and all its `Reviews` from the book's primary key that came back from the browser/RESTful API. In many situations, you can cut out the loading of the entity classes and set the foreign keys instead.

This technique applies to most of the disconnected updates I've shown so far, but let me give you an example of moving a review from one book to another. (I know—this scenario is unlikely in the real world. But it makes for a simple example.) The following listing carries out the update after the user types the request. The code assumes that the `ReviewId` of the `Review` the user wants to change and the new `BookId` that they want to attach the review to are returned in a variable called `dto`.

Listing 3.21 Updating the foreign key to change a relationship

```
var reviewToChange = context
    .Find<Review>(dto.ReviewId);
reviewToChange.BookId = dto.NewBookId;
context.SaveChanges();
```

Finds the review that you want to move by using the primary key returned from the browser

Changes the foreign key in the review to point to the book it should be linked to

Calls SaveChanges, which finds the foreign key in the review changed, so it updates that column in the database

The benefit of this technique is that you don't have to load the `Book` entity class or use an `Include` command to load all the `Reviews` associated with this book. In our example Book App, these entities aren't too big, but in a real application, the principal and dependent entities could be quite large. (Some Amazon products have thousands of reviews, for example.) In disconnected systems, in which we often send only the primary keys over the disconnect, this approach can be useful for cutting down on database accesses and, hence, improving performance.

> **NOTE** When updating relationships via foreign keys, you may need to access entities that don't have a `DbSet<T>` property in the application's DbContext, so how can you read in the data? Listing 3.21 uses the `Find<T>` method, but if you need a more complex query, you can access any entity via the `Set<T>` method, such as `context.Set<Review>().Where(p => p.NumVotes > 5)`.

3.5 Deleting entities

The final way to change the data in the database is to delete a row from a table. Deleting data is easier than making the updates we've already discussed, but it does have a few points to be aware of. Before I describe how to delete entities from the database, I want to introduce an approach called *soft delete*, in which an entity is hidden instead of deleted.

> **NOTE** I have some extra information about using soft delete in section 6.1.7, which covers certain situations in real applications.

3.5.1 Soft-delete approach: Using a global query filter to hide entities

One school of thought says that you shouldn't delete anything from a database but use a status to hide it, known as a soft delete. (See Udi Dahan's post "Don't Delete—Just Don't" at http://mng.bz/6glD.) I think this approach is a sensible one, and EF Core provides a feature called global query filter that allows a soft delete to be implemented simply.

The thinking behind a soft delete is that in real-world applications, data doesn't stop being data; it transforms into another state. In the case of our books example, a book may not still be on sale, but the fact that the book existed isn't in doubt, so why delete it? Instead, you set a flag to say that the entity is to be hidden in all queries and relationship. To see how this process works, you'll add the soft-delete feature to the list of `Book` entities. To do so, you need to do two things:

- *Add a* `boolean` *property called* `SoftDeleted` *to the* `Book` *entity class.* If that property is `true`, the `Book` entity instance is soft-deleted; it shouldn't be found in a normal query.
- *Add a global query filter via EF Core's fluent configuration commands.* The effect is to apply an extra `Where` filter to any access to the `Books` table.

Adding the `SoftDeleted` property to a `Book` entity instance is straightforward. This code snippet shows the `Book` entity class with the `SoftDeleted` property:

```
public class Book
{
    //… other properties left out for clarity
    public bool SoftDeleted { get; set; }
}
```

Adding the global query filter to the `DbSet<Book>Books` property means adding an EF Core configuration command to the application's DbContext. Chapter 7 covers this configuration command, but it's shown in bold in the following listing so that you have an idea of what's going on.

Listing 3.22 Adding a global query filter to the `DbSet<Book>Books` property

```
public class EfCoreContext : DbContext
{
    //… Other parts removed for clarity

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
        //… other configration parts removed for clarity

        modelBuilder.Entity<Book>()
            .HasQueryFilter(p => !p.SoftDeleted);
    }
}
```

Adds a filter to all accesses to the Book entities. You can bypass this filter by using the IgnoreQueryFilters operator.

To soft-delete a `Book` entity, you need to set the `SoftDeleted` property to `true` and call `SaveChanges`. Then any query on the `Book` entities will exclude the `Book` entities that have the `SoftDeleted` property set to `true`.

If you want to access all the entities that have a model-level filter, you add the `IgnoreQueryFilters` method to the query, such as `context.Books.IgnoreQueryFilters()`. This method bypasses any query filter on that entity.

NOTE I have built a library called `EfCore.SoftDeleteServices` that provides a code for both configuring and using this form of soft delete. See http://mng.bz/op7r for more information.

Now that we've covered the soft-delete approach, let's cover the ways to truly delete an entity from the database. We'll start with a straightforward example and work up to deleting an entity that has relationships.

3.5.2 Deleting a dependent-only entity with no relationships

I've chosen the `PriceOffer` entity class to show a basic delete because it's a dependent entity. Therefore, you can delete it without affecting other entities. This listing finds a `PriceOffer` and then deletes it.

Listing 3.23 Removing (deleting) an entity from the database

```
var promotion = context.PriceOffers
    .First();

context.Remove(promotion);
context.SaveChanges();
```

Finds the first PriceOffer

Removes that PriceOffer from the application's DbContext. The DbContext works out what to remove based on its parameter type.

SaveChanges calls DetectChanges, which finds a tracked PriceOffer entity marked as deleted and then deletes it from the database.

Calling the `Remove` method sets the `State` of the entity provided as the parameter to `Deleted`. Then, when you call `SaveChanges`, EF Core finds the entity marked as `Deleted` and creates the correct database commands to delete the appropriate row from the table the entity referred to (in this case, a row in the PriceOffers table). The SQL command that EF Core produces for SQL Server is shown in the following snippet:

```
SET NOCOUNT ON;
DELETE FROM [PriceOffers]
WHERE [PriceOfferId] = @p0;
SELECT @@ROWCOUNT;
```

3.5.3 Deleting a principal entity that has relationships

Section 3.3.1 discussed principal and dependent relationships and the nullability of the foreign key. Relational databases need to keep *referential integrity*, so if you delete a row in a table that other rows are pointing to via a foreign key, something has to happen to stop referential integrity from being lost.

> **DEFINITION** *Referential integrity* is a relational database concept indicating that table relationships must always be consistent. Any foreign-key field must agree with the primary key referenced by the foreign key (see http://mng.bz/XY0M).

Following are three ways that you can set a database to keep referential integrity when you delete a principal entity with dependent entities:

- You can tell the database server to delete the dependent entities that rely on the principal entity, known as *cascade deletes*.

- You can tell the database server to set the foreign keys of the dependent entities to `null`, if the column allows that.
- If neither of those rules is set up, the database server will raise an error if you try to delete a principal entity with dependent entities.

3.5.4 Deleting a book with its dependent relationships

In this section, you're going to delete a `Book` entity, which is a principal entity with three dependent relationships: `Promotion`, `Reviews`, and `AuthorsLink`. These three dependent entities can't exist without the `Book` entity; a non-nullable foreign key links these dependent entities to a specific `Book` row.

By default, EF Core uses cascade deletes for dependent relationships with non-nullable foreign keys. Cascade deletes make deleting principal entities easier from the developer's point of view, because the other two rules need extra code to handle deleting the dependent entities. But in many business applications, this approach may not be appropriate. This chapter uses the cascade delete approach because it's EF Core's default for non-nullable foreign keys.

With that caveat in mind, let's see cascade delete in action by using the default cascade-delete setting to delete a `Book` that has relationships. This listing loads the `Promotion` (`PriceOffer` entity class), `Reviews`, `AuthorsLink`, and `Tags` relationships with the `Book` entity class before deleting that `Book`.

Listing 3.24 Deleting a book that has three dependent entity classes

```
var book = context.Books
    .Include(p => p.Promotion)
    .Include(p => p.Reviews)
    .Include(p => p.AuthorsLink)
    .Include(p => p.Tags)
    .Single(p => p.Title
         == "Quantum Networking");

context.Books.Remove(book);
context.SaveChanges();
```

The four Includes make sure that the four dependent relationships are loaded with the Book.

Finds the Quantum Networking book, which you know has a promotion, two reviews, one BookAuthor link, and one BookTag

Deletes that book

SaveChanges calls DetectChanges, which finds a tracked Book entity marked as deleted, deletes its dependent relationships, and then deletes the book.

My test data contains a book with the title *Quantum Networking*, which has one `PriceOffer`, two `Reviews`, and a `BookAuthor` entity associated with it. The foreign keys of all those dependent entities I mentioned point to the *Quantum Networking* book. After the code in listing 3.24 has run, EF Core deletes the `Book`, the `PriceOffer`, the two `Reviews`, the single `BookAuthor` link, and the single (hidden) `BookTag`.

That last statement, indicating that all are deleted by EF Core, is an important point. Because you put in the four `Includes`, EF Core knew about the dependent entities and performed the delete. If you didn't incorporate the `Includes` in your code, EF Core wouldn't know about the dependent entities and couldn't delete the three dependent entities. In that case, the problem of keeping referential integrity would fall to the database server, and its response would depend on how the `DELETE ON` part of the foreign-key constraint was set up. Databases created by EF Core for these entity classes would, by default, be set to use cascade deletes.

> **NOTE** The `Author` and `Tag` linked to the `Book` aren't deleted because they are not dependent entities of the `Book`; only the `BookAuthor` and `BookTag` linking entities are deleted. This arrangement makes sense because the `Author` and `Tag` might be used on other `Books`.

Section 8.8.1 shows how to configure the way that EF Core handles the deletion of a dependent entity in a relationship. Sometimes, it's useful to stop a principal entity from being deleted if a certain dependent entity is linked to it. In our example Book App, for example, if a customer orders a book, you want to keep that order information even if the book is no longer for sale. In this case, you change the EF Core's on-delete action to `Restrict` and remove the `ON DELETE CASCADE` from the foreign-key constraint in the database so that an error will be raised if an attempt to delete the book is made.

> **NOTE** When you're deleting a principal entity with a dependent entity that has a nullable foreign key (known as an *optional dependent relationship*), subtle differences exist between the way that EF Core handles the delete and the way that the database handles the delete. I explain this situation in section 8.8.1 via a useful table 8.1.

Summary

- Entity instances have a `State`, whose values can be `Added`, `Unchanged`, `Modified`, `Deleted`, or `Detached`. This `State` defines what happens to the entity when `SaveChanges` is called.
- If you `Add` an entity, its `State` is set to `Added`. When you call `SaveChanges`, that entity is written out to the database as a new row.
- You can update a property, or properties, in an entity class by loading the entity class as a tracked entity, changing the property/properties, and calling `SaveChanges`.
- Real-world applications use two types of update scenarios—connected and disconnected state—that affect the way you perform the update.
- EF Core has an `Update` method, which marks the whole of the entity class as updated. You can use this method when you want to update the entity class and have all the data already available to you.
- When you're updating a relationship, you have two options, with different advantages and disadvantages:
  - You can load the existing relationship with the primary entity and update that relationship in the primary entity. EF Core will sort things out from there. This option is easier to use but can create performance issues when you're dealing with large collections.

  - You can create, update, or delete the dependent entity. This approach is harder to get right but typically is faster because you don't need to load any existing relationships.
- To delete an entity from the database, you use the `Remove` method, followed by the `SaveChanges` method.

For EF6.x readers:

- The `Update` method is a welcome new command in EF Core. In EF6.x, you have to use `DbContext.Entry(object).State` to achieve that feature.
- EF Core provides shorthand for `Add`, `Update`, and `Remove`. You can apply any of these commands to the context itself, as in `context.Add(book)`.
- In EF6.x, by default, `SaveChanges` validates the data before adding an entity to or updating an entity in the database. EF Core doesn't run any validation on `SaveChanges`, but it's easy to add back (see chapter 4).
- EF6.x allows you to define many-to-many relationships directly and looks after creating the linking table and managing the rows to make that process work. NET Core 5 adds this feature to EF Core; section 3.4.4 covers this topic.

4

Using EF Core in business logic

This chapter covers

- Understanding business logic and its use of EF Core
- Looking at three types of business logic, from the easy to the complex
- Reviewing each type of business logic, with pros and cons
- Adding a step that validates the data before it's written to the database
- Using transactions to daisy-chain code sequences

Real-world applications are built to supply a set of services, ranging from holding a simple list of things on your computer to managing a nuclear reactor. Every real-world problem has a set of rules, often referred to as *business rules*, or by the more generic name *domain rules*. (This book uses *business rules*.)

The code you write to implement a business rule is known as *business logic* or *domain logic*. Because business rules can be complex, the business logic you write can also be complex. Just think about all the checks and steps that should be done when you order something online.

Business logic can range from a simple check of status to massive artificial intelligence (AI) code, but in nearly all cases, business logic needs access to a database. Although all the approaches in chapters 2 and 3 come into play, the way you apply those EF Core commands in business logic can be a little different, which is why I've written this chapter.

This chapter describes a pattern for handling business logic that compartmentalizes some of the complexity to reduce the load on you, the developer. You'll also learn several techniques for writing different types of business logic that use EF Core to access the database. These techniques range from using software classes for validation to standardizing your business logic's interface to make frontend code simpler. The overall aim is to help you quickly write accurate, understandable, and well-performing business logic.

4.1 The questions to ask and the decisions you need to make before you start coding

Our CRUD code in chapters 2 and 3 adapted and transformed data as it moved into and out of the database. Some of that code was complex, and I showed you the Query Object pattern to make a large query more manageable. Similarly, business logic can range from the simple to the complex.

DEFINITION In this chapter, I use the term *business rule* to represent a human-readable statement of some logic that needs to be implemented, such as "The price of a book cannot be negative." I also use the term *business logic*, which is the code that implements all the business rules needed for a particular feature in the application.

Before you start working on your business logic, you should think about the answers to some questions:

- Do you understand the business rules for the feature you're implementing?
- Do the business rules make sense, or are they incomplete?
- Are there any edge cases or exceptions that you need to cover?
- How can you prove that your implementation matches the business rules?
- How easy will it be to change your code if the business rules change?

4.1.1 The three levels of complexity of your business logic code

When you have some grasp of the business rules you need to implement, you should have some idea of how complex the business logic is. Most of the rules are going to be simple to write, but a few are going to be really complex. The trick is to implement the simple business logic quickly but use a more structured approach for the more complex business logic.

Based on my experiences, I created a list of three levels of complexity of business logic with different patterns for each level: validation, simple, and complex.

The following three sections describe these three levels of complexity and how they will affect the code you write. But be aware that these three patterns aren't strict rules. Some business rules may be simple, but you may decide to use a more complex pattern because it's easier to unit-test. Nevertheless, this list is useful for discussing the types and patterns you can use for writing business logic.

VALIDATION CODE TO CHECK THE DATA USED TO CHANGE AN ENTITY CLASS

When you work with CUD (create, update, and delete) code, as in chapter 3, you may need to check whether the data is in a certain range. The `Review`'s `NumStars` property must be in the range 0 to 5, for example. This sort of test is known as *validation*. For me, validation is the starting point for calling the code *business logic* instead of *CRUD code*.

This type of business logic is common; you see it everywhere (see the nearby sidebar "Does all the business logic code live in a specific business logic layer?" before section 4.2). The simplest validation business logic normally uses `if-then` statements that test data values, but a useful set of attributes called Data Annotations can automate some of the validation code you need to write. (You will see Data Annotations later, in section 4.7.1.)

But there are lots of levels of validation, from simple range checking to validating that a person's driving license is valid via some sort of checking service, which makes defining this starting level of business logic more difficult. But as I said at the start, these levels are guidelines, and the "check person's driving license" validation example would elevate that code to the next level of business logic.

SIMPLE BUSINESS LOGIC (THAT IS, LITTLE OR NO BRANCHING AND EASY TO UNDERSTAND)

The next type is business logic that has little or no branching—that is, few or no `if-then` branching statements and no calling out to other business code. The code is easy to understand because you can read it and see every step that has to be executed in order. A good example would be code to create a book with its authors—that needs code to create the `Book`, then find or create the `Authors`, and finally add the `BookAuthor` linking entity classes. The code is simple, with no branching, but it still takes many lines of code to create a book with its authors.

I am always surprised by how much "simple" business logic like this there is in a real application; typically, I find that a lot of admin functions fall into this category. Therefore, having a simple pattern for building and checking this type of business logic is crucial to you for building your code quickly.

COMPLEX BUSINESS LOGIC (THAT IS, CODE THAT NEEDS SERIOUS EFFORT TO WRITE CORRECTLY)

I call the hardest business logic to write *complex*. There isn't a good definition of this term, but for this type of code, you need to think hard about the problem before you can implement it. Here's a quote from one of the leading books on writing business logic, which portrays the challenge of writing complex business code:

> *The heart of software is its ability to solve domain (business)-related problems for its users. All other features, vital though they may be, support this basic purpose. When the domain*

> *is complex, this is a difficult task, calling for the concentrated effort of talented and skilled people.*
>
> —Eric Evans, *Domain-Driven Design*[1]

This type of business logic is complex enough that I have developed a structured approach that isolates the business logic from the database and the frontend. That way, I can concentrate on the pure business problem—another application of the Separation of Concerns principle (which I talk about in detail in section 5.5.2).

Does all the business logic code live in a specific business logic layer?

No. In real-world applications, especially ones that interact with a human being, you want the user experience to be as good as possible. For that reason, some business logic lives in the presentation layer.

The obvious logic to go into the presentation layer is validation business logic because the earlier you can give feedback to the user, the better. Most frontend systems have built-in features that facilitate validation and good feedback of errors to the user.

Another area is business logic that has many steps. Often, it's better for the user when complex business logic flows are shown as a sequence of pages or steps in a wizard.

Even in the backend of an application, I spread my business logic over several layers (that is, projects) in my Book App. I explain how and why I do that in this chapter.

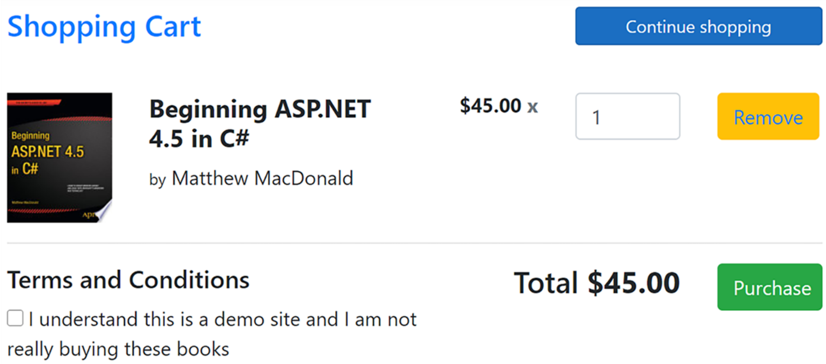
4.2 Complex business logic example: Processing an order for books

I start with the complex business logic because that logic will introduce you to a powerful approach to handling business—an approach taken from Eric Evan's book *Domain-Driven* design, which I quote in the preceding section. First, though, take a look at a complex business feature that you'll want to implement in the Book App. The example you'll build is handling a user's order for books. Figure 4.1 shows the checkout page of the Book App. You're going to implement the code that runs when the user clicks the Purchase button.

NOTE You can try the checkout process by downloading the Book App code from the associated Git repo and running it locally. The Book App uses an HTTP cookie to hold your basket and your identity (which saves you from having to log in). No money is needed; as the terms and conditions text says, you aren't actually going to buy a book.

---

[1] *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional, 2003)



Shopping Cart

Beginning ASP.NET 4.5 in C#

by Matthew MacDonald

$45.00 x 1

Terms and Conditions

☐ I understand this is a demo site and I am not really buying these books

Total $45.00

Figure 4.1 The checkout page of the Book App. When the user clicks the Buy Book button next to a book, the app adds the book to their basket and then displays the Checkout page, which shows all the books in the user's basket. Clicking the Purchase button calls the business logic that creates the order, which is the code we are going to write.

4.3 Using a design pattern to implement complex business logic

Before you start writing code to process an order, take a look at a pattern that will help you write, test, and performance-tune your business logic. The pattern is based on the Domain-Driven Design (DDD) concepts expounded by Eric Evans, but the business logic code isn't inside the entity classes. This pattern is known as a *transactions script* or *procedural* pattern of business logic because the code is contained in a standalone method.

This procedural pattern is easy to understand and uses the basic EF Core commands you have already seen. But many people see the procedural approach as being a DDD antipattern, known as an *anemic domain model* (see http://mng.bz/nM7g). Later, in part 3 of this book, you will extend this approach to a fully DDD design.

This section and chapter 13 present my interpretation of Evans' DDD approach and plenty of other ways to apply DDD with EF. Although I offer my approach, which I hope will help you, don't be afraid to look for other approaches.

4.3.1 Five guidelines for building business logic that uses EF Core

The following list explains the five guidelines that make up the business logic pattern you'll be using in this chapter. Most of the pattern comes from DDD concepts, but some is the result of writing lots of complex business logic and seeing areas to improve:

- *The business logic has first call on how the database structure is defined.* Because the problem you're trying to solve (which Evans calls the *domain model*) is the heart of the problem, the logic should define the way the whole application is designed.

Therefore, you try to make the database structure and the entity classes match your business logic data needs as much as you can.

- *The business logic should have no distractions*. Writing the business logic is difficult enough in itself, so you isolate it from all the other application layers other than the entity classes. When you write the business logic, you must think only about the business problem you're trying to fix. You leave the task of adapting the data for presentation to the service layer in your application.
- *Business logic should think that it's working on in-memory data*. Evans taught me to write business logic as though the data is in memory. You need to have some *load* and *save* parts, of course, but for the core of your business logic, treat the data (as much as is practical) as though it's a normal, in-memory class or collection.
- *Isolate the database access code into a separate project*. This rule came out of writing an e-commerce application with complex pricing and delivery rules. Before, I'd used EF directly in my business logic, but I found that it was hard to maintain and difficult to performance-tune. Instead, you should use another project, a companion to the business logic, to hold all the database access code.
- *The business logic shouldn't call EF Core's* `SaveChanges` *directly*. You should have a class in the service layer (or a custom library) whose job it is to run the business logic. If there are no errors, this class calls `SaveChanges`. The main reason for this rule is to have control of whether to write out the data, but it has other benefits, which I'll describe in section 4.4.5.

Figure 4.2 shows the application structure you'll create to help you apply these guidelines when implementing business logic. In this case, you'll add two new projects to the original Book App structure described in chapter 2:

- The pure business logic project, which holds the business logic classes that work on the in-memory data provided by the companion business database access methods.
- The business database access project, which provides a companion class for each pure business logic class that needs database access. Each companion class makes the pure business logic class think that it's working on an in-memory set of data.

Figure 4.2 has five numbers, with comments, that match the five guidelines.

4.4 Implementing the business logic for processing an order

Now that I've described the business need, with its business rules, and the pattern you're going to use, you're ready to write code. The aim is to break the implementation into smaller steps that focus on specific parts of the problem at hand. You'll see how this business logic pattern helps you to focus on each part of the implementation in turn.

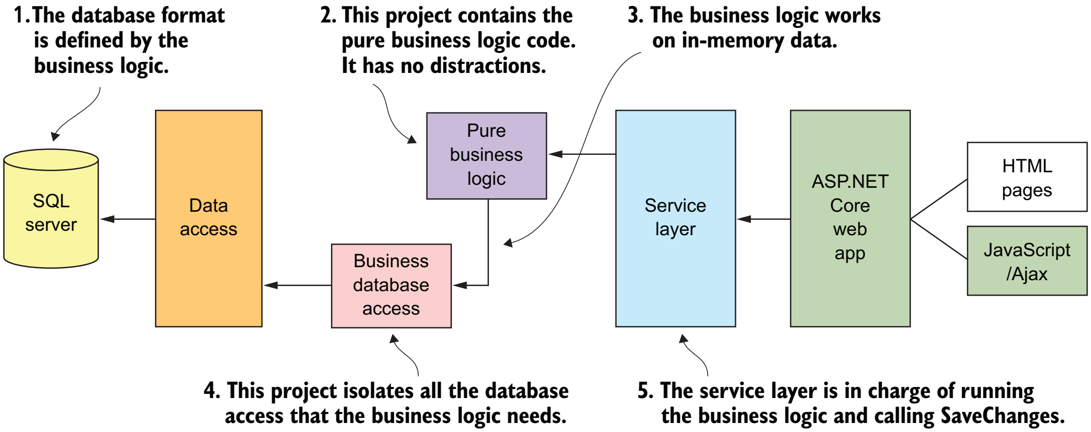


Figure 4.2 The projects inside our Book App, with two new projects for handling complex business logic. The "Pure business logic" project contains the isolated business logic, which thinks it is working on an in-memory set of classes. The "Business database access" project provides an interface that the pure business logic can use to access the database. The service layer's job is to adapt the data from the ASP.NET Core application to send to the pure business logic in the form it wants that data to be in and call the final `SaveChanges` to save if the business logic doesn't report any errors.

You're going to implement the code in sections that match the five guidelines listed in section 4.3.1. At the end, you'll see how this combined code is called from the ASP.NET Core application that the Book App is using.

4.4.1 Guideline 1: Business logic has first call on defining the database structure

This guideline says that the design of the database should follow the business needs—in this case, represented by six business rules. Only three of these rules are relevant to the database design:

- An order must include at least one book (implying that there can be more).
- The price of the book must be copied to the order, because the price could change later.
- The order must remember the person who ordered the books.

These three rules dictates an `Order` entity class that has a collection of `LineItem` entity classes—a one-to-many relationship. The `Order` entity class holds the information about the person placing the order, and each `LineItem` entity class holds a reference to the book order, how many, and at what price.

Figure 4.3 shows what these two tables, LineItem and Orders, look like in the database. To make the image more understandable, I show the Books table (in gray) that each LineItem row references.

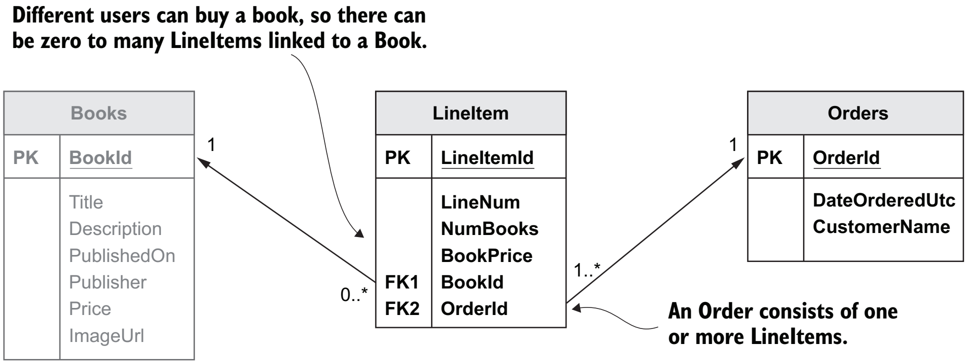


Figure 4.3 The new LineItem and Orders tables added to allow orders for books to be taken. There is one Orders row per purchase, with a lineItem row for each Book in the order.

NOTE The Orders table name is plural because you added a `DbSet<Order> Orders` property to the application's DbContext, and by default, EF Core uses the property name, Orders, as the table name. You haven't added a property for the `LineItem` entity class because it's accessed via the Order's relational link. In that case, EF Core, by default, uses the class name, `LineItem`, as the table name. You can set the table name to a specific name; see section 7.11.1.

4.4.2 Guideline 2: Business logic should have no distractions

Now you're at the heart of the business logic code, and the code here will do most of the work. This code is going to be the hardest part of the implementation that you write, but you want to help yourself by cutting off any distractions. That way, you can stay focused on the problem.

To do so, write the pure business code with reference to only two other parts of the system: the entity classes shown in figure 4.3 (`Order`, `LineItem`, and `Book`) and your companion class that will handle all the database accesses. Even with this minimization of scope, you're still going to break the job into a few parts.

CHECKING FOR ERRORS AND FEEDING THEM BACK TO THE USER: VALIDATION

The business rules contain several checks, such as "The Terms and Conditions box must be ticked." The rules also say that you need to give good feedback to the user so that they can fix any problems and complete their purchase. These sorts of checks, called *validation*, are common throughout an application.

You have two main approaches to handling the passing of errors back up to higher levels. One is to throw an exception when an error occurs, and the other is to pass back the errors to the caller via a status interface. Each option has its own advantages and disadvantages. This example uses the second approach: passing the errors back in some form of status class to the higher level to check.

To help, you'll create a small abstract class called `BizActionErrors`, shown in listing 4.1. This class provides a common error-handling interface for all your business logic. The class contains a C# method called `AddError` that the business logic can call to add an error and an *immutable list* (a list that can't be changed) called `Errors`, which holds all the validation errors found while running the business logic.

You'll use a class called `ValidationResult` to store each error because it's the standard way of returning errors with optional, additional information on the exact property the error was related to. Using the `ValidationResult` class instead of a simple string fits with another validation method you'll add later in this chapter.

Listing 4.1 Abstract base class providing error handling for your business logic

```
public abstract class BizActionErrors
{
    private readonly List<ValidationResult> _errors
        = new List<ValidationResult>();

    public IImmutableList<ValidationResult>
        Errors => _errors.ToImmutableList();

    public bool HasErrors => _errors.Any();

    protected void AddError(string errorMessage,
        params string[] propertyNames)
    {
        _errors.Add( new ValidationResult
            (errorMessage, propertyNames));
    }
}
```

Abstract class that provides error handling for business logic

Holds the list of validation errors privately

Provides a public, immutable list of errors

Creates a bool HasErrors to make checking for errors easier

Allows a simple error message, or an error message with properties linked to it, to be added to the errors list

Validation result has an error message and a possibly empty list of properties it's linked to

Using this abstract class means that your business logic is easier to write and all your business logic has a consistent way of handling errors. The other advantage is that you can change the way errors are handled internally without having to change any of your business logic code.

Your business logic for handling an order does a lot of validation, which is typical for an order, because it often involves money. Other business logic may not do any validation, but the base class `BizActionErrors` will automatically return a `HasErrors` of `false`, which means that all business logic can be dealt with in the same way.

4.4.3 Guideline 3: Business logic should think that it's working on in-memory data

Now you'll start on the main class: `PlaceOrderAction`, which contains the pure business logic. This class relies on the companion class `PlaceOrderDbAccess` to present the data as an in-memory set (in this case, a dictionary) and to write the created order to the database. Although you're not trying to hide the database from the pure business logic, you do want it to work as though the data is normal .NET classes.

Listing 4.2 shows the `PlaceOrderAction` class, which inherits the abstract class `BizActionErrors` to handle returning error messages to the user. It also uses two methods that the companion `PlaceOrderDbAccess` class provides:

- `FindBooksByIdsWithPriceOffers`—Takes the list of `BookId`s and returns a dictionary with the `BookId` as the key and the `Book` entity class as the value and any associated `PriceOffer`s
- `Add`—Adds the `Order` entity class with its `LineItem` collection to the database

Listing 4.2 `PlaceOrderAction` class with build-a-new-order business logic

```
public class PlaceOrderAction :
    BizActionErrors,
    IBizAction<PlaceOrderInDto,Order>
{
    private readonly IPlaceOrderDbAccess _dbAccess;

    public PlaceOrderAction(IPlaceOrderDbAccess dbAccess)
    {
        _dbAccess = dbAccess;
    }

    public Order Action(PlaceOrderInDto dto)
    {
        if (!dto.AcceptTAndCs)
        {
            AddError(
"You must accept the T&Cs to place an order.");
            return null;
        }
        if (!dto.LineItems.Any())
        {
            AddError("No items in your basket.");
            return null;
        }

        var booksDict =
            _dbAccess.FindBooksByIdsWithPriceOffers
                 (dto.LineItems.Select(x => x.BookId));
        var order = new Order
        {
            CustomerId = dto.UserId,
            LineItems =
                FormLineItemsWithErrorChecking
                     (dto.LineItems, booksDict)
        };

        if (!HasErrors)
            _dbAccess.Add(order);

        return HasErrors ? null : order;
    }
```

The BizActionErrors class provides error handling for the business logic.

The IBizAction interface makes the business logic conform to a standard interface.

The PlaceOrderAction uses PlaceOrderDbAccess class to handle database accesses.

This method is called by the BizRunner to execute this business logic.

Some basic validation

The PlaceOrderDbAccess class finds all the bought books, with optional PriceOffers.

Creates the Order, using FormLineItemsWithErrorChecking to create the LineItems

Adds the order to the database only if there are no errors

If there are errors, returns null; otherwise, returns the order

```
    private List<LineItem>  FormLineItemsWithErrorChecking
        (IEnumerable<OrderLineItem> lineItems,
         IDictionary<int,Book> booksDict)

    {
        var result = new List<LineItem>();
        var i = 1;

        foreach (var lineItem in lineItems)
        {
            if (!booksDict.
                ContainsKey(lineItem.BookId))
                    throw new InvalidOperationException
("An order failed because book, " +
 $"id = {lineItem.BookId} was missing.");

            var book = booksDict[lineItem.BookId];
            var bookPrice =
                book.Promotion?.NewPrice ?? book.Price;
            if (bookPrice <= 0)
                AddError(
$"Sorry, the book '{book.Title}' is not for sale.");
            else
            {
                //Valid, so add to the order
                result.Add(new LineItem
                {
                    BookPrice = bookPrice,
                    ChosenBook = book,
                    LineNum = (byte)(i++),
                    NumBooks = lineItem.NumBooks
                });
            }
        }
        return result;
    }
}
```

This private method handles the creation of each LineItem for each book ordered.

Goes through each book type that the person ordered

Treats a missing book as a system error and throws an exception

Calculates the price at the time of the order

More validation that checks whether the book can be sold

Everything is OK, so create the LineItem entity class with the details.

Returns all the LineItems for this order

You'll notice that you add another validation check to ensure that the book the user selected is still in the database. This check wasn't in the business rules, but it could occur, especially if malicious inputs were provided. In this case, you make a distinction between errors that the user can correct, which are returned by the `Errors` property, and system errors (in this case, a missing book), for which you throw an exception that the system should log.

You may have seen at the top of the class that you apply an interface in the form of `IBizAction<PlaceOrderInDto,Order>`. This interface ensures that this business logic class conforms to a standard interface that you use across all your business logic. You'll see this in section 4.7.1, when you create a generic class to run and check the business logic.

4.4.4 Guideline 4: Isolate the database access code into a separate project

Our guideline says to put all the database access code that the business logic needs in a separate, companion class. This technique ensures that all the database accesses are in one place, making testing, refactoring, and performance tuning much easier.

Another benefit that a reader of my blog noted is that this guideline can help if you're working with an existing, older database. In this case, the database entities may not be a good match for the business logic you want to write. If so, you can use the `BizDbAccess` methods as an *Adapter pattern* that converts the older database structure to a form more easily processed by your business logic.

> **DEFINITION** The *Adapter pattern* converts the interface of a class to another interface that the client expects. This pattern lets classes work together that couldn't otherwise do so because of incompatible interfaces. See https://sourcemaking.com/design_patterns/adapter.

You make sure that your pure business logic, class `PlaceOrderAction`, and business database access class `PlaceOrderDbAccess` are in separate projects. That approach allows you to exclude any EF Core libraries from the pure business logic project, ensuring that all database access is done via the companion class, `PlaceOrderDbAccess`. In my own projects, I split the entity classes into a separate project from the EF code. Then my pure business logic project doesn't have the `Microsoft.EntityFrameworkCore` NuGet library, so my business logic can't execute any database commands directly; it has to rely on the `PlaceOrderDbAccess` class for any data accesses.

For simplicity, the example code holds the entity classes in the same project as the application's `DbContext`. Listing 4.3 shows our `PlaceOrderDbAccess` class, which implements two methods to provide the database accesses that the pure business logic needs:

- The `FindBooksByIdsWithPriceOffers` method, which finds and loads each `Book` entity class, with any optional `PriceOffer`.
- The `Add` method, which adds the finished `Order` entity class to the application's DbContext property, `Orders`, so that it can be saved to the database after EF Core's `SaveChanges` method is called.

Listing 4.3 `PlaceOrderDbAccess`, which handles all the database accesses

```
public class PlaceOrderDbAccess : IPlaceOrderDbAccess
{
    private readonly EfCoreContext _context;

    public PlaceOrderDbAccess(EfCoreContext context)
    {
        _context = context;
    }

    public IDictionary<int, Book>
        FindBooksByIdsWithPriceOffers
```

All the BizDbAccess need the application's DbContext to access the database.

This method finds all the books that the user wants to buy.

```
        (IEnumerable<int> bookIds)
{
    return _context.Books
        .Where(x => bookIds.Contains(x.BookId))
        .Include(r => r.Promotion)
        .ToDictionary(key => key.BookId);
}

public void Add(Order newOrder)
{
    _context.Add(newOrder);
}
}
```

The BizLogic hands a collection of BookIds, which the checkout has provided.

Finds a book for each Id, using the LINQ Contains method to find all the keys

Includes any optional promotion, which the BizLogic needs for working out the price

Returns the result as a dictionary to make it easier for the BizLogic to look them up

This method adds the new order to the DbContext's Orders DbSet collection.

The `PlaceOrderDbAccess` class implements an interface called `IPlaceOrderDbAccess`, which is how the `PlaceOrderAction` class accesses this class. In addition to helping with dependency injection, which is covered in chapter 5, using an interface allows you to replace the `PlaceOrderDbAccess` class with a test version—a process called *stubbing* or *mocking*—when you're unit-testing the `PlaceOrderAction` class. Section 17.7 covers this topic in more detail.

4.4.5 Guideline 5: Business logic shouldn't call EF Core's SaveChanges

The final rule says that the business logic doesn't call EF Core's `SaveChanges`, which would update the database directly. There are a few reasons for this rule:

- You consider the service layer to be the main orchestrator of database accesses: it's in command of what gets written to the database.
- The service layer calls `SaveChanges` only if the business logic returns no errors.

To help you run your business logic, I've built a series of simple classes that I use to run any business logic; I call these classes `BizRunners`. They're generic classes, able to run business logic with different input and output types. Different variants of the `BizRunner` can handle different input/output combinations and async methods (chapter 5 covers async/await with EF Core), as well as some with extra features, which are `PlaceOrderAction` (covered in section 4.7.3).

Each `BizRunner` works by defining a generic interface that the business logic must implement. Your class in the `BizLogic` project runs an action that expects a single input parameter of type `PlaceOrderInDto` and returns an object of type `Order`. Therefore, the `PlaceOrderAction` class implements the interface as shown in the following listing, but with its input and output types (`IBizAction<PlaceOrderInDto,Order>`).

Listing 4.4 The interface that allows the `BizRunner` to execute business logic

```
public interface IBizAction<in TIn, out TOut>
{
    IImmutableList<ValidationResult>
        Errors { get; }
    bool HasErrors { get; }
```

The BizAction uses the TIn and a TOut to define the input and output of the Action method.

Returns the error information from the business logic

```
    TOut Action(TIn dto);
}
```

The action that the BizRunner will call

When you have the business logic class implement this interface, the `BizRunner` knows how to run that code. The `BizRunner` itself is small, as you'll see in the following listing, which shows that it's called `RunnerWriteDb<TIn, TOut>`. This `BizRunner` variant is designed to work with business logic that has an input, provides an output, and writes to the database.

Listing 4.5 The `BizRunner` that runs the business logic and returns a result or errors

```
public class RunnerWriteDb<TIn, TOut>
{
    private readonly IBizAction<TIn, TOut> _actionClass;
    private readonly EfCoreContext _context;

    public IImmutableList<ValidationResult>
        Errors => _actionClass.Errors;
    public bool HasErrors => _actionClass.HasErrors;

    public RunnerWriteDb(
        IBizAction<TIn, TOut> actionClass,
        EfCoreContext context)
    {
        _context = context;
        _actionClass = actionClass;
    }

    public TOut RunAction(TIn dataIn)
    {
        var result = _actionClass.Action(dataIn);
        if (!HasErrors)
            _context.SaveChanges();
        return result;
    }
}
```

Error information from the business logic is passed back to the user of the BizRunner.

Handles business logic that conforms to the IBizAction<TIn, TOut> interface

Calls RunAction in your service layer or in your presentation layer if the data comes back in the right form

Runs the business logic you gave it

If there are no errors, calls SaveChanges to execute any add, update, or delete methods

Returns the result that the business logic returned

The `BizRunner` pattern hides the business logic and presents a common interface/API that other classes can use. The caller of the `BizRunner` doesn't need to worry about EF Core, because all the calls to EF Core are in the `BizDbAccess` code or in the `BizRunner`. That fact in itself is reason enough to use the `BizRunner` pattern, but as you'll see later, this pattern allows you to create other forms of `BizRunner` that add extra features.

> **NOTE** You may want to check out an open-source library I created, called `EfCore.GenericBizRunner`, which provides the same features as the `BizRunner` but in a library. It uses generic classes that run your business logic without requiring you to write extra code. See http://mng.bz/vz7J for more information.

One important point about the `BizRunner` is that it should be the only method allowed to call `SaveChanges` during the lifetime of the application's DbContext. Why? The business logic isn't thinking about the database, so it's quite normal for the business logic to add or update an entity class at any time, and an error may be found later. To stop the changes made before the error was found from being written to the database, you're relying on `SaveChanges` to *not* be called during the lifetime of the application's `DbContext`.

In an ASP.NET application, controlling the lifetime of the application's `DbContext` is fairly easy to manage, because a new instance of the application's `DbContext` is created for each HTTP request. In longer-running applications, this situation is a problem. In the past, I've avoided it by making the `BizRunner` create a new, hidden instance of the application's `DbContext` so that I can be sure no other code is going to call `SaveChanges` on that `DbContext` instance.

4.4.6 Putting it all together: Calling the order-processing business logic

Now that you've learned all the parts of this complex business logic pattern, you're ready to see how to call this code. Listing 4.6 shows the `PlaceOrderService` class in the service layer, which calls the `BizRunner` to execute the `PlaceOrderAction` that does the order processing.

> **NOTE** I use an HTTP cookie to hold the user's selection of what books they want to buy. I refer to this cookie as the *basket cookie*. This cookie works because an HTTP cookie can store a small amount of data on the user's computer. I use ASP.NET Core's cookie features to access the user's basket cookie. For more information, see http://mng.bz/4ZNa.

If the business logic is successful, the code clears the basket cookie and returns the `Order` entity class key so that a confirmation page can be shown to the user. If the order fails, it doesn't clear the basket cookie, and the checkout page is shown again, with the error messages, so that the user can correct any problems and retry.

Listing 4.6 The `PlaceOrderService` class that calls the business logic

```
public class PlaceOrderService
{
    private readonly BasketCookie _basketCookie;

    private readonly
        RunnerWriteDb<PlaceOrderInDto, Order> _runner;
    public IImmutableList<ValidationResult>
        Errors => _runner.Errors;

    public PlaceOrderService(
        IRequestCookieCollection cookiesIn,
        IResponseCookies cookiesOut,
        EfCoreContext context)
    {
```

This class handles the basket cookie, which contains the user-selected books.

Defines the input, PlaceOrderInDto, and output, Order, of this business logic

Holds any errors sent back from the business logic

The constructor takes in the cookie in/out data, plus the application's DbContext.

```
        _basketCookie = new BasketCookie(
            cookiesIn, cookiesOut);
        _runner =
            new RunnerWriteDb<PlaceOrderInDto, Order>(
                new PlaceOrderAction(
                    new PlaceOrderDbAccess(context)),
                context);
    }

    public int PlaceOrder(bool acceptTAndCs)
    {
        var checkoutService = new CheckoutCookieService(
            _basketCookie.GetValue());

        var order = _runner.RunAction(
            new PlaceOrderInDto(acceptTAndCs,
            checkoutService.UserId,
            checkoutService.LineItems));

        if (_runner.HasErrors) return 0;

        checkoutService.ClearAllLineItems();
        _basketCookie.AddOrUpdateCookie(
            checkoutService.EncodeForCookie());

        return order.OrderId;
    }
}
```

Creates a BasketCookie using the cookie in/out data from ASP.NET Core

Creates the BizRunner, with the business logic, that is to be run

This method is the one to call when the user clicks the Purchase button.

CheckoutCookieService is a class that encodes/decodes the basket data.

Runs the business logic with the data it needs from the basket cookie

If the business logic has errors, it returns immediately. The basket cookie is not cleared.

The order was placed successfully, so it clears the basket cookie.

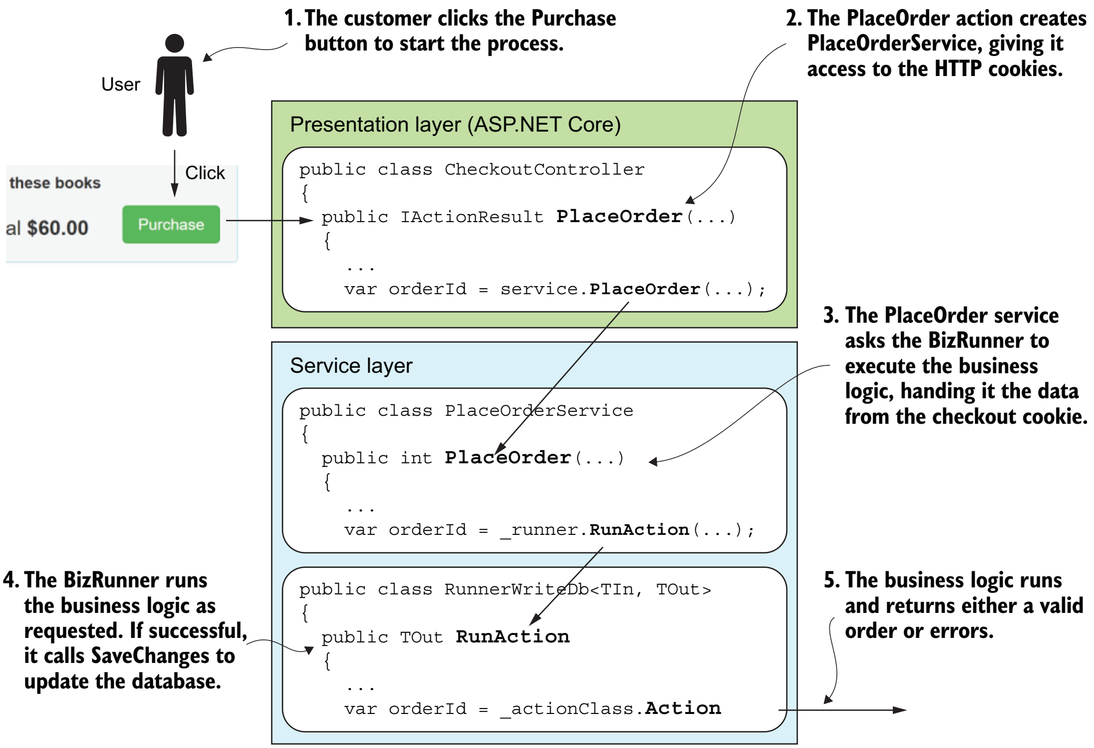
Returns the OrderId, which allows ASP.NET to confirm the order details to the user

In addition to running the business logic, this class acts as an Adapter pattern; it transforms the data from the basket cookie into a form that the business logic accepts, and on a successful completion, it extracts the `Order` entity class's primary key, `OrderId`, to send back to the ASP.NET Core presentation layer.

This Adapter-pattern role is typical of the code that calls the business logic because a mismatch often occurs between the presentation layer format and the business logic format. This mismatch can be small, as in this example, but you're likely to need to do some form of adaptation in all but the simplest calls to your business logic. That situation is why my more-sophisticated EfCore.GenericBizRunner library has a built-in Adapter pattern feature.

4.4.7 Placing an order in the Book App

Now that we've covered the business logic for processing an order, the `BizRunner`, and the `PlaceOrderService` that executes the business logic, let's see how to use this logic in the context of the Book App. Figure 4.4 shows the process, from the user clicking the Purchase button through running the business logic and returning a result. I don't go into the presentation code in detail here, as this chapter is about using EF Core in business logic, but I do cover some of it in chapter 5, which is about using EF Core in ASP.NET Core applications.



```
public class CheckoutController
{
  public IActionResult PlaceOrder(...)
  {
    ...
    var orderId = service.PlaceOrder(...);
```

```
public class PlaceOrderService
{
  public int PlaceOrder(...)
  {
    ...
    var orderId = _runner.RunAction(...);
```

```
public class RunnerWriteDb<TIn, TOut>
{
  public TOut RunAction
  {
    ...
    var orderId = _actionClass.Action
```

1. The customer clicks the Purchase button to start the process.

2. The PlaceOrder action creates PlaceOrderService, giving it access to the HTTP cookies.

3. The PlaceOrder service asks the BizRunner to execute the business logic, handing it the data from the checkout cookie.

4. The BizRunner runs the business logic as requested. If successful, it calls SaveChanges to update the database.

5. The business logic runs and returns either a valid order or errors.

Figure 4.4 The series of steps from the user's clicking the Purchase button to the service layer, where the `BizRunner` executes the business logic to process the order

From the click of the Purchase button in figure 4.4, the ASP.NET Core action, `PlaceOrder`, in the `CheckoutController` is executed. This action creates a class called `PlaceOrderService` in the service layer, which holds most of the Adapter pattern logic. The caller provides that class with read/write access to the cookies, as the checkout data is held in an HTTP cookie on the user's device.

You saw the `PlaceOrderService` class in listing 4.6. Its `PlaceOrder` method extracts the checkout data from the HTTP cookie and creates a DTO in the form that the business logic needs. Then it calls the generic `BizRunner` to run the business logic that it needs to execute. When the `BizRunner` has returned from the business logic, two routes are possible:

- *The order was successfully placed (no errors)*. In this case, the `PlaceOrder` method cleared the basket cookie and returned the `OrderId` of the placed order, so the ASP.NET Core code could show a confirmation page with a summary of the order.
- *The order was unsuccessful (errors present)*. In this case, the `PlaceOrder` method returned immediately to the ASP.NET Core code, which detected errors,

redisplayed the checkout page, and added the error messages so that the user could rectify the errors and try again.

NOTE You can try the checkout process by downloading the book app code and running it locally to see the results. To try the error path, don't check the Terms and Conditions (T&C) box.

4.4.8 The pros and cons of the complex business logic pattern

I have used this pattern for complex business logic for years. I think that it's an excellent approach overall, but it's code-heavy, by which I mean that you have to write extra structural code to implement it. Therefore, I use it only for complex business logic. The following sections cover the pros and cons in detail.

ADVANTAGES OF THIS PATTERN

This pattern follows the DDD approach, which is well respected and widely used. It keeps the business logic "pure" in that it doesn't know about the database, which has been hidden via the `BizDbAccess` methods that provide a per-business logic repository. Also, the `BizDbAccess` class allows you to test your business logic without using a database, as your unit tests can provide a replacement class (known as a stub or mock) that can provide test data as required.

DISADVANTAGES OF THIS PATTERN

The key disadvantage is you have to write more code to separate the business logic from the database accesses, which takes more time and effort. If the business logic is simple, or if most of the code works on the database, the effort of creating a separate class to handle database accesses isn't worthwhile.

4.5 Simple business logic example: ChangePriceOfferService

For my example of my simple business logic, you are going to build business logic to handle the addition or removal of a price promotion for a book. This example has business rules, but as you will see, those rules are bound up with a lot of database accesses. The rules are

- If the `Book` has a `PriceOffer`, the code should delete the current `PriceOffer` (remove the price promotion).
- If the `Book` doesn't have a `PriceOffer`, we add a new price promotion.
- If the code is adding a price promotion, the `PromotionalText` must not be null or empty.

As you'll see in section 4.5.2, the code is a mixture of business rules and database accesses, which I define as a simple business logic type.

4.5.1 My design approach for simple business logic

For simple business logic, I want to have minimal extra structure because I have deemed that the business logic is simple enough and/or so interlinked with the database accesses that it doesn't need to be isolated. As a result, the five guidelines stated in section 4.3.1 are not used, making the code quicker to build. The downside is that the business logic is mixed with other code, which can make the business logic difficult to understand and harder to unit-test—trade-offs that you have to manage for faster development.

Typically, I place simple business logic in the service layer, not the BizLogic layer, because my simple business logic needs access to the application's DbContext, and the BizLogic layer does not allow that access. I generally place my simple business logic with CRUD classes that work on the same feature. In the `ChangePriceOfferService` example, I place the `ChangePriceOfferService` class in the `AdminServices` folder alongside the other CRUD services.

4.5.2 Writing the ChangePriceOfferService code

The `ChangePriceOfferService` class contains two methods: a `GetOriginal` method, which is a simple CRUD command to load the `PriceOffer`, and an `AddRemovePriceOffer` method that handles the creation or removal of the `PriceOffer` class for a `Book`. The second method contains business logic and is shown in the following listing.

Listing 4.7 `AddRemovePriceOffer` **method in** `ChangePriceOfferService`

```
public ValidationResult AddRemovePriceOffer(PriceOffer promotion)
   {
    var book = _context.Books
        .Include(r => r.Promotion)
        .Single(k => k.BookId
                     == promotion.BookId);

    if (book.Promotion != null)
    {
        _context.Remove(book.promotion);
        _context.SaveChanges();
        return null;
    }

    if (string.IsNullOrEmpty(promotion.PromotionalText))
    {
        return new ValidationResult(
            "This field cannot be empty",
            new []{ nameof(PriceOffer.PromotionalText)});
    }

    book.Promotion = promotion;
```

This method deletes a PriceOffer if present; otherwise, it adds a new PriceOffer.

Loads the book, with any existing promotion

If the book has an existing Promotion, removes that promotion

Deletes the PriceOffer entry that was linked to the chosen book

Returns null, which means that the method finished successfully

Validation check. The PromotionalText must contain some text.

Returns an error message, with the property name that was incorrect

Assigns the new PriceOffer to the selected book

```
        _context.SaveChanges();
        return null;
    }
```

The SaveChanges method updates the database.

The addition of a new price promotion was successful, so the method returns null.

4.5.3 The pros and cons of this business logic pattern

You have written some business logic implemented in a different way from the more complex business logic for processing an order, which I have described as simple business logic. The major differences between the simple business logic and the complex business logic are

- The simple business logic didn't follow the DDD-inspired guidelines from section 4.3.1. In particular, it didn't isolate the database access from the business logic.
- The simple business logic was placed in the service layer (instead of in the BizLogic layer) alongside the CRUD services related to the basket.

This pattern has the following pros and cons.

ADVANTAGES OF THIS PATTERN

This pattern has little or no set structure, so you can write the code in the simplest way to archive the required business goal. Normally, the code will be shorter than the complex business pattern, which has extra classes to isolate the business logic from the database.

The business logic is also self-contained, with all the code in one place. Unlike the complex business logic example, this business logic handles everything. It doesn't need a `BizRunner` to execute it, for example, because the code calls `SaveChanges` itself, making it easier to alter, move, and test because it doesn't rely on anything else.

Also, by putting the business logic classes in the service layer, I can group these simple business logic services in the same folder as the CRUD services related to this business feature. As a result, I can find all the basic code for a feature quickly, because the complex business code is in another project.

DISADVANTAGES OF THIS PATTERN

You don't have the DDD-inspired approach of the complex business logic pattern to guide you, so the onus is on you to design the business logic in a sound way. Your experience will aid you in picking the best pattern to use and writing the correct code. Simplicity is the key here. If the code is easy to follow, you got it right; otherwise, the code is too complex and needs to follow the complex business logic pattern.

4.6 Validation business logic example: Adding review to a book, with checks

The final example is an upgrade to a CRUD example in chapter 3. In that chapter, you added a `Review` to a `Book`. But that version was missing some vital business rules:

- The `NumStars` property must be between 0 and 5.
- The `Comment` property should have some text in it.

In this section, you are going to update the CRUD code to add a validation check. The following listing shows you the improved `AddReviewWithChecks` method but concentrates on the validation part.

Listing 4.8 The improved CRUD code with business validation checks added

This method adds a review to a book, with validation checks on the data.

```
public IStatusGeneric AddReviewWithChecks(Review review)
{
    var status = new StatusGenericHandler();
    if (review.NumStars < 0 || review.NumStars > 5)
        status.AddError("This must be between 0 and 5.",
            nameof(Review.NumStars));
    if (string.IsNullOrWhiteSpace(review.Comment))
        status.AddError("Please provide a comment with your review.",
            nameof(Review.Comment));
    if (!status.IsValid)
        return status;

    var book = _context.Books
        .Include(r => r.Reviews)
        .Single(k => k.BookId
                     == review.BookId);
    book.Reviews.Add(review);
    _context.SaveChanges();
    return status;
}
```

Creates a status class to hold any errors

Adds an error to the status if the star rating is in the correct range

This second check ensures that the user provided some sort of comment.

If there are any errors, the method returns immediately with those errors.

The CRUD code that adds a review to a book

Returns the status, which will be valid if no errors were found

NOTE The `IStatusGeneric` interface and `StatusGenericHandler` class used in listing 4.8 come from a NuGet package called `GenericServices.StatusGeneric`. This library provides a simple but comprehensive way to return a good/bad status that matches the .NET Core validation approach. The companion NuGet package, called `EfCore.GenericServices.AspNetCore`, provides ways to convert the `IStatusGeneric` status to ASP.NET Core's `ModelState` Razor-based pages or to HTTP returns for Web API Controllers.

This method is a CRUD method with business validation added, which is typical of this type of business logic. In this case, you used `if-then` code to check the property, but you could use `DataAnnotations` instead. As I said earlier, this type of validation is typically done in the frontend, but duplicating the validation of sensitive data in the backend code can make the application more robust. Later, in section 4.7.1, I show you how you can validate data before it's written to the database, which gives you another option.

4.6.1 *The pros and cons of this business logic pattern*

The validation business logic is the CRUD services you saw in chapter 3, enhanced by adding validation checks. Therefore, I place validation business logic classes in the service layer alongside the other CRUD services.

ADVANTAGES OF THIS PATTERN

You are already aware of the CRUD services from chapter 3, so you don't need to learn another pattern—only add validation checks and return a status. Like many other people, however, I consider these validation business logic classes to be the same as CRUD services with some extra checks in them.

DISADVANTAGES OF THIS PATTERN

The only disadvantage is that you need to do something with the status that the pattern returns, such as redisplaying the input form with an error message. But that's the downside of providing extra validation rather than the validation business logic design.

4.7 Adding extra features to your business logic handling

This pattern for handling business logic makes it easier to add extra features to your business logic handling. In this section, you'll add two features:

- Entity class validation to `SaveChanges`
- Transactions that daisy-chain a series of business logic code

These features use EF Core commands that aren't limited to business logic. Both features could be used in other areas, so you might want to keep them in mind when you're working on your application.

4.7.1 Validating the data that you write to the database

I have already talked about validating data before it gets to the database, but this section shows you how to add validation when writing to the database. NET contains a whole ecosystem to validate data, to check the value of a property against certain rules (such as checking whether an integer is within the range of 1 to 10 or a string isn't longer than 20 characters). This ecosystem is used by many of Microsoft's frontend systems.

> **EF6** If you're scanning for EF6.x changes, read the next paragraph. EF Core's `SaveChanges` doesn't validate the data before writing to the database, but this section shows how to add it back.

In the previous version of EF (EF6.x), data that was being added or updated was validated by default before being written to the database. In EF Core, which is designed to be more lightweight and faster, no validation occurs when adding data to or updating the database. The idea is that the validation is often done at the frontend, so why repeat the validation?

As you've seen, the business logic contains lots of validation code, and it's often useful to move this code into the entity classes as a validation check, especially if the error is related to a specific property in the entity class. This example is another case of breaking a complex set of rules into several parts.

Listing 4.9 moves the test to check that the book is for sale into the validation code, rather than having to do it in the business logic. The listing also adds two new validation checks to show you the various forms that validation checks can take, making the example more comprehensive.

Figure 4.5 shows the `LineItem` entity class with two types of validation added. The first type is a `[Range(min,max)]` attribute, known as Data Annotations (see section 7.4), which is added to the `LineNum` property. The second validation method to apply is the `IValidatableObject` interface. This interface requires you to add a method called `IValidatableObject.Validate`, in which you can write your own validation rules and return errors if those rules are violated.

Listing 4.9 Validation rules applied to the `LineNum` entity class

```
public class LineItem : IValidatableObject
{
    public int LineItemId { get; set; }

    [Range(1,5, ErrorMessage =
        "This order is over the limit of 5 books.")]
    public byte LineNum { get; set; }

    public short NumBooks { get; set; }

    public decimal BookPrice { get; set; }

    // relationships

    public int OrderId { get; set; }
    public int BookId { get; set; }

    public Book ChosenBook { get; set; }

    IEnumerable<ValidationResult> IValidatableObject.Validate
        (ValidationContext validationContext)
    {
        var currContext =
            validationContext.GetService(typeof(DbContext));

        if (ChosenBook.Price < 0)
            yield return new ValidationResult(
$"Sorry, the book '{ChosenBook.Title}' is not for sale.");

        if (NumBooks > 100)
            yield return new ValidationResult(
    "If you want to order a 100 or more books"+
" please phone us on 01234-5678-90",
                new[] { nameof(NumBooks) });
    }
}
```

The IValidatableObject interface adds a IValidatableObject.Validate method.

Adds an error message if the LineNum property is not in range

The IValidatableObject interface requires this method to be created.

Allows access to the current DbContext if necessary to get more information

Moves the Price check out of the business logic into this validation

Extra validation rule: an order for more than 100 books needs to phone in an order.

Returns the name of the property with the error to provide a better error message

I should point out that in the `IValidatableObject.Validate` method, you access a property outside the `LineNum` class: the `Title` of the `ChosenBook`. `ChosenBook` is a navigational property, and when the `DetectChanges` method is called, the *relational fixup* feature (see figure 1.10, stage 3) will ensure that the `ChosenBook` property isn't null. As a result, the validation code in listing 4.9 can access navigational properties that the business logic might not have.

> **NOTE** In addition to using the extensive list of built-in validation attributes, you can create your own validation attributes by inheriting the `ValidationAttribute` class on your own class. See http://mng.bz/9cec for more on the standard validation attributes that are available and for how to use the `ValidationAttribute` class.

After adding the validation rule code to your `LineItem` entity class, you need to add a validation stage to EF Core's `SaveChanges` method, called `SaveChangesWithValidation`. Although the obvious place to put this stage is inside the application's DbContext, you'll create an extension method instead. This method will allow `SaveChangesWithValidation` to be used on any `DbContext`, which means that you can copy this class and use it in your application.

The following listing shows this `SaveChangesWithValidation` extension method, and listing 4.11 shows the private method `ExecuteValidation` that `SaveChangesWithValidation` calls to handle the validation.

Listing 4.10 `SaveChangesWithValidation` added to the application's DbContext

```
public static ImmutableList<ValidationResult>
    SaveChangesWithValidation(this DbContext context)
{
    var result = context.ExecuteValidation();

    if (result.Any()) return result;

    context.SaveChanges();

    return result;
}
```

SaveChangesWithValidation returns a list of ValidationResults.

SaveChangesWithValidation is an extension method that takes the DbContext as its input.

The ExecuteValidation is used in SaveChangesWithChecking/SaveChangesWithCheckingAsync.

If there are errors, return them immediately and don't call SaveChanges.

There aren't any errors, so I am going to call SaveChanges.

Returns the empty set of errors to signify that there are no errors

Listing 4.11 `SaveChangesWithValidation` calls `ExecuteValidation` method

```
private static ImmutableList<ValidationResult>
    ExecuteValidation(this DbContext context)
{
    var result = new List<ValidationResult>();
    foreach (var entry in
        context.ChangeTracker.Entries()
            .Where(e =>
                (e.State == EntityState.Added) ||
                (e.State == EntityState.Modified)))
    {
```

Uses EF Core's ChangeTracker to get access to all the entity classes it is tracking

Filters the entities that will be added or updated in the database

```
            var entity = entry.Entity;
            var valProvider = new
                ValidationDbContextServiceProvider(context);
            var valContext = new
                ValidationContext(entity, valProvider, null);
            var entityErrors = new List<ValidationResult>();
            if (!Validator.TryValidateObject(
                entity, valContext, entityErrors, true))
            {
                result.AddRange(entityErrors);
            }
        }
        return result.ToImmutableList();
    }
```

Implements the IServiceProvider interface and passes the DbContext to the Validate method

The Validator.TryValidateObject is the method that validates each class.

Any errors are added to the list.

Returns the list of all the errors found (empty if there are no errors)

The main code is in the `ExecuteValidation` method, because you need to use it in sync and async versions of `SaveChangesWithValidation`. The call to `context.ChangeTracker.Entries` calls the DbContext's `DetectChanges` to ensure that all the changes you've made are found before the validation is run. Then the code looks at all the entities that have been added or modified (updated) and validates them all.

One piece of code I want to point out in listing 4.11 is a class called `ValidationDbContextServiceProvider`, which implements the `IServiceProvider` interface. This class is used when you create `ValidationContext`, so it is available in any entity classes that have the `IValidatableObject` interface, allowing the `Validate` method to access the current application's DbContext if necessary. Having access to the current DbContext allows you to create better error messages by obtaining extra information from the database.

You design the `SaveChangesWithValidation` method to return the errors rather than throw an exception. You do this to fit in with the business logic, which returns errors as a list, not an exception. You can create a new `BizRunner` variant, `RunnerWriteDbWithValidation`, that uses `SaveChangesWithValidation` instead of the normal `SaveChanges` and returns errors from the business logic or any validation errors found when writing to the database. The next listing shows the `BizRunner` class `RunnerWriteDbWithValidation`.

Listing 4.12 `BizRunner` variant `RunnerWriteDbWithValidation`

```
public class RunnerWriteDbWithValidation<TIn, TOut>
{
    private readonly IBizAction<TIn, TOut> _actionClass;
    private readonly EfCoreContext _context;

    public IImmutableList<ValidationResult>
        Errors { get; private set; }
    public bool HasErrors => Errors.Any();

    public RunnerWriteDbWithValidation(
        IBizAction<TIn, TOut> actionClass,
        EfCoreContext context)
```

This version needs its own Errors/HasErrors properties, as errors come from two sources.

Handles business logic that conforms to the IBizAction<TIn, TOut> interface

```
    {
        _context = context;
        _actionClass = actionClass;
    }

    public TOut RunAction(TIn dataIn)
    {
        var result = _actionClass.Action(dataIn);
        Errors = _actionClass.Errors;
        if (!HasErrors)
        {
            Errors =
                _context.SaveChangesWithValidation()
                    .ToImmutableList();
        }
        return result;
    }
}
```

This method is called to execute the business logic and handle any errors.

Runs the business logic I gave it

Any errors from the business logic are assigned to the local errors list.

If no errors, calls SaveChanges-WithChecking

Any validation errors are assigned to the Errors list.

Returns the result that the business logic returned

The nice thing about this new variant of the `BizRunner` pattern is that it has exactly the same interface as the original, nonvalidating `BizRunner`. You can substitute `RunnerWriteDbWithValidation<TIn, TOut>` for the original `BizRunner` without needing to change the business logic or the way that the calling method executes the `BizRunner`.

In section 4.7.2, you'll produce yet another variant of the `BizRunner` that can run multiple business logic classes in such a way that they look like a single business logic method. This is possible because of the business logic pattern described at the start of this chapter.

4.7.2 Using transactions to daisy-chain a sequence of business logic code

As I said earlier, business logic can get complex. When it comes to designing and implementing a large or complex piece of business logic, you have three options:

- *Option 1*—Write one big method that does everything.
- *Option 2*—Write a few smaller methods, with one overarching method to run them in sequence.
- *Option 3*—Write a few smaller methods, each of which updates the database, but combine them into one Unit Of Work (see sidebar in section 3.2.2).

Option 1 normally isn't a good idea because the method will be so hard to understand and refactor. It also has problems if parts of the business logic are used elsewhere, because you could break the DRY (don't repeat yourself) software principle.

Option 2 can work but can have problems if later stages rely on database items written by earlier stages, which could break the atomic unit rule mentioned in chapter 1: when there are multiple changes to the database, they all succeed, or they all fail.

This leaves option 3, which is possible because of a feature of EF Core (and most relational databases) called *transactions*. In section 3.2.2, the sidebar "Why you should call `SaveChanges` only once at the end of your changes" introduced the Unit Of Work and showed how `SaveChanges` saves all the changes inside a transaction to make sure that all the changes were saved or, if the database rejected any part of the change, that no changes were saved to the database.

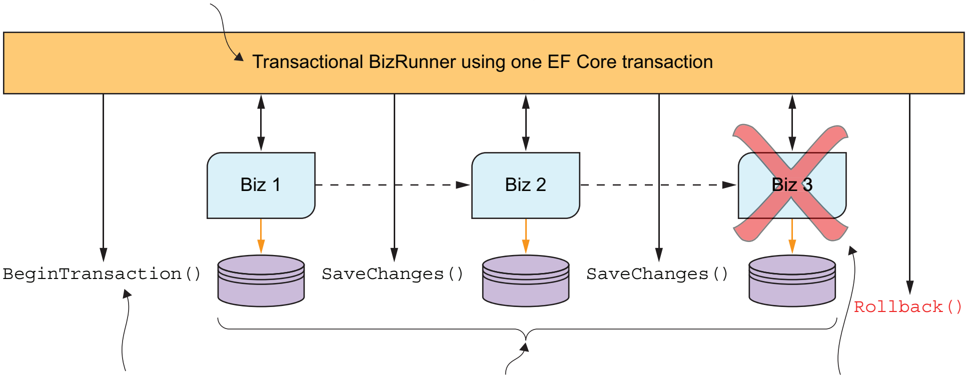
In this case, you want to spread the Unit Of Work over several smaller methods; let's call them `Biz1`, `Biz2`, and `Biz3`. You don't have to change `Biz` methods; they still think that they are working on their own and will expect `SaveChanges` to be called when each `Biz` method finishes. But when you create an overarching transaction, all three `Biz` methods, with their `SaveChanges` call, will work as one Unit Of Work. As a result, a database rejection/error in `Biz3` will reject any database changes made by `Biz1`, `Biz2`, and `Biz3`.

This database rejection works because when you use EF Core to create an explicit relational database transaction, it has two effects:

- Any writes to the database are hidden from other database users until you call the transaction's `Commit` method.
- If you decide that you don't want the database writes (say, because the business logic has an error), you can discard all database writes done in the transaction by calling the transaction `RollBack` command.

Figure 4.5 shows three separate pieces of business logic, each expecting a call to `SaveChanges` to update the database but being run by a class called the *transactional* `BizRunner`. After each piece of business logic has run, the `BizRunner` calls `SaveChanges`, which means that anything the business logic writes out is now available for subsequent

1. A special BizRunner runs each business logic class in turn. Each business logic stage uses an application DbContext that has an EF Core BeginTransaction applied to it.



2. BeginTransaction is called at the start. This marks the starting point of an explicit local transaction.

3. Each business logic runs as normal, with writes to the database. BizRunner then calls SaveChanges to save each stage's changes to the local transaction.

4. Biz 3 has an error, and RollBack is called. This removes all the database changes done within the transaction.

Figure 4.5 An example of executing three separate business logic stages under one transaction. When the last business logic stage returns an error, the other database changes applied by the first two business logic stages are rolled back.

business logic stages via the local transaction. In the final stage, the business logic, `Biz 3`, returns errors, which causes the `BizRunner` to call the `RollBack` command, which has the effect of removing any database writes done by `Biz 1` and `Biz 2`.

The next listing shows the code for the new transactional `BizRunner`, which starts a transaction on the application's DbContext before calling any of the business logic.

Listing 4.13 `RunnerTransact2WriteDb` running two business logic stages in series

```
public class RunnerTransact2WriteDb<TIn, TPass, TOut>
    where TOut : class
{
    private readonly IBizAction<TIn, TPass>
        _actionPart1;
    private readonly IBizAction<TPass, TOut>
        _actionPart2;
    private readonly EfCoreContext _context;

    public IImmutableList<ValidationResult>
        Errors { get; private set; }
    public bool HasErrors => Errors.Any();

    public RunnerTransact2WriteDb(
        EfCoreContext context,
        IBizAction<TIn, TPass> actionPart1,
        IBizAction<TPass, TOut> actionPart2)
    {
        _context = context;
        _actionPart1 = actionPart1;
        _actionPart2 = actionPart2;
    }

    public TOut RunAction(TIn dataIn)
    {
        using (var transaction =
            _context.Database.BeginTransaction())
        {
            var passResult = RunPart(
                _actionPart1, dataIn);
            if (HasErrors) return null;
            var result = RunPart(
                _actionPart2, passResult);

            if (!HasErrors)
            {
                transaction.Commit();
            }
            return result;
        }
    }
```

The three types are input, class passed from Part1 to Part2, and output.

The BizRunner can return null if there are errors, so it has to be a class.

Defines the generic BizAction for the two business logic parts

Holds any error information returned by the business logic

The constructor takes both business classes and the application DbContext.

Starts the transaction within a using statement

If there are errors, returns null. (The rollback is handled by the dispose.)

The private method, RunPart, runs the first business part.

If the first part of the business logic was successful, runs the second business logic

If there are no errors, commits the transaction to the database

Returns the result from the last business logic

If commit is not called before the using end, RollBack undoes all the changes.

```
    private TPartOut RunPart<TPartIn, TPartOut>(
        IBizAction<TPartIn, TPartOut> bizPart,
        TPartIn dataIn)
        where TPartOut : class
    {
        var result = bizPart.Action(dataIn);
        Errors = bizPart.Errors;
        if (!HasErrors)
        {
            _context.SaveChanges();
        }
        return result;
    }
}
```

This private method handles running each part of the business logic.

Runs the business logic and copies the business logic's Errors

If the business logic was successful, calls SaveChanges

Returns the result from the business logic it ran

In your `RunnerTransact2WriteDb` class, you execute each part of the business logic in turn, and at the end of each execution, you do one of the following:

- *No errors*—You call `SaveChanges` to save to the transaction any changes that business logic has run. That save is within a local transaction, so other methods accessing the database won't see those changes yet. Then you call the next part of the business logic, if there is one.
- *Has errors*—You copy the errors found by the business logic that just finished to the `BizRunner` error list and exit the `BizRunner`. At that point, the code steps outside the `using` clause that holds the transaction, which causes disposal of the transaction. Because no transaction `Commit` has been called, the disposal will cause the transaction to execute its `RollBack` method, which discards the database writes to the transaction. Those writes are never written to the database.

If you've run all the business logic with no errors, you call the `Commit` command on the transaction. This command does an atomic update of the database to reflect all the changes to the database that are contained in the local transaction.

4.7.3 Using the RunnerTransact2WriteDb class

To test the `RunnerTransact2WriteDb` class, you'll split the order-processing code you used earlier into two parts:

- `PlaceOrderPart1`—Creates the `Order` entity, with no `LineItems`
- `PlaceOrderPart2`—Adds the `LineItems` for each book bought to the `Order` entity that was created by the `PlaceOrderPart1` class

`PlaceOrderPart1` and `PlaceOrderPart2` are based on the `PlaceOrderAction` code you've already seen, so I don't repeat the business code here.

Listing 4.14 shows you the code changes that are required for `PlaceOrderService` (shown in listing 4.6) to change over to use the `RunnerTransact2WriteDb` `BizRunner`. The listing focuses on the part that creates and runs the two stages, `Part1` and `Part2`, with the unchanged parts of the code left out so you can see the changes easily.

Listing 4.14 The `PlaceOrderServiceTransact` class showing the changed parts

```
public class PlaceOrderServiceTransact
{
    //… code removed as the same as in listing 4.5

    public PlaceOrderServiceTransact(
        IRequestCookieCollection cookiesIn,
        IResponseCookies cookiesOut,
        EfCoreContext context)
    {
        _checkoutCookie = new CheckoutCookie(
            cookiesIn, cookiesOut);
        _runner = new RunnerTransact2WriteDb
            <PlaceOrderInDto, Part1ToPart2Dto, Order>(
            context,
            new PlaceOrderPart1(
                new PlaceOrderDbAccess(context)),
            new PlaceOrderPart2(
                new PlaceOrderDbAccess(context)));
    }

    public int PlaceOrder(bool tsAndCsAccepted)
    {
        //… code removed as the same as in listing 4.6
    }
}
```

This version of PlaceOrderService uses transactions to execute two business logic classes: PlaceOrderPart1 and PlaceOrderPart2.

This BizRunner handles multiple business logic inside a transaction.

The BizRunner needs the input, the class passed from Part1 to Part2, and the output.

The BizRunner needs the application's DbContext.

Provides an instance of the first part of the business logic

Provides an instance of the second part of the business logic

The important thing to note is that the business logic has no idea whether it's running in a transaction. You can use a piece of business logic on its own or as part of a transaction. Similarly, listing 4.14 shows that only the caller of transaction-based business logic, which I call the `BizRunner`, needs to change. Using a transaction makes it easy to combine multiple business logic classes under one transaction without needing to change any of your business logic code.

The advantage of using transactions like this one is that you can split and/or reuse parts of your business logic while making these multiple business logic calls look to your application, especially its database, like one call. I've used this approach when I needed to create and then immediately update a complex, multipart entity. Because I needed the Update business logic for other cases, I used a transaction to call the Create business logic followed by the Update business logic, which saved me development effort and kept my code DRY.

The disadvantage of this approach is that it adds complexity to the database access, which can make debugging a little more difficult, or the use of database transactions could cause a performance issue. Also, be aware that if you use the `EnableRetryOnFailure` option (see section 11.8) to retry database accessed on errors, you need to handle possible multiple calls to your business logic.

Summary

- The term *business logic* describes code written to implement real-world business rules. The business logic code can range from the simple to the complex.
- Depending on the complexity of your business logic, you need to choose an approach that balances how easy it is to solve the business problem against the time it takes you to develop and test your solution.
- Isolating the database access part of your business logic into another class/project can make the pure business logic simpler to write but take longer to develop.
- Putting all the business logic for a feature in one class is quick and easy but can make the code harder to understand and test.
- Creating a standardized interface for your business logic makes calling and running the business logic much simpler for the frontend.
- Sometimes, it's easier to move some of the validation logic into the entity classes and run the checks when that data is being written to the database.
- For business logic that's complex or being reused, it might be simpler to use a database transaction to allow a sequence of business logic parts to be run in sequence but, from the database point of view, look like one atomic unit.

For readers who are familiar with EF6.x:

- Unlike EF6.x, EF Core's `SaveChanges` method doesn't validate data before it's written to the database. But it's easy to implement a method that provides this feature in EF Core.

5

Using EF Core in ASP.NET Core web applications

This chapter covers

- Using EF Core in ASP.NET Core
- Using dependency injection in ASP.NET Core
- Accessing the database in ASP.NET Core MVC actions
- Using EF Core migrations to update a database
- Using async/await to improve scalability

In this chapter, you'll pull everything together by using ASP.NET Core to build a real web application. Using ASP.NET Core brings in issues that are outside EF Core, such as dependency injection (covered in section 5.4) and async/await (covered in section 5.10). But they're necessary if you're going to use EF Core in this type of application.

This chapter assumes that you've read chapters 2–4 and know about querying and updating the database and what business logic is. This chapter is about where to place your database access code and how to call it in a real application. It also covers the specific issues of using EF Core in an ASP.NET Core (including Blazor Server) applications. For that reason, this chapter includes quite a bit about ASP.NET Core, but it's all focused on using EF Core well in this type of application.

I end with more general information on various ways to obtain an instance of the application's DbContext for cases such as background tasks.

5.1 Introducing ASP.NET Core

The ASP.NET Core website states that "ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, Internet-connected applications" (http://mng.bz/QmOw). This summary is a good one, but ASP.NET Core has so many great features that it's hard to pick which ones to comment on.

> **NOTE** I recommend Andrew Lock's book *ASP.NET Core in Action* (Manning, 2020) for a detailed description of ASP.NET Core's many features.

I've been using ASP.NET MVC5, the precursor of ASP.NET Core, for years. I thought it was a good framework, if a bit slow in performance. But for me, ASP.NET Core blows ASP.NET MVC5 out of the water, with a phenomenal improvement in performance and new ways to show data, such as Razor Pages and Blazor.

> **TIP** When I first tried ASP.NET Core, I was disappointed by its performance; it turns out that the default logging slows things down in development mode. When I replaced the normal loggers with my quicker, in-memory logging, the Book App page that was listing the book was three times faster! So watch out for too much logging slowing your application.

In this book, you will build the Book App, which is a web application, using ASP.NET Core to show how EF Core works with a real application. ASP.NET Core can be used in a number of ways, but for the Book App examples. we will use ASP.NET Core's Model-View-Controller (MVC) pattern.

5.2 Understanding the architecture of the Book App

Chapter 2 presented a diagram of the Book App, and chapter 4 extended it with two more projects to handle the business logic. Figure 5.1 shows you the combined architecture after chapter 4, with all the projects in the application. As you go through this chapter, you'll learn how and why we split the database access code across the various projects. One reason is to make your web application easier to write, refactor, and test.

This layered architecture, which creates a single executable containing all the code, works well with many cloud providers that can spin up more instances of the web application if it's under a heavy load; your host will run multiple copies of a web application and place a load balancer to spread the load over all the copies. This process is known as *scaling out* in Microsoft Azure and *auto scaling* in Amazon Web Services (AWS).

> **NOTE** In part 3, I update the architecture of the Book App to use the modular monolith, Domain-Driven Design, and clean architecture. See the useful Microsoft document about layered and clean architectures at http://mng.bz/5jD1.

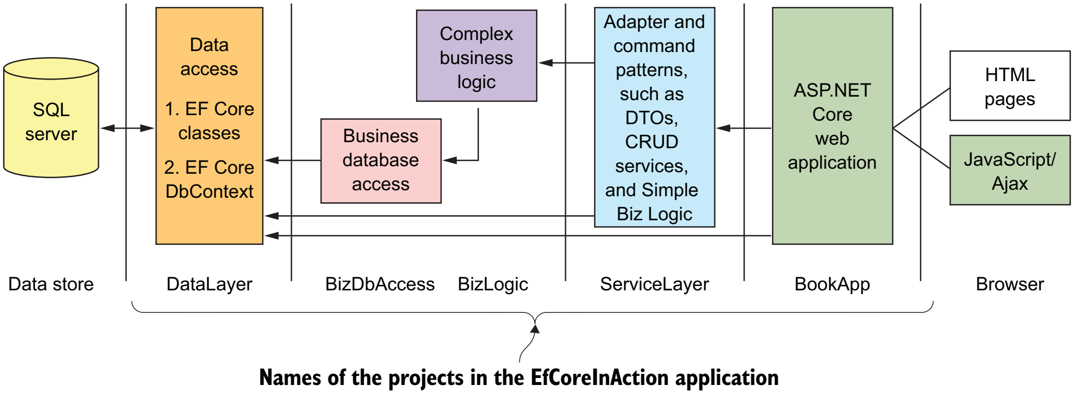


Figure 5.1 All the projects in the Book App. The arrows show the main routes by which EF Core data moves up and down the layers.

5.3 Understanding dependency injection

ASP.NET Core uses *dependency injection* (DI) extensively, as does .NET in general. You need to understand DI because it's the method used in ASP.NET Core to get an instance of the application's DbContext.

> **DEFINITION** *Dependency injection* is a way to link together your application dynamically. Normally, you'd write var `myClass = new MyClass()` to create a new instance of `MyClass`. That code works, but you've hardcoded the creation of that class, and you can change it only by changing your code. With DI, you can *register* your `MyClass` with a DI provider, using, say, an interface such as `IMyClass`. Then, when you need the class, you use `IMyClass myClass`, and the DI provider will dynamically create an instance and *inject* it into the `IMyClass myClass` parameter/property.

Using DI has lots of benefits, and here are the main ones:

- DI allows your application to link itself dynamically. The DI provider will work out what classes you need and create them in the right order. If one of your classes needs the application's DbContext, for example, the DI can provide it.
- Using interfaces and DI together means that your application is more loosely coupled; you can replace a class with another class that matches the same interface. This technique is especially useful in unit testing: you can provide a replacement version of the service with another, simpler class that implements the interface (called *stubbing* or *mocking* in unit tests).
- Other, more advanced features exist, such as using DI to select which class to return based on certain settings. If you're building an e-commerce application, in development mode, you might want to use a dummy credit card handler instead of the normal credit card system.

I use DI a lot and wouldn't build any real application without it, but I admit that it can be confusing the first time you see it.

> **NOTE** This section gives you a quick introduction to DI so that you understand how to use DI with EF Core. If you want more information on DI in ASP.NET Core, see Microsoft's documentation at http://mng.bz/Kv16. For an overall view of DI, consider the book *Dependency Injection Principles, Practices, and Patterns*, by Steven Van Deursen and Mark Seemann (Manning, 2019), which has a whole chapter on NET Core DI (http://mng.bz/XdjG).

5.3.1 Why you need to learn about DI in ASP.NET Core

Chapter 2 showed you how to create an instance of the application's DbContext by using the following snippet of code:

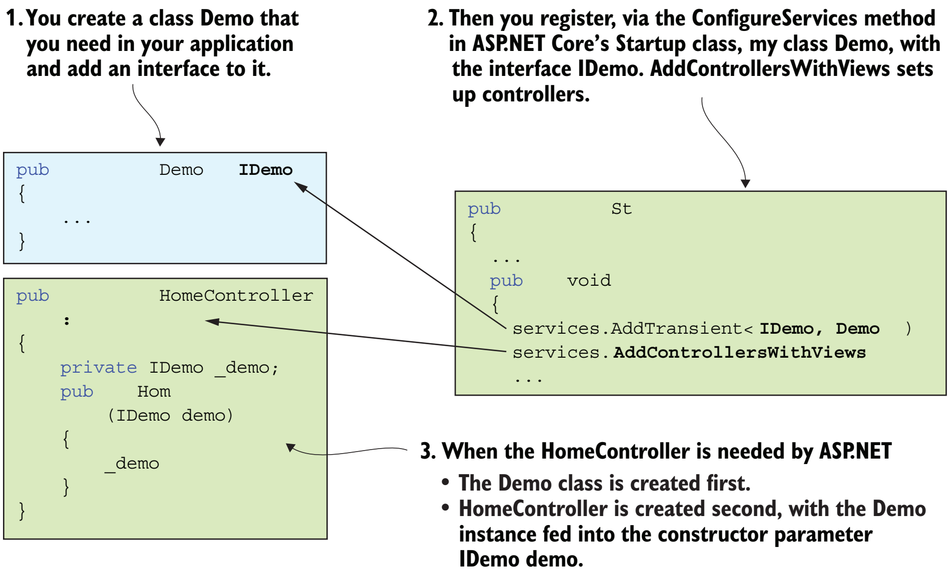
```
const string connection =
     "Data Source=(localdb)\\mssqllocaldb;" +
     "Database=EfCoreInActionDb.Chapter02;" +
     "Integrated Security=True;";
var optionsBuilder =
    new DbContextOptionsBuilder
           <EfCoreContext>();

optionsBuilder.UseSqlServer(connection);
var options = optionsBuilder.Options;

using (var context = new EfCoreContext(options))
{…
```

That code works but has a few problems. First, you're going to have to repeat this code for each database access you make. Second, this code uses a fixed database access string, referred to as a *connection string*, which isn't going to work when you want to deploy your site to a host, because the database location for the hosted database will be different from the database you use for development.

You can work around these two problems in several ways, such as by overriding the `OnConfiguration` method in the application's DbContext (covered in section 5.11.1). But DI is a better way to handle this situation and is what ASP.NET Core uses. Using a slightly different set of commands, you can tell the DI provider how to create your application's DbContext—a process called *registering a service*—and then ask the DI for an instance of your application's DbContext anywhere in ASP.NET Core's system that supports DI.

5.3.2 A basic example of dependency injection in ASP.NET Core

Setting up the code to configure the application's DbContext is a little complicated and can hide the DI part. My first example of DI in ASP.NET Core, shown in figure 5.2, uses a simple class called `Demo`, which you'll use in an ASP.NET controller. This example will be useful in section 5.7, when I show you how to use DI to make your code simpler to call.



1. You create a class Demo that you need in your application and add an interface to it.

2. Then you register, via the ConfigureServices method in ASP.NET Core's Startup class, my class Demo, with the interface IDemo. AddControllersWithViews sets up controllers.

```
pub             Demo    IDemo
{
    ...
}
```

```
pub             HomeController
    :
{
    private IDemo _demo;
    pub     Hom
        (IDemo demo)
    {
        _demo
    }
}
```

```
pub             St
{
  ...
  pub    void
  {
    services.AddTransient< IDemo, Demo   )
    services.AddControllersWithViews
    ...
```

3. When the HomeController is needed by ASP.NET
   - The Demo class is created first.
   - HomeController is created second, with the Demo instance fed into the constructor parameter IDemo demo.

Figure 5.2 An example of a class called `Demo` being inserted via DI into a controller's constructor. The code on the right registers your `IDemo/Demo` pair, and the `AddControllersWithViews` command registers all the ASP.NET Core controllers. When ASP.NET Core needs the `HomeController` (used for showing HTML pages), DI will create the `HomeController`. Because the `HomeController` needs an `IDemo` instance, DI will create one and inject it into the `HomeController`'s constructor.

Figure 5.2 shows that by registering the `IDemo/Demo` pair with ASP.NET Core's DI, you can access it in your `HomeController` class. Classes that are registered are referred to as *services*.

The rule is that any DI service can be referenced, or *injected*, in any other DI service. In figure 5.2, you register your `IDemo/Demo` class and call the `AddControllersWithViews` configuration method to register the ASP.NET Core's controller classes—specifically, in this example, the `HomeController` class. This allows you to use the `IDemo` interface in the `HomeController's` constructor, and the DI provides an instance on the `Demo` class. In DI terms, you use *constructor injection* to create an instance of the class that you've registered. You'll use DI in various ways in this chapter, but the rules and terms defined here will help you make sense of these later examples.

5.3.3 The lifetime of a service created by DI

One feature of DI that's important when talking about EF Core is the *lifetime* of an instance created by DI—how long the instance exists before being lost or disposed of. In our `IDemo/Demo` example, you registered the instance as *transient*; every time you ask for an instance of `Demo`, it creates a new one. If you want to use your own classes with DI, you most likely declare a *transient* lifetime; that's what I use for all my services, as it means that each instance starts with its default setup. For simple, valuelike classes, such as data setup at startup, you may declare them as *singleton* (you get the same instance every time).

The application's DbContext is different. It has its lifetime set to *scoped*, which means that however many instances of the application's DbContext you ask for during one HTTP request, you get the same instance. But when that HTTP request ends, that instance is gone (technically, because DbContext implements `IDisposable`, it's disposed of), and you get a new, scoped instance in the next HTTP request. Figure 5.3 shows the three sorts of lifetimes, with a new letter for each new instance.

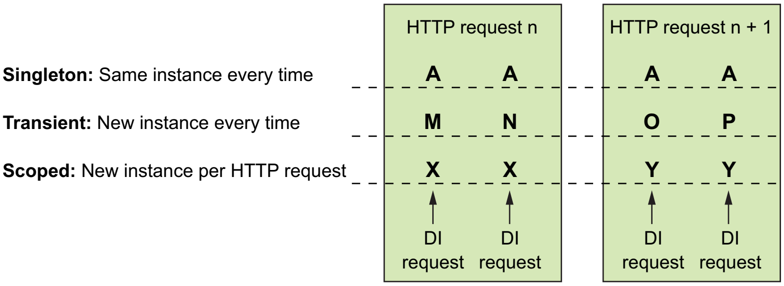


Figure 5.3 Instances produced by DI have three types of lifetimes: singleton, transient, and scoped. This figure shows those three types with four injections for each, two per HTTP request. The letters represent each instance. If a letter is used multiple times, all those injections are the same instance of the class.

You need to use a scoped lifetime for the application's DbContext in case you inject the application's DbContext into multiple classes. Sometimes, for example, it's good to break a complex update into multiple classes. If you do, you need the application's DbContext to be the same in all the classes; otherwise, changes made in one class would not appear in another class.

Let's break a complex update into a `Main` class and a `SubPart` class, where the `Main` class obtains an instance of the `SubPart` via an `ISubPart` interface in its constructor. Now the `Main` part calls a method in the `ISubPart` interface, and the `SubPart` code loads an entity class and changes a property. At the end of the whole update, the `Main` code calls `SaveChanges`. If the two applications' DbContext injected into `Main` and `SubPart` classes are different, the change that the `SubPart` class made is lost.

This situation may sound obscure or unusual, but in even medium-size applications, it can happen a lot. I often break complex code into separate classes, either because the whole code is so big or because I want to unit-test different parts of the code separately.

Conversely, each HTTP request must have its own instance of the application's DbContext, because EF Core's DbContext isn't *thread-safe* (see section 5.11.1). This fact is why the application's DbContext has a scoped lifetime for each HTTP request and is one of the reasons why DI is so useful.

5.3.4 Special considerations for Blazor Server applications

If you are using a Blazor frontend talking to a ASP.NET Core backend, known as a *Blazor Server hosting model*, you need to change your approach to registering and/or obtaining an instance of your application's DbContext. The problem is that with a Blazor frontend, you can send calls for a database access in parallel, which means that multiple threads will try to use one instance of your application's DbContext, which isn't allowed.

You have a few ways to get around this problem, but the simplest is to create a new instance of your application's DbContext for every database access. EF Core 5 has provided a DbContext factory method that creates a new instance every time you call it (see section 5.4.3). The DbContext factory method prevents multiple threads from trying to use the same instance of your application's DbContext.

The downside of using the DbContext factory method is that different classes registered to DI won't use the same DbContext instance. The scoped lifetime DbContext instance example in section 5.3.3, for example, would cause problems because the `Main` class and a `SubPart` class would have different instances of your application's DbContext. One solution to this problem is to have the `Main` class obtain an instance of the application's DbContext and pass that instance to the `SubPart` class, either by creating the `SubPart` itself or via a method parameter.

Even the DbContext factory approach can have problems with services that are long-lived. The EF Core team has written guidance on using EF Core with a Blazor Server application with an example application that shows some of the techniques; see http://mng.bz/yY7G.

5.4 Making the application's DbContext available via DI

Now that you understand DI, you're ready to set up your application's DbContext as a service so that you can access it later via DI. You do this at the startup of the ASP.NET Core web application by registering the application's DbContext with the DI provider, using information that tells EF Core what sort of database you're accessing and where it's located.

5.4.1 Providing information on the database's location

When developing your application, you'll want to run it on your development machine and access a local database for testing. The type of the database will be defined by the business need, but the location of the database on your development machine is up to you and whatever database server you're using.

For web applications, the location of the database normally isn't hardcoded into the application because it'll change when the web application is moved to its host, where real users can access it. Therefore, the location and various database configuration settings are typically stored as a *connection string*. This string is stored in an application setting file that ASP.NET reads when it starts. ASP.NET Core has a range of application setting files, but for now, you'll concentrate on the three standard ones:

- *appsetting.json*—Holds the settings that are common to development and production
- *appsettings.Development.json*—Holds the settings for the development build
- *appsettings.Production.json*—Holds the settings for the production build (when the web application is deployed to a host for users to access it)

> **NOTE** There's a lot more to application setting files in ASP.NET Core that we haven't covered. Please look at the APS.NET Core documentation for a more complete description.

Typically, the development connection string is stored in the appsettings.Development .json file. Listing 5.1 shows a connection string suitable for running an SQL database locally on a Windows PC.

> **NOTE** The Visual Studio installation includes a feature called *SQL Server Express*, which allows you to use SQL Server for development.

Listing 5.1 `appsettings.Development.json` file with database connection string

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=EfCoreInActionDb
 ;Trusted_Connection=True"
  },
  … other parts removed as not relevant to database access
}
```

You need to edit your appsettings.Development.json file to add the connection string for your local, development database. This file may or may not have a `ConnectionStrings` section, depending on whether you set Authentication to Individual User Accounts. (The Individual User Accounts option needs its own database, so Visual Studio adds a connection string for the authorization database to the appsetting.json file.) You can call your connection string anything you like; this example uses the name `DefaultConnection` in our application.

5.4.2 Registering your application's DbContext with the DI provider

The next step is registering your application's DbContext with the DI provider at startup. Any configuration to be done when ASP.NET Core starts up is done in the aptly named `Startup` class. This class is executed when the ASP.NET Core application starts and contains several methods to set up/configure the web application.

The application's DbContext for ASP.NET Core has a constructor that takes a `DbContextOptions<T>` parameter defining the database options. That way, the database connection string can change when you deploy your web application (see section 5.8). As a reminder, here's what the Book App's DbContext constructor looks like, shown in bold in this code snippet:

```
public class EfCoreContext : DbContext
{
    //… properties removed for clarity

    public EfCoreContext(
        DbContextOptions<EfCoreContext> options)
        : base(options) {}

    //… other code removed for clarity
}
```

The following listing shows how the application's DbContext is registered as a service in an ASP.NET Core application. This registration is done in the `ConfigureServices` method in the `Startup` class of your ASP.NET Core application, along with all the DI services you need to register.

Listing 5.2 Registering your DbContext in ASP.NET Core's `Startup` class

Sets up a series of services to use with controllers and Views

This method in the Startup class sets up services.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    var connection = Configuration
        .GetConnectionString("DefaultConnection");

    services.AddDbContext<EfCoreContext>(
        options => options.UseSqlServer(connection));

    //… other service registrations removed
}
```

You get the connection string from the appsettings.json file, which can be changed when you deploy.

Configures the application's DbContext to use SQL Server and provide the connection

Your first step is getting the connection string from the application's `Configuration` class. In ASP.NET Core, the `Configuration` class is set up during the `Startup` class constructor, which reads the appsetting files. Getting the connection string that way allows you to change the database connection string when you deploy the code to a host. Section 5.8.1, which is about deploying an ASP.NET Core application that uses a database, covers how this process works.

The second step—making the application's DbContext available via DI—is done by the `AddDbContext` method, which registers the application's DbContext, `EfCoreContext`, and the `DbContextOptions<EfCoreContext>` instances as services. When you use the type `EfCoreContext` in places where DI intercepts, the DI provider will create an instance of the application's DbContext, using the `DbContextOptions<EfCoreContext>` options. Or if you ask for multiple instances in the same HTTP request, the DI provider will return the same instances. You'll see this process in action when you start using the application's DbContext to do database queries and updates in section 5.6.

5.4.3 Registering a DbContext Factory with the DI provider

As stated in section 5.3.4, Blazor Server applications need careful managing of the instances of your application's DbContext, as do some other application types. In EF Core 5, the `IDbContextFactory<TContext>` interface was added along with a method to register the DbContext factory, as shown in the following listing.

Listing 5.3 Registering a DbContext factory in ASP.NET Core's `Startup` class

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    var connection = Configuration
        .GetConnectionString("DefaultConnection");

    services.AddDbContextFactory<EfCoreContext>(
        options => options.UseSqlServer(connection));

    //… other service registrations removed
}
```

This method in the Startup class sets up services.

Sets up a series of services to use with controllers and Views

You get the connection string from the appsettings.json file, which can be changed when you deploy.

Configures the DbContext factory to use SQL Server and provide the connection

Typically, you use the `AddDbContextFactory` method only with Blazor in the frontend or in applications where you cannot control the parallel access to the same application's DbContext, which breaks the thread-safe rule (see section 5.11.1). Many other applications, such as ASP.NET Core, manage parallel accesses for you, so you can obtain an instance of the application's DbContext via DI.

5.5 Calling your database access code from ASP.NET Core

Having configured the application DbContext and registered it as a DI service, you're ready to access the database. In these examples, you're going to run a query to display the books and run commands that update the database. You'll focus on how to execute these methods from ASP.NET Core; I assume that you've already grasped how to query and update the database from previous chapters.

> **NOTE** The example code is mainly about using ASP.NET Core MVC, but all the examples of using DI also apply to all forms of ASP.NET Core: Razor Pages, MVC, and Web API. A few sections also cover the Blazor Server applications, because the handling of obtaining an instance of the application's DbContext by DI is different.

5.5.1 A summary of how ASP.NET Core MVC works and the terms it uses

First, here's a quick summary of how to use ASP.NET Core to implement our Book App. To display the various HTML pages, you'll use an ASP.NET Core *controller*, which is the class that handles delivering HTML pages via Razor `Views`. To do this, you'll create a class called `HomeController`, which inherits from ASP.NET Core's `Controller` class. This controller has several Razor `Views` linked to its methods, which in ASP.NET Core are known as *action methods*.

Our Book App's `HomeController` has an action method called `Index`, which shows the book list, and one called `About`, which provides a summary page for the site. You have other controllers to handle checkout, existing orders, admin actions, and so on. Although you could put all your database access code inside each action method of each controller, I rarely do because I use a software design principle called Separation of Concerns (SoC), which the next subsection explains.

5.5.2 Where does the EF Core code live in the Book App?

As you learned in section 5.2, our Book App is built using a layered architecture, which is meant to represent an architecture that could be used in a real-world application. In this section, you'll see where to place the various pieces of EF Core's database access code and why.

> **DEFINITION** *Separation of Concerns* is the idea that a software system must be decomposed into parts that overlap in functionality as little as possible. It's linked to two other principles: coupling and cohesion. With *coupling*, you want each project in your application to be as self-contained as possible, and with *cohesion*, each project in your application should have code that provides similar or strongly related functions. See http://mng.bz/wHJS for more information.

Figure 5.4 maps where the database access code is located in your application, using the earlier architecture diagram (figure 5.1). The bubbles show what type of database code you'll find in each layer. Notice that the ASP.NET Core project and the pure business logic (BizLogic) project have no EF Core query/update code in them.

Applying SoC principles has benefits throughout the application. You learned about the reason for splitting out the complex business logic in chapter 4. But in this chapter, you'll see the benefits for the ASP.NET Core project:

- The ASP.NET Core frontend is all about displaying data, and doing that well is a big task that needs lots of concentration. Therefore, you'll use the service layer to handle both the EF Core commands and the transformation of the database data into a form that the ASP.NET Core frontend can easily use—often via DTOs, also known as ViewModels in ASP.NET Core. Then you can concentrate on making the best user experience rather than think about whether you have the database query right.

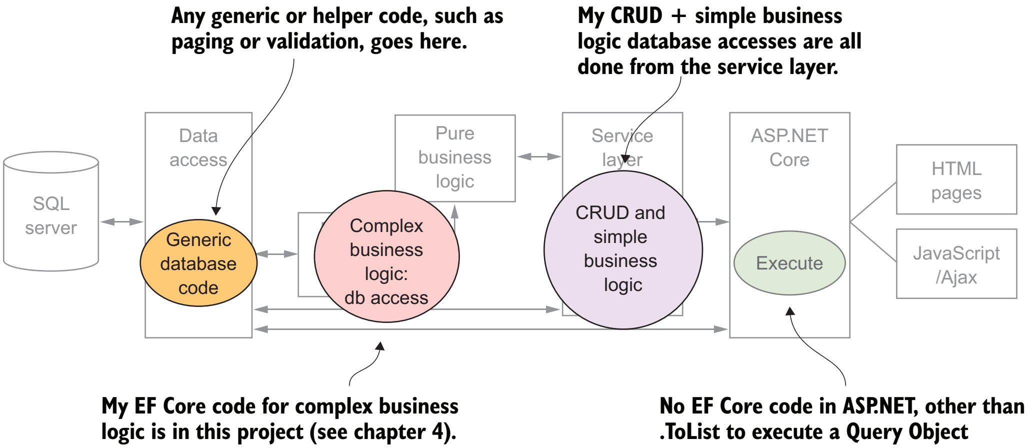


Figure 5.4 **Locations of the database access code (the EF Core code) in the Book App. Separating the EF Core code in this way makes it easier to find, understand, refactor, and test.**

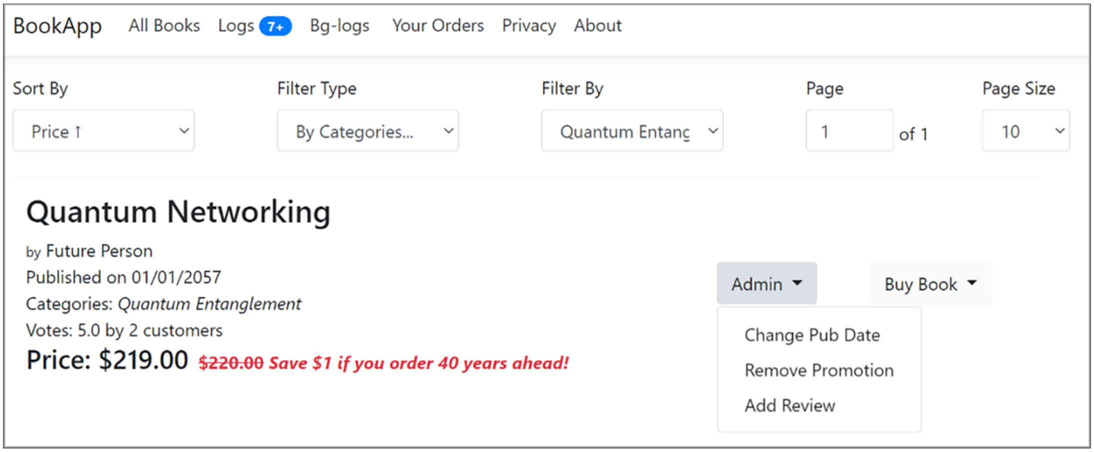
- ASP.NET controllers often have multiple pages/actions (say, one to list items, one to add a new item, one to edit an item, and so on), each of which would need its own database code. By moving the database code out to the service layer, you can create individual classes for each database access rather than have the code spread throughout a controller.
- It's much easier to unit-test your database code if it's in the service layer rather than when it's in an ASP.NET Core controller. You can test ASP.NET Core controllers, but testing can get complicated if your code accesses properties such as `HtppRequest` (which it does), because it's hard to replicate some of these features to get your unit test to work.

NOTE You can run tests against your full ASP.NET Core application by using the `Microsoft.AspNetCore.Mvc.Testing` NuGet package. This testing is known as integration testing when you are testing the whole application, whereas unit testing focuses on testing small parts of the application. You can find more about integration testing at http://mng.bz/MXa7.

5.6 Implementing the book list query page

Now that I've set the scene, you're going to implement the ASP.NET Core part of the list of books in our Book App. To remind you what the site looks like, figure 5.5 shows a screenshot of the Book App, with the list of books and the local admin update features.

In chapter 2, you wrote a class called `ListBooksService` that handled the complexities of transforming, sorting, filtering, and paging the books to display. You'll want to use this class in an ASP.NET Core action called `Index` in the controller



Sort By: Price ↑
Filter Type: By Categories...
Filter By: Quantum Entang
Page: 1 of 1
Page Size: 10

Quantum Networking

by Future Person
Published on 01/01/2057
Categories: *Quantum Entanglement*
Votes: 5.0 by 2 customers
Price: $219.00 ~~$220.00~~ *Save $1 if you order 40 years ahead!*

Figure 5.5 The home page of the Book App, showing the list of books and the admin features, including the Change Pub(lication) Date of a book

`HomeController`. The main issue is that to create an instance of the `ListBooksService` class, you need an instance of the application's DbContext.

5.6.1 *Injecting an instance of the application's DbContext via DI*

The standard way of providing an instance of the application's DbContext to an ASP.NET Core application (and other types of hosted applications) is via DI injection via a class's constructor (see section 5.3.2). For an ASP.NET Core application, you might add a constructor in the controller that has the application's DbContext class as a parameter (dependency injection by constructor).

Listing 5.4 shows the start of the ASP.NET Core `HomeController`, where you've added a constructor and copied the injected `EfCoreContext` class to a local field that can be used to create an instance of the `BookListService` class that you need to list the books. This code uses the DI approach from section 5.3.2 and figure 5.2 but replaces the `Demo` class with the application's DbContext class, `EfCoreContext`.

Listing 5.4 The `Index` action in the `HomeController` displays the list of books

```
public class HomeController : Controller
{
    private readonly EfCoreContext _context;

    public HomeController(EfCoreContext context)
    {
        _context = context;
    }

    public IActionResult Index
        (SortFilterPageOptions options)
```

The application's DbContext is provided by ASP.NET Core via DI.

ASP.NET action, called when the home page is called up by the user

The options parameter is filled with sort, filter, and page options via the URL.

```
    {
        var listService =
            new ListBooksService(_context);

        var bookList = listService
            .SortFilterPage(options)
            .ToList();

        return View(new BookListCombinedDto
            (options, bookList));
    }
```

ListBooksService is created by using the application's DbContext from the private field _context.

The SortFilterPage method is called with the sort, filter, and page options provided.

Sends the options (to fill in the controls at the top of the page) and the list of BookListDtos to display as an HTML table

The ToList() method executes the LINQ commands, causing EF Core to translate the LINQ into the appropriate SQL to access the database and return the result as a list.

After you've used the local copy of the application's DbContext to create your `ListBooksService`, you can call its `SortFilterPage` method. This method takes the parameters returned from the various controls on the list page and returns an `IQueryable<BookListDto>` result. Then you add the `ToList` method to the end of the result, which causes EF Core to execute that `IQueryable` result against the database and return the list of book information the user has asked for. This result is given to an ASP.NET Core view to display.

You could've had the `SortFilterPage` method return a `List<BookListDto>` result, but that approach would've limited you to using a synchronous database access. As you'll see in section 5.10 on async/await, by returning an `IQueryable<BookListDto>` result, you can choose to use a normal (synchronous) or an async version of the final command that executes the query.

5.6.2 *Using the DbContext Factory to create an instance of a DbContext*

In some applications, such as a Blazor Server app (see section 5.3.4), the normal scoping of your application's DbContext doesn't work. In this case, you can inject EF Core's `IDbContextFactory<TContext>` by using DI. This decoupling is useful for Blazor applications, in which EF Core recommends using the `IDbContextFactory`, and may be useful in other scenarios.

Here is an example taken from the BlazorServerEFCoreSample provided by the EF Core team. In this example, the `DbContext Factory` is injected into a Blazor Razor page, as shown in the following listing. Only the use of the `DbContext Factory` and the creation of the DbContext have comments.

Listing 5.5 Example of injecting the `DbContext Factory` into a Razor page

```
@page "/add"

@inject IDbContextFactory<ContactContext> DbFactory
@inject NavigationManager Nav
@inject IPageHelper PageHelper
```

The DbContext Factory is injected into the Razor page.

```
@if (Contact != null)
{
    <ContactForm Busy="@Busy"
                 Contact="@Contact"
                 IsAdd="true"
                 CancelRequest="Cancel"
                 ValidationResult=
"@(async (success) => await ValidationResultAsync(success))" />
}
@if (Success)
{
    <br />
    <div class="alert alert-success">The contact was successfully
     added.</div>
}
@if (Error)
{
    <br />
    <div class="alert alert-danger">Failed to update the contact
     (@ErrorMessage).</div>
}

@code {
    //… various fields left out
    private async Task ValidationResultAsync(bool success)
    {
        if (Busy)
            return;

        if (!success)
        {
            Success = false;
            Error = false;
            return;
        }

        Busy = true;

        using var context = DbFactory.CreateDbContext();
        context.Contacts.Add(Contact);

        try
        {
            await context.SaveChangesAsync();
            Success = true;
            Error = false;
            // ready for the next
            Contact = new Contact();
            Busy = false;
        }
        catch (Exception ex)
        {
            Success = false;
            Error = true;
            ErrorMessage = ex.Message;
```

Another technique to handle Blazor Server apps. It won't handle extra requests until the first request has finished.

Creates a new instance of the application's DbContext. Note the use of var for disposing.

The new Contact information is added to the DbContext.

Saves the Contact to the database

```
            Busy = false;
        }
    }

    private void Cancel()
    {
        Nav.NavigateTo($"/{PageHelper.Page}");
    }
}
```

Note that the DbContext instances that created the `DbContext Factory` are not managed by the application's service provider and therefore must be disposed by the application. In the Blazor Razor page shown in listing 5.5, the `using var context =` … will dispose the DbContext instance when the scope of the local context variable is exited.

NOTE You can find the Razor page shown in listing 5.5 at http://mng.bz/aorz.

5.7 Implementing your database methods as a DI service

Although the constructor injection approach you used in the preceding section works, there's another way to use DI that provides better isolation of the database access code: *parameter injection*. In ASP.NET Core, you can arrange for a service to be injected into an *action* method via a parameter marked with the attribute `[FromServices]`. You can provide a specific service that each action method in your controller needs; this approach is both more efficient and simpler to unit-test. To see how it works, you're going to use a class called `ChangePubDateService` that's in your service layer to update the publication date of a book. This class allows the admin user to change the publication date of a book, as shown in figure 5.6.

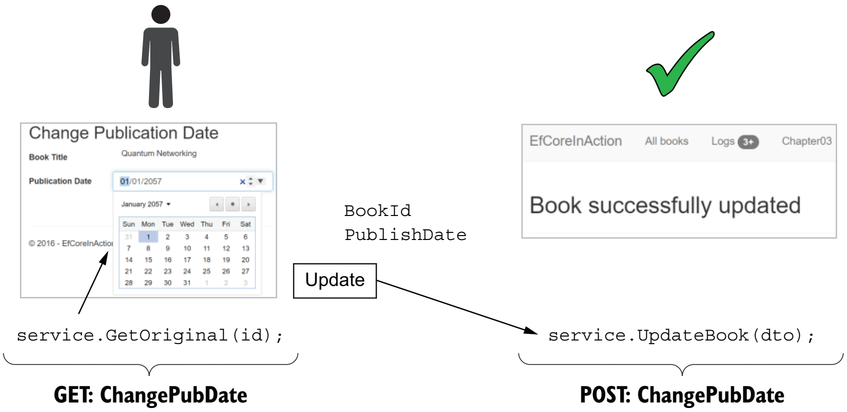


Figure 5.6 The two stages in changing the publication date of a book. The `GET` stage calls the `GetOriginal` method to show the user the book and its current publication date. Then the `POST` stage calls the `UpdateBook` method with the user set date.

You can see that the process has two stages:

- You show the admin user the current publication date and allow them to change it.
- The update is applied to the database, and you tell the user that it was successful.

To use parameter injection of your `ChangePubDateService` class, you need to do two things:

- Register your class, `ChangePubDateService`, with the DI so that it becomes a service you can inject by using DI.
- Use parameter injection to inject the class instance, `ChangePubDate`, into the two ASP.NET action methods that need it (GET and `POST`).

This approach works well for building ASP.NET Core applications, and I've used it in all my ASP.NET MVC projects for many years. In addition to providing good isolation and making testing easier, this approach makes the ASP.NET Core controller action methods much easier to write. You'll see in section 5.7.2 that the code inside the `ChangePubDate` action method is simple and short.

5.7.1 Registering your class as a DI service

You can register a class with DI in ASP.NET in numerous ways. The standard way is to add an `IChangePubDateService` interface to the class. Technically, you don't need an interface, but using one is good practice and can be helpful in unit-testing. You also use the interface in section 5.7.3 to make registering your classes simpler.

The following listing shows the `IChangePubDateService` interface. Don't forget that the ASP.NET Core controller will be dealing with something of type `IChangePubDateService`, so you need to make sure that all the public methods and properties are available in the interface.

Listing 5.6 The `IChangePubDateService` interface needed to register the class in DI

```
public interface IChangePubDateService
{
    ChangePubDateDto GetOriginal(int id);
    Book UpdateBook(ChangePubDateDto dto);
}
```

Then you register this interface/class with the DI service. The default way to do this in ASP.NET Core is to add a line to the `ConfigureServices` method in the `Startup` class. This listing shows the updated method, with the new code in bold. You add the `ChangePubDateService` as a transient, because you want a new version created every time you ask for it.

Listing 5.7 The ASP.NET Core `ConfigureService` method in the `Startup` class

```
public void ConfigureServices (IServiceCollection services)
{
```

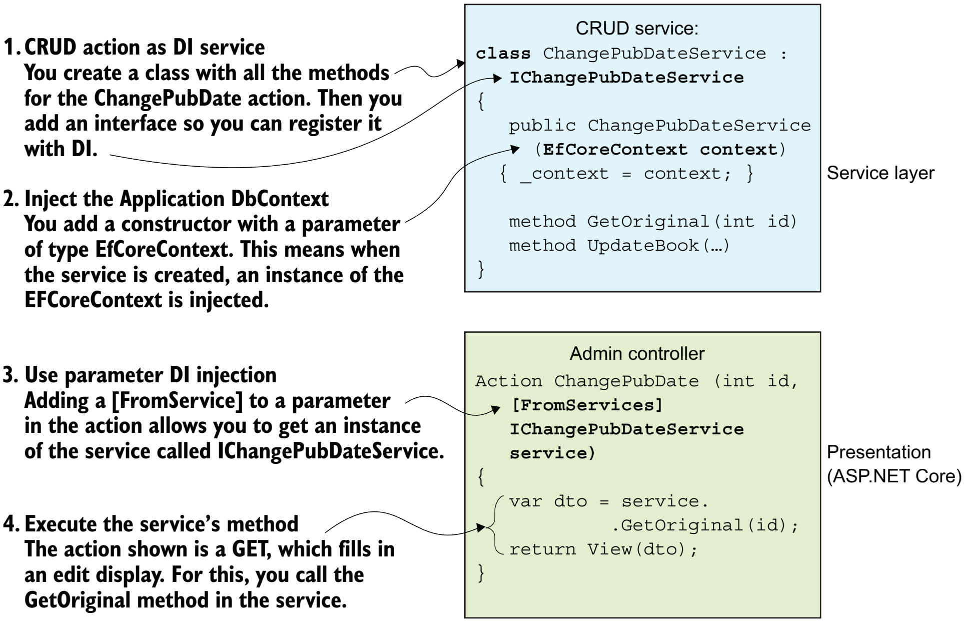
```
    // Add framework services.
    services.AddControllersWithViews();
    var connection = Configuration
        .GetConnectionString("DefaultConnection");
    services.AddDbContext<EfCoreContext>(
        options => options.UseSqlServer(connection))

    services.AddTransient
        <IChangePubDateService, ChangePubDateService>();
}
```

Registers the ChangePubDateService class as a service, with the IChangePubDateService interface as the way to access it

5.7.2 Injecting ChangePubDateService into the ASP.NET action method

Having set up the `ChangePubDateService` class as a service that can be injected via DI, now you need to create an instance in your ASP.NET Core `AdminController`. The two ASP.NET Core action methods are both called `ChangePubDate`; one is a `GET` to fill in the edit page, and one is a `POST` to do the update.

Figure 5.7 shows how DI creates the `ChangePubDateService` service, which has an instance of `EfCoreDbContext` injected via its constructor. Then the `ChangePubDateService` is injected into the `AdminController`'s `GET` action via parameter injection. As



1. CRUD action as DI service
You create a class with all the methods for the ChangePubDate action. Then you add an interface so you can register it with DI.

2. Inject the Application DbContext
You add a constructor with a parameter of type EfCoreContext. This means when the service is created, an instance of the EFCoreContext is injected.

```
class ChangePubDateService :
    IChangePubDateService
{
    public ChangePubDateService
     (EfCoreContext context)
   { _context = context; }

    method GetOriginal(int id)
    method UpdateBook(…)
}
```

3. Use parameter DI injection
Adding a [FromService] to a parameter in the action allows you to get an instance of the service called IChangePubDateService.

4. Execute the service's method
The action shown is a GET, which fills in an edit display. For this, you call the GetOriginal method in the service.

```
Action ChangePubDate (int id,
    [FromServices]
    IChangePubDateService
    service)
{
    var dto = service.
              .GetOriginal(id);
    return View(dto);
}
```

Figure 5.7 Using DI to provide a service often requires the DI provider to create other classes first. In this fairly simple case, there are at least four levels of DI. The `AdminController`'s `ChangePubDate` is called (bottom rectangle); then the `[FromServices]` attribute on one of the method's parameters tells the DI provider to create an instance of the `ChangePubDateService` class. The `ChangePubDateService` (top rectangle) class requires an instance of the `EfCoreDbContext` class, so the DI provider must create that instance too, which in turn requires the `DbContextOptions<EfCoreContext>` to be created so that the `EfCoreDbContext` `class` can be created.

you will see, the DI provider is called numerous times to create all the classes needed to handle the HTTP request.

You could have provided an instance of the `ChangePubDateService` class via constructor injection, as you did with the application's DbContext, but that approach has a downside. `AdminController` contains several other database update commands, such as adding a review to a book, adding a promotion to a book, and so on. Using DI constructor injection would mean you were needlessly creating an instance of `ChangePubDateService` class when one of these other commands is being called. By using DI parameter injection into each action, you take only the time and memory cost of creating the single service you need. The following listing shows the `ChangePubDate` ASP.NET `GET` action that's called when someone clicks the Admin > Change Pub Date link, wanting to change the publication date.

Listing 5.8 The `ChangePubDate` action method in `AdminController`

The action called if the user clicks the Admin > Change Pub Date link

Receives the primary key of the book that the user wants to change

```
public IActionResult ChangePubDate
    (int id,
    [FromServices]IChangePubDateService service)
{
    var dto = service.GetOriginal(id);
    return View(dto);
}
```

ASP.NET DI injects the ChangePubDateService instance.

Uses the service to set up a DTO to show the user

Shows the page that allows the user to edit the publication date

Line 3 (in bold) in this listing is the important one. You've used parameter injection to inject, via DI, an instance of the `ChangePubDateService` class. The same line is also in the `POST` version of the `ChangePubDate` action.

Note that the `ChangePubDateService` class needs the `EfCoreContext` class that's the application's DbContext, in its constructor. That's fine because DI is recursive; it'll keep filling in parameters, or other DI injections, as long as each class that's needed has been registered.

5.7.3 Improving registering your database access classes as services

Before leaving the topic of DI, I want to introduce a better way of registering your classes as services via DI. The previous example, in which you made your `ChangePubDateService` class into a service, required you to add code to register that class as a service in ASP.NET Core's `ConfigureServices`. This process works, but it's time-consuming and error-prone, as you need to add a line of code to register each class that you want to use as a service.

In the first edition of this book, I suggested using a DI library called Autofac (https://autofaccn.readthedocs.io/en/latest) because it has a command that registers all the classes with interfaces in an assembly (also known as a *project*). Since then, I've come across a tweet by David Fowler that links to a set of dependency injection container benchmarks; see http://mng.bz/go2l. From that page, I found out that the ASP.NET Core DI container is a lot faster than AutoFac! At that point, I built a library called `NetCore.AutoRegisterDi` (see http://mng.bz/5jDz), which has only one job: to register all classes with interfaces in an assembly by using the .NET Core DI Provider.

> **NOTE** After I created my `NetCore.AutoRegisterDi` library, Andrew Lock pointed me to an existing library called Scrutor; see his article at http://mng.bz/6gly. Scrutor has more features for selecting classes to register than my `NetCore.AutoRegisterDi` does, so do have a look at Scrutor.

How I organize the registering of services with the NET Core DI container

The `NetCore.AutoRegisterDi` library is simple: it scans one or more assembles; looks for standard public, nongeneric classes that have public interfaces; and registers them with NET Core's DI provider. It has some simple filtering and some lifetime-setting capabilities, but not much more (it's only ~80 lines of code). But this simple piece of code gives you two benefits over manually registering your classes/interfaces with the DI provider:

- It saves you time because you don't have to register every interface/class manually.
- More important, it automatically registers your interfaces/classes so that you don't forget.

The second reason is why I find this library to be so useful: I can't forget to register a service. The following listing shows you a typical call to the `NetCore.AutoRegisterDi` library.

Listing 5.9 Using `NetCore.AutoRegisterDi` to register classes as DI services

This method takes zero to many assemblies to scan. If no assembly is provided, it will scan the calling assembly.

You can get references to the assemblies by providing a class that is in that assembly.

```
var assembly1ToScan = Assembly.GetAssembly(typeof(ass1Class));
var assembly2ToScan = Assembly.GetAssembly(typeof(ass2Class));

service.RegisterAssemblyPublicNonGenericClasses(
       assembly1ToScan, assembly2ToScan)
    .Where(c => c.Name.EndsWith("Service"))
    .AsPublicImplementedInterfaces();
```

This optional filter system allows you to filter the classes that you want to register.

Registers all the classes that have public interfaces. By default, the services are registered as transient, but you can change that registration by adding a ServiceLifetime parameter or attributes.

I could put a call like the one shown in listing 5.9 in the `Configure` method in ASP.NET Core's `Startup` class that registers all the assemblies, but I don't. I prefer to add an extension method in every project that has classes that need to register as a DI service. That way, I have isolated the setup of each project into one class in each project that needs it.

Each extension method uses the `NetCore.AutoRegisterDi` library to register the standard classes/services in the project. The extension method also has space for additional code, such as handcoded registration of classes/services that can't be registered automatically, such as generic classes/services.

The following listing shows an example of the extension method in the service layer. This code needs the `NetCore.AutoRegisterDi` NuGet package to be added to that project.

Listing 5.10 Extension method in ServiceLayer that handles all the DI service registering

```
public static class NetCoreDiSetupExtensions
{
    public static void RegisterServiceLayerDi
        (this IServiceCollection services)
    {
        services.RegisterAssemblyPublicNonGenericClasses()
            .AsPublicImplementedInterfaces();


    }
}
```

The NetCore.AutoRegisterDi library understands NET Core DI, so you can access the IServiceCollection interface.

Creates a static class to hold my extension

This class is in the ServiceLayer, so I give the method a name with that Assembly name in it.

Calling the RegisterAssemblyPublicNonGenericClasses method without a parameter means that it scans the calling assembly.

This method will register all the public classes with interfaces with a Transient lifetime.

For handcoded registrations that NetCore.AutoRegisterDi can't do, such as generic classes

The Book App in part 1 of the book has classes/services that need registering in the ServiceLayer, BizDbAccess, and BizLogic projects. To do so, you copy the code in listing 5.10 into the other projects and change the name of the method so that each one can be identified. A call to each method automatically registers the standard services because by default, the `RegisterAssemblyPublicNonGenericClasses` scans the assembly that it's called from.

Now that you have individual versions of listing 5.8 in each of the three projects that need them, you need to call each one to set up each project. You do so by adding the following code to the `Configure` method in ASP.NET Core's `Startup` class.

Listing 5.11 Calling all your registration methods in the projects that need them

```
public void ConfigureServices(IServiceCollection services)
{
    //… other registrations left out

    services.RegisterBizDbAccessDi();
    services.RegisterBizLogicDi();
    services.RegisterServiceLayerDi();
}
```

This method in the Startup class sets up services for ASP.NET Core.

You add your registration extension methods here.

The result is that all the classes you have written with public interfaces in the ServiceLayer, BizDbAccess, and BizLogic projects will automatically be registered as DI services.

5.8 Deploying an ASP.NET Core application with a database

After developing your ASP.NET Core application with a database, at some point you'll want to copy it to a web server so that others can use it. This process is called *deploying* your application to a *host*. This section shows how.

> **NOTE** For more information on ASP.NET Core deployment, Andrew Lock's book *ASP.NET Core in Action*, 2nd ed. (Manning, 2020; see https://www.manning.com/books/asp-net-core-in-action-second-edition) has a chapter on deployment; or see Microsoft's online documentation at http://mng.bz/op7M.

5.8.1 Knowing where the database is on the web server

When you run your ASP.NET Core application locally during development, it accesses a database server on your development computer. This example uses Visual Studio, which comes with a local SQL server for development that's available via the reference `(localdb)\mssqllocaldb`. As explained in section 5.4.1, the connection string for that database is held in the appsettings.Development.json file.

When you deploy your application to a web server, Visual Studio by default rebuilds your application with the `ASPNETCORE_ENVIRONMENT` variable set to `Production`. This setting causes your application to try to load the appsetting.json file, followed by the appsettings.Production.json file. The appsettings.Production.json file is the place where you (or the publishing system) put the connection string for your host database.

> **TIP** At startup, appsettings.Production.json is read last and overrides any setting with the same name in the appsetting.json file. Therefore, you can put your development connection string setting in the appsetting.json file if you want to, but best practice is to put it in the appsettings.Development.json file.

You can set your hosted database's connection string manually with Visual Studio's Publish feature; right-click the ASP.NET Core project in Solution Explorer view and select Publish. When you publish your application, Visual Studio creates/updates the appsettings.Production.json file with the connection string you provided and deploys that file with the application. On startup, the constructor of the ASP.NET Core's `Startup` class reads both files, and the appsettings.Production.json connection string is used.

Most Windows hosting systems provide a Visual Studio publish profile that you can import to the Publish feature. That profile makes setting up deployment much easier, as it not only details where the ASP.NET Core application should be written to, but also provides the connection string for the hosted database.

Cloud systems such as Azure Web App service have a feature that can override properties in your appsettings.json file on deployment. This means you can set your database connection, which contains the database username and password, within Azure; your username and password never exist on your development system and, hence, are more secure.

5.8.2 Creating and migrating the database

When your application and its database are running on a web server, control of the database changes. On your development machine, you can do pretty much anything to the database, but after you deploy to a web server, the rules can change. Depending on the host or your company's business rules, what you can do to the database will vary.

A version of the Book App from the first edition of this book, for example, was hosted on a cost-effective (cheap!) shared hosting platform (WebWiz in the United Kingdom), which doesn't allow your application to create or delete the database. I've also used Microsoft's Azure cloud system, on which I can delete and create a database, but creating a database takes a long time.

The simplest approach, which works on all the systems I've come across, is getting the hosting system to create an empty database and then applying the commands to alter the database structure. The easiest way is via EF Core migrations, which I'm about to describe, but there are other ways.

> **WARNING** Before I start, I need to warn you that changing the database structure of a website needs to be approached carefully, especially for 24/7 websites that need to keep working during a database change. Lots of things can go wrong, and the effect could be lost data or a broken website.

This chapter describes EF Core migrations, which are a good system but has their limitations. Chapter 9 presents ways of handling database migrations, including more-sophisticated techniques, and discusses the pros and cons of each approach.

5.9 Using EF Core's migration feature to change the database's structure

This section describes how to use EF Core's migration feature to update a database. You can use migrations on both your development machine and your host, but as explained in section 5.8.2, the challenging one is the database on your web host. This book has is a whole chapter (chapter 9) on migrations, but this section gives you an overview of using migrations in ASP.NET Core applications.

5.9.1 Updating your production database

As you may remember from chapter 2, which briefly introduced EF Core migrations, you can type two commands into Visual Studio's Package Manager Console (PMC):

- `Add-Migration`—Creates migration code in your application to create/update your database structure
- `Update-Database`—Applies the migration code to the database referred to by the application's DbContext

The first command is fine, but the second command will update only the default database, which is likely to be on your development machine, not your production database. What happens when you want to deploy your web application to some sort of web host, and the database isn't at the right level to match the code? You have four ways to update your production database if you're using EF Core's migration feature:

- You can have your application check and migrate the database during startup.
- You can migrate the database in a continuous integration (CI) and continuous delivery (CD) pipeline.
- You can have a standalone application migrate your database.
- You can extract the SQL commands needed to update your database and then use a tool to apply those SQL commands to your production database.

The simplest option is the first one, which I'm going to describe here. It does have limitations, such as not being designed to work in multiple-instance web hosting (called *scaling out* in Azure). But having the application do the migration is simple and is a good first step in using EF Core's migrations in an ASP.NET Core application.

WARNING Microsoft recommends that you update a production database by using SQL commands, which is the most robust approach. But it requires quite a few steps and tools that you may not have on hand, so I cover the simpler `Database.Migrate` approach. Chapter 9 covers every aspect of database migrations, including the advantages and limitations of each approach.

5.9.2 Having your application migrate your database on startup

The advantage of having your application apply any outstanding database migrations at startup is that you can't forget to do it: deploying a new application will stop the old application and then start the new application. By adding code that's run when the application starts, you can call the `context.Database.Migrate` method, which applies any missing migrations to the database before the main application starts—simple, until it goes wrong, which is why chapter 9, dedicated to database migrations, discusses all these issues. But for now, let's keep to the simple approach.

Having decided to apply the migration on startup, you need to decide where to call your migration code. The recommended approach to adding any startup code to an ASP.NET Core application is to append your code to the end of the `Main` method in ASP.NET Core's `Program` class. The normal code in the `Main` method is shown in this code snippet:

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}
```

The best way to add the migration code is to build an extension method holding the EF Core code you want to run and append it after the `CreateHostBuilder(args).Build()` call. The following listing shows the ASP.NET Core's `Program` class with one new line (in bold) added to call your extension method, called `MigrateDatabaseAsync`.

NOTE I will be using async/await commands in this section. I cover async/await in section 5.10.

Listing 5.12 ASP.NET Core `Program` class, including a method to migrate the database

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var host = CreateHostBuilder(args).Build();
        await host.MigrateDatabaseAsync();
        await host.RunAsync();
    }
    //… other code not shown
}
```

You change the Main method to being async so that you can use async/await commands in your SetupDatabaseAsync method.

This call runs the Startup.Configure method, which sets up the DI services you need to setup/migrate your database.

Calls your extension method to migrate your database

At the end, you start the ASP.NET Core application.

The `MigrateDatabaseAsync` method should contain all the code you want to run at startup to migrate, and possibly seed, your database. The following listing shows one example of how you might use this method to migrate your database.

Listing 5.13 The `MigrateDatabaseAsync` extension method to migrate the database

Creates a scoped service provider. After the using block is left, all the services will be unavailable. This approach is the recommended way to obtain services outside an HTTP request.

```
public static async Task MigrateDatabaseAsync
    (this IHost webHost)
{
    using (var scope = webHost.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        using (var context = services
            .GetRequiredService<EfCoreContext>())
        {
            try
            {
                await context.Database.MigrateAsync();
                //Put any complex database seeding here
            }
            catch (Exception ex)
            {
                var logger = services
                    .GetRequiredService<ILogger<Program>>();
                logger.LogError(ex,
                "An error occurred while migrating the database.");

                throw;
            }
        }
    }
}
```

Creates an extension method that takes in IHost

Creates an instance of the application's DbContext that has a lifetime of only the outer using statement

Calls EF Core's MigrateAsync command to apply any outstanding migrations at startup

You can add a method here to handle complex seeding of the database if required.

If an exception occurs, you log the information so that you can diagnose it.

Rethrows the exception because you don't want the application to carry on if a problem with migrating the database occurs

The series of calls at the start of the listing is the recommended way to get a copy of the application's DbContext inside the `Configure` method in the ASP.NET Core `Startup` class. This code creates a scoped lifetime instance (see section 5.3.3) of the application's DbContext that can be safely used to access the database.

The key commands in listing 5.13, inside the `try` block (in bold), call EF Core's `MigrateAsync` command. This command applies any database migration that exists but hasn't already been applied to the database.

> **EF6** The EF Core approach to database setup is different from that of EF6.x. On first use of the DbContext, EF6.x runs various checks by using *database initializers*, whereas EF Core does nothing at all to the database on initialization. Therefore, you need to add your own code to handle migrations. The downside is that you need to write some code, but the upside is that you have total control of what happens.

SETTING UP INITIAL DATABASE CONTENT DURING STARTUP

In addition to migrating the database, you may want to add default data to the database at the same time, especially if it's empty. This process, called *seeding* the database, covers adding initial data to the database or maybe updating data in an existing database. The main way to seed your database with static data is via migrations, which I cover in chapter 9. The other option is to run some code when the migration has finished. This option is useful if you have dynamic data or complex updates that the migration seeding can't handle.

An example of running code after the migration is adding example `Books`, with `Authors`, `Reviews`, and so on to the Book App if no books are already present. To do this, you create an extension method, `SeedDatabaseAsync`, which is shown in the following listing. The code is added after the call to the `Database.MigrateAsync` method in listing 5.13.

Listing 5.14 Our example `MigrateAndSeed` extension method

```
public static async Task SeedDatabaseAsync
    (this EfCoreContext context)
{
    if (context.Books.Any()) return;

    context.Books.AddRange(
        EfTestData.CreateFourBooks());
    await context.SaveChangesAsync();
}
```

Extension method that takes in the application's DbContext

If there are existing books, you return, as you don't need to add any.

Database has no books, so you seed it; in this case, you add the default books.

SaveChangesAsync is called to update the database.

In this example `SeedDatabaseAsync` method, you check whether any books are in the database and then add them only if the database is empty (has just been created, for example). This example is a simple one, and here are others:

- Loading data from a file on startup (see the `SetupHelpers` class in the ServiceLayer in the associated GitHub repo)

- Filling in extra data after a specific migration—if you added a `FullName` property/column, for example, and wanted to fill it in from the `FirstName` and `LastName` columns

WARNING I tried doing a database update like the previous `FullName` example on a large database with tens of thousands of rows to update, and it failed. The failure occurred because the update was done via EF Core on startup, and it took so long for the ASP.NET Core application to start that Azure timed out the web application. I now know that I should have done the update by using SQL in the migration (see the example in section 9.5.2), which would have been a lot faster.

If you want to run your seed database method only when a new migration has been applied, you can use the DbContext method `Database.GetPendingMigrations` to get the list of migrations that are about to be applied. If this method returns an empty collection, there is no pending migration in the current database. You must call `GetPendingMigrations` before you execute the `Database.Migrate` method, because the pending migrations collection is empty when the `Migrate` method has finished.

EF6 In EF6.x, the `Add-Migration` command adds a class called `Configuration`, which contains a method called `Seed` that's run every time the application starts. EF Core uses the `HasData` configuration method, which allows you to define data to be added during a migration (chapter 9).

5.10 Using async/await for better scalability

Async/await is a feature that allows a developer to easily use *asynchronous programming*, running tasks in parallel. Up to this point in this book, I have not used async/await because I hadn't explained this feature. But you need to know that in real applications that have multiple requests happening at the same time, such as ASP.NET Core, most of your database commands will use async/await.

Async/await is a big topic, but in this section, you'll look only at how using async/await can benefit an ASP.NET Core's application scalability. It does this by releasing resources while waiting for the database server to carry out the command(s) that EF Core has asked it to do.

NOTE If you want to find out more about async/await's other features, such as running tasks in parallel, have a look at the Microsoft documentation at http://mng.bz/nM7K.

5.10.1 Why async/await is useful in a web application using EF Core

When EF Core accesses the database, it needs to wait for the database server to run the commands and return the result. For large datasets and/or complex queries, this process can take hundreds of milliseconds or even seconds. During that time, a web application is holding on to a thread from the application's thread pool. Each access to the web application needs a thread from the thread pool, and there's an upper limit.

Using an async/await version of an EF Core command means that the user's current thread is released until the database access finishes, so someone else can use that thread. Figure 5.8 shows two cases. In case A, two users are simultaneously accessing the website by using normal synchronous accesses, and they clash, so two threads are needed from the thread pool. In case B, user 1's access is a long-running database access that uses an async command to release the thread while it's waiting for the database. This allows user 2 to reuse the thread that the async command released while user 2 is waiting for the database.

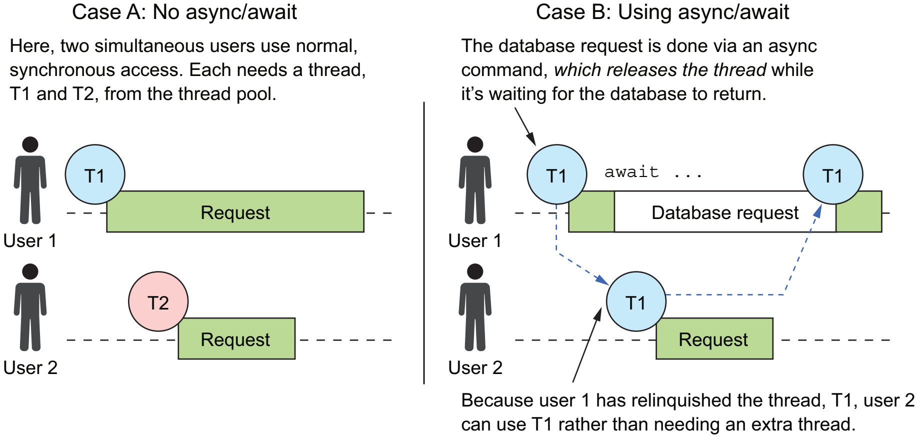


Figure 5.8 Differences in database access. In the normal, synchronous database access in case A, two threads are needed to handle the two users. In case B, user 1's database access is accomplished with an async command, which frees the thread, T1, making it available for user 2.

> **NOTE** You can read a more in-depth explanation of what async/await does in an ASP.NET web application at http://mng.bz/vz7M.

The use of async/await improves the scalability of your website: your web server will be able to handle more concurrent users. The downside is that async/await commands take slightly longer to execute because they run more code. A bit of analysis is needed to get the right balance of scalability and performance.

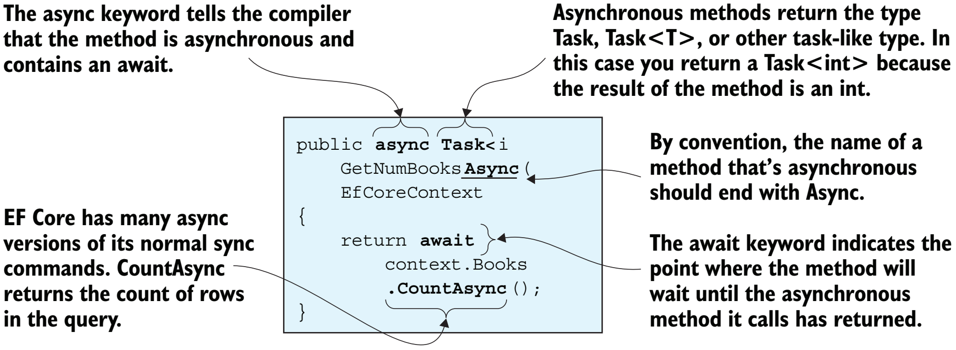
5.10.2 Where should you use async/await with database accesses?

The general advice from Microsoft is to use async methods wherever possible in a web application because they give you better scalability. In real applications, that's what I do. I haven't done it in the part 1 (and 2) Book App, because it's a little easier to understand the code without `await` statements everywhere, but the part 3 Book App, which is significantly enhanced, uses async throughout.

Sync commands are slightly faster than the equivalent async command (see table 14.5 for the actual differences), but the time difference is so small that sticking to Microsoft's guideline "Always use async commands in ASP.NET applications" is the right choice.

5.10.3 Changing over to async/await versions of EF Core commands

Let me start by showing you a method that calls an async version of an EF Core command; then I'll explain it. Figure 5.9 shows an async method that returns the total number of books in the database.



```
public async Task<i
    GetNumBooksAsync(
    EfCoreContext
{
    return await
        context.Books
        .CountAsync();
}
```

The async keyword tells the compiler that the method is asynchronous and contains an await.

Asynchronous methods return the type Task, Task<T>, or other task-like type. In this case you return a Task<int> because the result of the method is an int.

By convention, the name of a method that's asynchronous should end with Async.

EF Core has many async versions of its normal sync commands. CountAsync returns the count of rows in the query.

The await keyword indicates the point where the method will wait until the asynchronous method it calls has returned.

Figure 5.9 The anatomy of an asynchronous method, highlighting the parts of the code that are different from a normal synchronous method

EF Core contains an async version of all the applicable commands, all of which have a method name that ends with `Async`. As you saw in the preceding async method example, you need to carry the "async-ness" to the method in which you call the async EF Core command.

The rule is that after you use an async command, every caller must either be an async method or should pass on the task directly until it gets to the top-level caller, which must handle it asynchronously. ASP.NET Core supports async for all the main commands, such as controller actions, so this situation isn't a problem in such an application.

The next listing shows an async version of your `Index` action method from your `HomeController`, with the parts you have to change to make this command use an async database access, with the async parts in bold.

Listing 5.15 The async Index action method from the `HomeController`

```
public async Task<IActionResult> Index
    (SortFilterPageOptions options)
{
    var listService =
        new ListBooksService(_context);
```

You make the Index action method async by using the async keyword, and the returned type has to be wrapped in a generic task.

```
    var bookList = await listService
        .SortFilterPage(options)
        .ToListAsync();

    return View(new BookListCombinedDto
        (options, bookList));
}
```

You must await the result of the ToListAsync method, which is an async command.

You can change SortFilterPage to async by replacing .ToList() with .ToListAsync().

Because you design your `SortFilterPage` method to return `IQueryable<T>`, it's simple to change database access to async by replacing the `ToList` method with the `ToListAsync` method.

> **TIP** Business logic code is often a good candidate for using async databases' access methods because their database accesses often contain complex read/write commands. I've created async versions of the `BizRunners` in case you need them. You can find them in the service layer in the BizRunners directory (see http://mng.bz/PPlw).

Another part of async is the `CancellationToken`, a mechanism that allows you to stop an async method manually or on a timeout. All the async LINQ and EF Core commands, such as `SavChangesAsync`, take in an optional `CancellationToken`. Section 5.11 demonstrates the use of a `CancellationToken` to stop any recurring background tasks when ASP.NET Core is stopped.

5.11 Running parallel tasks: How to provide the DbContext

In some situations, running more than one thread of code is useful. By this, I mean that running a separate *task*—a parallel set of code that runs "at the same time" as the main application. I put "at the same time" in quotes because if there's only one CPU, the two tasks need to share it.

Parallel tasks are useful in various scenarios. Say you're accessing multiple, external sources that you need to wait for before they return a result. By using multiple tasks running in parallel, you gain performance improvements. In another scenario, you might have a long-running task, such as processing order fulfillment in the background. You use parallel tasks to avoid blocking the normal flow and making your website look slow and unresponsive. Figure 5.10 shows an example background task in which a long-running process is run on another thread so that the user isn't held up.

Running parallel tasks isn't specific to ASP.NET Core; it can occur in any application. But larger web applications often use this feature, so I explain it in this chapter. The solution you will build is a background service that runs every hour and logs how many `Reviews` are in the database. This simple example will show you how to do two things:

- Obtain an instance of your application's DbContext to run in parallel
- Use the ASP.NET Core's `IHostedService` feature to run your background task

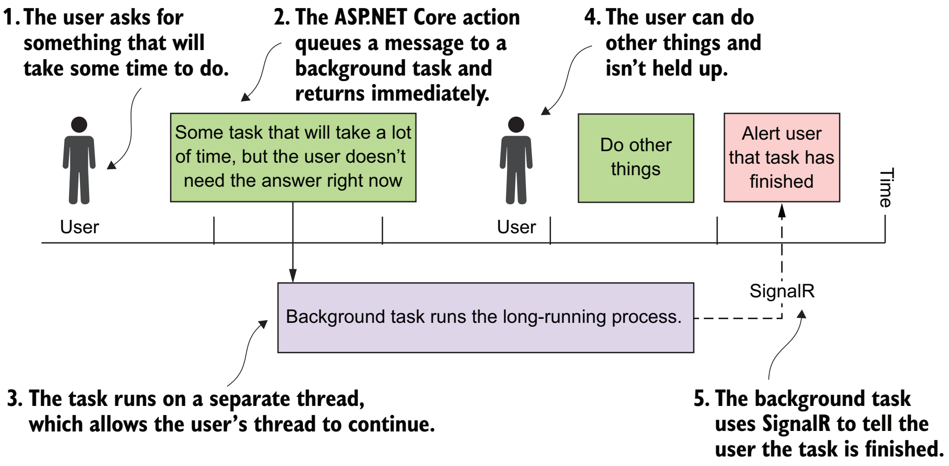


Figure 5.10 Moving long-running processes to a background task that runs in parallel to the main website, which makes the website feel more responsive. In this example, I use an ASP.NET Core `backgroundService` to run the long-running task. When the task is finished, it uses `SignalR` to update the user's screen with a message saying that the long-running task has finished successfully. (`SignalR` is a library that allows an ASP.NET Core app to send messages to the user's screen.)

5.11.1 Obtaining an instance of your application's DbContext to run in parallel

If you want to run any code that uses EF Core in parallel, you can't use the normal approach of getting the application's DbContext because EF Core's DbContext isn't thread-safe; you can't use the same instance in multiple tasks. EF Core will throw an exception if it finds that the same DbContext instance is used in two tasks.

In ASP.NET Core, the correct way to get a DbContext to run in the background is by using a DI scoped service. This scoped service allows you to create, via DI, a DbContext that's unique to the task that you're running. To do this, you need to do three things:

- Get an instance of the `IServiceScopeFactory` via constructor injection.
- Use the `IServiceScopeFactory` to a *scoped DI service*.
- Use the scoped DI service to obtain an instance of the application's DbContext that is unique to this scope.

The following listing shows the method in your background task that uses the `IServiceScopeFactory` to obtain a unique instance of your application's DbContext. This method counts the number of `Reviews` in the database and logs that number.

Listing 5.16 The method inside the background service that accesses the database

```
private async Task DoWorkAsync(CancellationToken stoppingToken)
{
    using (var scope = _scopeFactory.CreateScope())
    {
```

The IHostedService will call this method when the set period has elapsed.

Uses the ScopeProviderFactory to create a new DI scoped provider

```
        var context = scope.ServiceProvider
            .GetRequiredService<EfCoreContext>();
        var numReviews = await context.Set<Review>()
            .CountAsync(stoppingToken);
        _logger.LogInformation(
            "Number of reviews: {numReviews}", numReviews);
    }
}
```

Logs the information

Because of the scoped DI provider, the DbContext instance created will be different from all the other instances of the DbContext.

Counts the reviews, using an async method. You pass the stoppingToken to the async method because doing so is good practice.

The important point of the code is that you provide `ServiceScopeFactory` to each task so that it can use DI to get a unique instance of the DbContext (and any other scoped services). In addition to solving the DbContext thread-safe issue, if you are running the method repeatedly, it's best to have a new instance of the application's DbContext so that data from the last run doesn't affect your next run.

5.11.2 Running a background service in ASP.NET Core

Earlier, I described how to get a thread-safe version of the application's DbContext; now you'll use it in a background service. The following background example isn't as complex as the one show in figure 5.10, but it covers how to write and run background services.

ASP.NET Core has a feature that allows you to run tasks in the background. This situation isn't really a database issue, but I show you the code for completeness. (I recommend that you look at Microsoft's ASP.NET Core documentation on background tasks at http://mng.bz/QmOj.) This listing shows the code that runs in another thread and calls the `DoWorkAsync` method shown in listing 5.16 every hour.

Listing 5.17 An ASP.NET Core background service that calls `DoWorkAsync` every hour

Inheriting the BackgroundService class means that this class can run continuously in the background.

```
public class BackgroundServiceCountReviews : BackgroundService
{
    private static TimeSpan _period =
        new TimeSpan(0,1,0,0);

    private readonly IServiceScopeFactory _scopeFactory;
    private readonly ILogger<BackgroundServiceCountReviews> _logger;

    public BackgroundServiceCountReviews(
        IServiceScopeFactory scopeFactory,
        ILogger<BackgroundServiceCountReviews> logger)
    {
        _scopeFactory = scopeFactory;
        _logger = logger;
    }
```

Holds the delay between each call to the code to log the number of reviews

The IServiceScopeFactory injects the DI service that you use to create a new DI scope.

```
    protected override async Task ExecuteAsync
        (CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            await DoWorkAsync(stoppingToken);
            await Task.Delay(_period, stoppingToken);
        }
    }

    private async Task DoWorkAsync…
    //see listing 5.16
}
```

The BackgroundService class has a ExecuteAsync method that you override to add your own code.

This loop repeatably calls the DoWorkAsync method, with a delay until the next call is made.

You need to register your background class with the NET DI provider, using the `AddHostedService` method. When the Book App starts, your background task will be run first, but when your background task gets to a place where it calls an async method and uses the `await` statement, control goes back to the ASP.NET Core code, which starts up the web application.

5.11.3 Other ways of obtaining a new instance of the application's DbContext

Although DI is the recommended method to get the application's DbContext, in some cases, such as a console application, DI may not be configured or available. In these cases, you have two other options that allow you to obtain an instance of the application's DbContext:

- Move your configuration of the application's DbContext by overriding the `OnConfiguring` method in the DbContext and placing the code to set up the DbContext there.
- Use the same constructor used for ASP.NET Core and manually inject the database options and connection string, as you do in unit tests (see chapter 17).

The downside of the first option is it uses a fixed connection string, so it always accesses the same database, which could make deployment to another system difficult if the database name or options change. The second option—providing the database options manually—allows you to read in a connection string from the appsettings.json or a file inside your code.

Another issue to be aware of is that each call will give you a new instance of the application's DbContext. From the discussions of lifetime scopes in section 5.3.3, at times you might want to have the same instance of the application's DbContext to ensure that tracking changes works. You can work around this issue by designing your application so that one instance of the application's DbContext is passed between all the code that needs to collaborate on database updates.

Summary

- ASP.NET Core uses dependency injection (DI) to provide the application's DbContext. With DI, you can dynamically link parts of your application by letting DI create class instances as required.
- The `ConfigureServices` method in ASP.NET Core's `Startup` class is the place to configure and register your version of the application's DbContext by using a connection string that you place in an ASP.NET Core application setting file.
- To get an instance of the application's DbContext to use with your code via DI, you can use constructor injection. DI will look at the type of each of the constructor's parameters and attempt to find a service for which it can provide an instance.
- Your database access code can be built as a service and registered with the DI. Then you can inject your services into the ASP.NET Core action methods via parameter injection: the DI will find a service that finds the type of an ASP.NET Core action method's parameter that's marked with the attribute `[FromServices]`.
- Deploying an ASP.NET Core application that uses a database requires you to define a database connection string that has the location and name of the database on the host.
- EF Core's migration feature provides one way to change your database if your entity classes and/or the EF Core configuration change. The `Migrate` method has some limitations when used on cloud hosting sites that run multiple instances of your web application.
- Async/await tasking methods on database access code can make your website handle more simultaneous users, but performance could suffer, especially on simple database accesses.
- If you want to use parallel tasks, you need to provide a unique instance of the application's DbContext by creating a new scoped DI provider.

For readers who are familiar with EF6.x:

- The way you obtain an instance of the application's DbContext in ASP.NET Core is via DI.
- Compared with EF6.x, EF Core has a different approach to creating the first instance of a DbContext. EF6.x has database initializers and can run a `Seed` method. EF Core has none of these EF6.x features but leaves you to write the specific code you want to run at startup.
- Seeding the database in EF Core is different from the way EF6.x works. The EF Core approach adds seeding to migrations, so they are run only if a migration is applied to the database; see chapter 9 for more information.

6

Tips and techniques for reading and writing with EF Core

This chapter covers

- Selecting the right approach to read data from the database
- Writing queries that perform well on the database side
- Avoiding problems when you use Query Filters and special LINQ commands
- Using AutoMapper to write `Select` queries more quickly
- Writing code to quickly copy and delete entities in the database

The first four chapters cover different ways to read/write to a database, and in chapter 5, you used that information to build the Book App—an ASP.NET Core web application. This chapter brings together lots of different tips and techniques for reading and writing data with EF Core.

The chapter is split into two sections: reading from the database and writing to the database. Each section covers certain read/write issues you may come across, but at the same time explains how EF Core achieves the solutions. The aim is to give you lots of practical tips by solving different problems and, at the same time, deepen your knowledge of how EF Core works. The tips are useful, but in the long run, becoming an expert on EF Core is going to make you a better developer.

> **TIP** Don't forget that the companion Git repo (http://mng.bz/XdlG) contains unit tests for every chapter of the book. For this chapter, look in the `Test` project in the master branch for classes starting with `Ch06_`. Sometimes, seeing the code is quicker than reading the words.

6.1 Reading from the database

This section covers different aspects and examples of reading data from a database. The aim is to expose you to some of the inner working of EF Core by looking at different problems and issues. On the way, you will pick up various tips that may be useful as you build applications with EF Core. Here is the list of topics on reading from the database via EF Core:

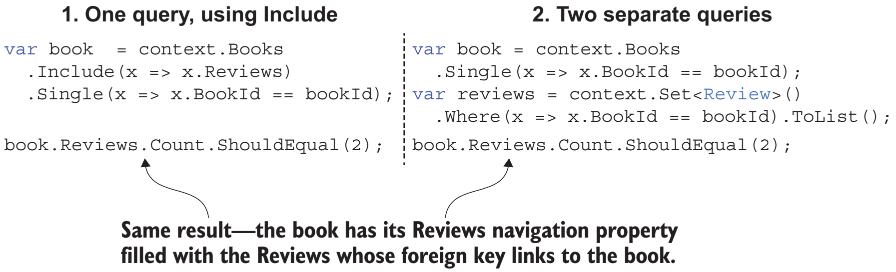
- Exploring the relational fixup stage in a query
- Understanding what `AsNoTracking` and its variant do
- Reading in hierarchical data efficiently
- Understanding how the `Include` method works
- Making loading navigational collections fail-safe
- Using Query Filters in real-world situations
- Considering LINQ commands that need special attention
- Using AutoMapper to automate building `Select` queries
- Evaluating how EF Core creates an entity class when reading data in

6.1.1 Exploring the relational fixup stage in a query

When you query the database by using EF Core, a stage called relational fixup runs to fill in the navigational properties of other entity classes included in the query. I described this process in section 1.9.2, where the `Book` entity was linked to its `Author`. Up to this point, all the queries you have seen link only the entity classes read in by the current query. But in fact, the relational fixup on a normal, read-write query can link outside a single query to any tracked entities, as described in this section.

Whenever you read in entity classes as tracked entities (your query didn't include the command `AsNoTracking`), the relation fixup stage will run to link up navigational properties. The important point is that the relation fixup stage doesn't only look at the data in your query; it also looks at all the existing tracked entities when it's filling in the navigational properties. Figure 6.1 shows two ways to load a `Book` with its `Reviews`, both of which fill in the `Book`'s `Reviews` navigational property.

As this simple example shows, the relational fixup that is run when a query has finished will fill in any navigational links based on the database key constraints, and it's pretty powerful, If you loaded all the `Books`, `Reviews`, `BookAuthor`, and `Authors` in four separate queries, for example, EF Core would correctly link up all the navigational



1. One query, using Include

```
var book   = context.Books
  .Include(x => x.Reviews)
  .Single(x => x.BookId == bookId);

book.Reviews.Count.ShouldEqual(2);
```

2. Two separate queries

```
var book = context.Books
  .Single(x => x.BookId == bookId);
var reviews = context.Set<Review>()
  .Where(x => x.BookId == bookId).ToList();
book.Reviews.Count.ShouldEqual(2);
```

Same result—the book has its Reviews navigation property filled with the Reviews whose foreign key links to the book.

Figure 6.1 This figure shows a single query that loads a `Book` with its `Reviews`, using the `Include` method to load the `Reviews` (see code on the left). The query on the right loads the book without its `Reviews`; then it does a second query that loads the `Reviews` separately. Both versions of the code produce the same result: a `Book` entity is loaded, and its `Reviews` navigational property is also loaded, with the `Reviews` linked to that `Book`.

properties. The following code snippet does just that: the books read in the first line start with no relationships filled in, but by the end of the four lines of code, the book's `Reviews` and `AuthorsLink` navigational properties are filled in, and the `BookAuthor`'s `Book` and `Author` navigational properties are also filled in:

```
var books = context.Books.ToList();
var reviews = context.Set<Review>().ToList();
var authorsLinks = context.Set<BookAuthor>().ToList();
var authors = context.Authors.ToList();
```

This feature of EF Core allows you to do some useful things. In section 6.1.3, you'll learn how to read hierarchical data efficiently by using this technique.

6.1.2 Understanding what AsNoTracking and its variant do

When you query the database via EF Core, you are doing so for a reason: to alter the data read in, such as changing the `Title` property in the `Book` entity, or to perform a read-only query, such as displaying the `Books` with their prices, authors, and so on. This section covers how the `AsNoTracking` and `AsNoTrackingWithIdentityResolution` methods improve the performance of a read-only query and affect the data read in. The following code snippet from chapter 1 uses `AsNoTracking` to display a list of `Books` and their `Authors` on the console:

```
var books = context.Books
            .AsNoTracking()
            .Include(a => a.Author)
            .ToList();
```

A normal query without either of two `AsNoTracking` methods will track the entity classes loaded by the query, allowing you to update or delete the entity classes you've loaded. But if you need only a read-only version, you can include two methods in your query. Both methods improve performance and ensure that changes to the data won't be written back to the database, but there are slight differences in the relationships returned:

- `AsNoTracking` produces a quicker query time but doesn't always represent the exact database relationships.
- `AsNoTrackingWithIdentityResolution` typically is quicker than a normal query but slower than the same query with `AsNoTracking`. The improvement is that the database relationships are represented correctly, with a entity class instance for each row in the database.

Let's start by looking at the differences in the data returned by a query that uses the two `AsNoTracking` variants. To give you the best performance, the `AsNoTracking` method doesn't execute the feature called identity resolution that ensures that there is only one instance of an entity per row in the database. Not applying the identity resolution feature to the query means that you might get an extra instances of entity classes.

Figure 6.2 shows what happens when you use the `AsNoTracking` and `AsNoTrackingWithIdentityResolution` methods on the super-simple database in chapter 1. That example has four books, but the first two books have the same author. As the figure shows, the `AsNoTracking` query creates four `Author` class instances, but the database has only three rows in the Author table.

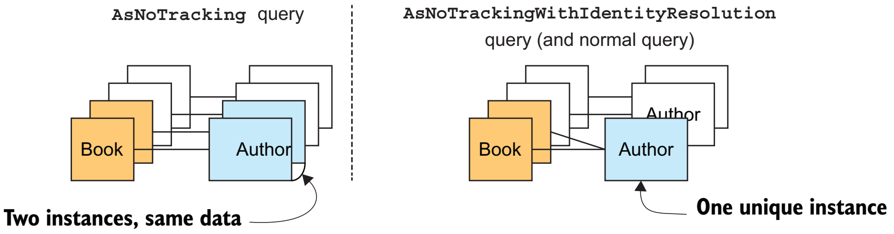


Figure 6.2 The first two books have the same author, Martin Fowler. In the `AsNoTracking` query on the left, EF Core creates four instances of the `Author` class, two of which contain the same data. A query containing `AsNoTrackingWithIdentityResolution` (or a normal query) on the right creates only three instances of the `Author` class, and the first two books point to the same instance.

In most read-only situations, such as displaying each book with the author's name, having four `Author` class instances doesn't matter because the duplicate classes contain the same data. In these types of read-only queries, you should use the `AsNoTracking` method because it produces the fastest query.

But if you are using the relationships in some way, such as to create a report of books which linked to other books by the same author, the `AsNoTracking` method might cause a problem. In a case like that one, you should use the `AsNoTrackingWithIdentityResolution` method.

HISTORY Some history: before EF Core 3.0, the `AsNoTracking` method included the identity resolution stage, but in EF Core 3.0, which had a big focus on performance, the identity resolution was removed from the `AsNoTracking` method. Removing the identity-resolution call produced some problems with existing applications, so EF Core 5 added the `AsNoTrackingWithIdentityResolution` method to fix the problems.

To give you an idea of the performance differences, I did a simple test of three queries, loading a hundred `Books` with their `Reviews`, `BookAuthor`, and `Author` entities. Table 6.1 shows the timings (second query).

Table 6.1 Result of running the same query using a normal, read-write query and queries that contain the `AsNoTracking` and `AsNoTrackingWithIdentityResolution` methods

`AsNoTracking` variants	Time (ms)	Percentage difference
`- no AsNoTracking` (normal query)	95	100%
`AsNoTracking`	40	42%
`AsNoTrackingWithIdentityResolution`	85	90%

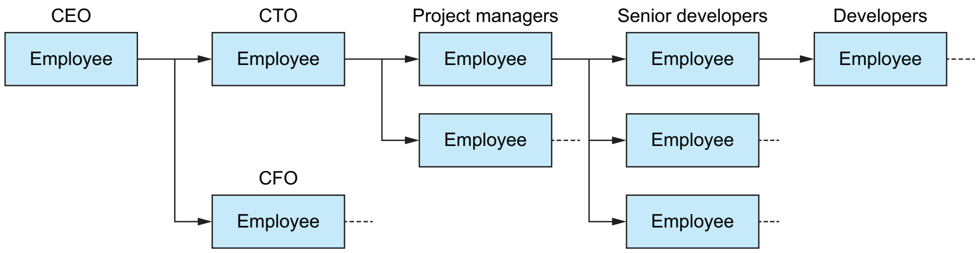
As you can see, `AsNoTracking` is fastest in this (unscientific) test and something like twice as fast as a normal query, so it's worth using. The `AsNoTrackingWithIdentityResolution` method is only slightly faster (in this case) than the normal read-write query, but as in the `AsNoTracking` version, the entities aren't tracked, which improves the performance of `SaveChanges` when it's looking for updated data.

Another feature of the `AsNoTracking` and `AsNoTrackingWithIdentityResolution` methods is that the relational fixup stage (see section 6.1.1) works only within the query. As a result, two queries using `AsNoTracking` or `AsNoTrackingWithIdentityResolution` will create new instances of each entity even if the first query loaded the same data. With normal queries, two separate queries would return the same entity class instances, because the relational fixup stage works across all tracked entities.

6.1.3 Reading in hierarchical data efficiently

I once worked for a client that had a lot of *hierarchical data*—data that has a series of linked entity classes with an indeterminate depth. The problem was that I had to parse the whole hierarchy before I could display it. I initially did this by eager loading for the first two levels; then I used explicit loading for deeper levels. This technique worked, but performance was slow, and the database was overloaded with lots of single database accesses.

This situation got me thinking: If the normal query relational fixup is so clever, could it help me improve the performance of the query? It could! Let me give you an example, using employees of a company. Figure 6.3 shows you the hierarchical structure of a company we want to load.



Hierarchical data can have an unknown level of depth—that is, you don't know how many .Include(x => x.WorksForMe) but need to load all the data.

Figure 6.3 One example of hierarchical data. The problem with this sort of data is that you don't know how deep it goes. But it turns out that one `.Include(x => x.WorksForMe)` is all you need. Then the relational fixup stage of the query will link the hierarchical data in the correct manner.

You could use `.Include(x => x.WorksForMe).ThenInclude(x => x.WorksForMe)` and so on, but a single `.Include(x => x.WorksForMe)` is enough, as the relational fixup can work out the rest. The next listing provides an example in which you want a list of all the employees working in development, with their relationships. The LINQ in this query is translated into one SQL query.

Listing 6.1 Loading all the employees working in development, with their relationships

```
var devDept = context.Employees
    .Include(x => x.WorksForMe)
    .Where(x => x.WhatTheyDo.HasFlag(Roles.Development))
    .ToList();
```

The database holds all the Employees.

One Include is all you need; relational fixup will work out what is linked to what.

Filters the employees down to ones who work in development

Listing 6.1 provides a tracked version of the hierarchical data, but if you want a read-only version, you can add the `AsNoTrackingWithIdentityResolution` method to the query. Note that `AsNoTracking` won't work, because the linking of the relationships relies on EF Core's relational fixup feature, which is turned off in the `AsNoTracking` method.

Before I found this approach, I was using explicit loading, which produced poor-performing queries. Swapping to this approach improved the time the single query took and also reduced the load on the database server.

NOTE You do need to work out which relationship to `Include`. In this case, I have a `Manager` navigational property (single) and a `WorksForMe` navigational property (collection). It turns out that including the `WorksForMe` property fills in both the `WorksForMe` collection and the `Manager` property. But including the `Manager` navigational property means that the `WorksForMe` collection is created only if there are entities to link to; otherwise, the

`WorksForMe` collection is null. I don't know why the two difference `Include` usages are different; that's why I test everything to make sure that I know how EF Core works.

6.1.4 Understanding how the Include method works

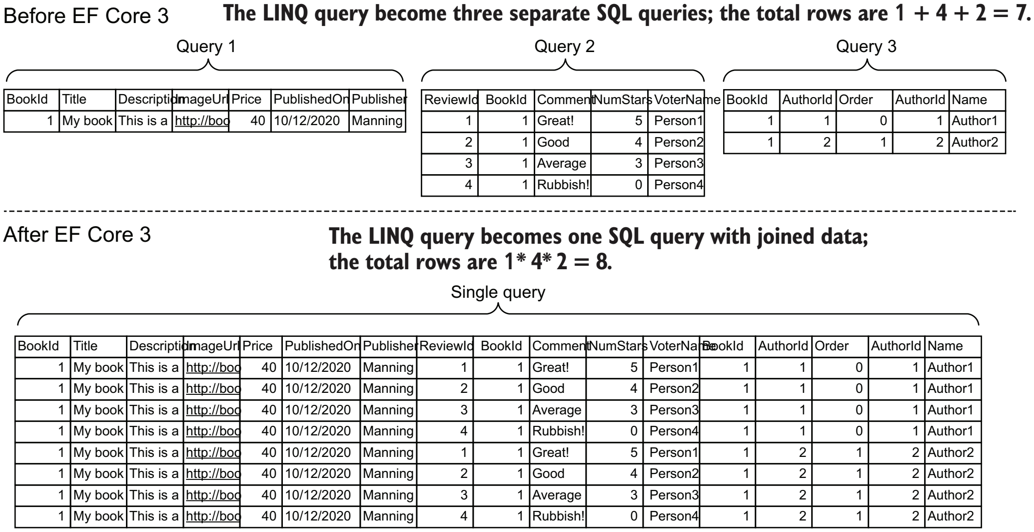
The simplest way to load an entity class with its relationships is to use the `Include` method, which is easy to use and normally produces an efficient database access. But it is worth knowing how the `Include` method works and what to watch out for.

The way that the `Include` method was converted to SQL changed when EF Core 3.0 came along. The EF Core 3.0 change provides performance improvements in many situations, but for some complex queries, it has a negative effect on performance. Take an example from the Book App database, and look at loading a `Book` with its `Reviews` and `Authors`. The following code snippet shows the query:

```
var query = context.Books
    .Include(x => x.Reviews)
    .Include(x => x.AuthorsLink)
        .ThenInclude(x => x.Author);
```

Figure 6.4 shows the different SQL queries produced by EF Core 2.2 and EF Core 3.0 for a `Book` that has four `Reviews` and two `Authors`.

The benefit of the EF Core 3.0 way of handling loading collections relationships is performance, which in many situations is quicker. I did a simple experiment, loading `Books` with ten `Reviews` and two `Authors` in EF Core 2.1 and EF Core 3.0, and the EF



BookId	Title	Descriptio	ImageUrl	Price	PublishedOn	Publisher
1	My book	This is a	http://boo	40	10/12/2020	Manning

ReviewId	BookId	Comment	NumStars	VoterName
1	1	Great!	5	Person1
2	1	Good	4	Person2
3	1	Average	3	Person3
4	1	Rubbish!	0	Person4

BookId	AuthorId	Order	AuthorId	Name
1	1	0	1	Author1
1	2	1	2	Author2

BookId	Title	Descriptio	ImageUrl	Price	PublishedOn	Publisher	ReviewId	BookId	Comment	NumStars	VoterNa	BookId	AuthorId	Order	AuthorId	Name
1	My book	This is a	http://boo	40	10/12/2020	Manning	1	1	Great!	5	Person1	1	1	0	1	Author1
1	My book	This is a	http://boo	40	10/12/2020	Manning	2	1	Good	4	Person2	1	1	0	1	Author1
1	My book	This is a	http://boo	40	10/12/2020	Manning	3	1	Average	3	Person3	1	1	0	1	Author1
1	My book	This is a	http://boo	40	10/12/2020	Manning	4	1	Rubbish!	0	Person4	1	1	0	1	Author1
1	My book	This is a	http://boo	40	10/12/2020	Manning	1	1	Great!	5	Person1	1	2	1	2	Author2
1	My book	This is a	http://boo	40	10/12/2020	Manning	2	1	Good	4	Person2	1	2	1	2	Author2
1	My book	This is a	http://boo	40	10/12/2020	Manning	3	1	Average	3	Person3	1	2	1	2	Author2
1	My book	This is a	http://boo	40	10/12/2020	Manning	4	1	Rubbish!	0	Person4	1	2	1	2	Author2

Figure 6.4 Comparing the way that EF Core loads data before and after EF Core 3 was released. The top version is how EF Core worked prior to EF Core 3—it used separate database queries to read in any collections. The lower version is what EF Core 3 and above do—it combines all the data into one big query.

Core 3.0 version was approximately 20% faster. But in some specific situations, it can be very slow indeed, as I cover next.

Performance problems occur if you have multiple collection relationships that you want to include in the query, and some of those relationships have a large number of entries in the collection. You can see the problem by looking at the two calculations on the far-right side of figure 6.4. This figure shows that the number of rows read in via EF Core versions before 3.0 is calculated by *adding* the rows. But in EF Core 3.0 and later, the number of rows read is calculated by *multiplying* the rows. Suppose that you are loading 3 relationships, each of which has 100 rows. The pre-3.0 version of EF Core would read in 100+100+100 = 300 rows, but EF Core 3.0 and later would use 100 * 100 * 100 = 1 million rows.

To see the performance issues, I created a test in which an entity had three one-to-many relationships, each of which had 100 rows in the database. The following snippet shows the normal `Include` approach to loading relationships in a query, which took 3500 milliseconds (a terrible result!):

```
var result = context.ManyTops
    .Include(x => x.Collection1)
    .Include(x => x.Collection2)
    .Include(x => x.Collection3)
    .Single(x => x.Id == id);
```

Fortunately, EF Core 5 provides a method called `AsSplitQuery` that tells EF Core to read each `Include` separately, as in the following listing. This operation took only 100 milliseconds, which is about 50 times faster.

Listing 6.2 Reading relationships separately and letting relational fixup join them up

```
var result = context.ManyTops
    .AsSplitQuery()
    .Include(x => x.Collection1)
    .Include(x => x.Collection2)
    .Include(x => x.Collection3)
    .Single(x => x.Id == id)
```

Causes each Include to be loaded separately, thus stopping the multiplication problem

If you find that a query that uses multiple `Includes` is slow, it could be because two or more included collections contain a lot of entries. In this case, add the `AsSplitQuery` method before your `Includes` to swap to the separate load of every included collection.

6.1.5 *Making loading navigational collections fail-safe*

I always try to make any code fail-safe, by which I mean that if I make a mistake in my code, I'd rather it fail with an exception than do the wrong thing silently. One area I worry about is forgetting to add the correct set of `Includes` when I'm loading an entity with relationships. It seems that I would never forget to do that, but in applications with lots of relationships, it can easily happen. In fact, I have done it many times, including in my clients' applications, which is why I use a fail-safe approach. Let me explain the problem and then my solution.

For any navigational property that uses a collection, I often see developers assign an empty collection to a collection navigational property, either in the constructor or via an assignment to the property (see the following listing).

Listing 6.3 A entity class with navigational collections set to an empty collection

```
public class BookNotSafe
{
    public int Id { get; set; }
    public ICollection<ReviewNotSafe> Reviews { get; set; }

    public BookNotSafe()
    {
        Reviews = new List<ReviewNotSafe>();
    }
}
```

This navigational property called Reviews has many entries—that is, a one-to-many relationship.

The navigational property called Reviews is preloaded with an empty collection, making it easier to add ReviewNotSave to the navigational property when the primary entity, BookNotSafe, is created.

Developers do this to make it easier to add entries to a navigational collection on a newly created instance of an entity class. The downside is that if you forget the `Include` to load a navigational property collection, you get an empty collection when the database might have data that should fill that collection.

You have another problem if you want to replace the whole collection. If you don't have the `Include`, the old entries in the database aren't removed, so you get a combination of new and old entities, which is the wrong answer. In the following code snippet (adapted from listing 3.17), instead of replacing the two existing `Reviews`, the database ends up with three `Reviews`:

```
var book = context.Books
      //missing .Include(x => x.Reviews)
       .Single(p => p.BookId == twoReviewBookId);

book.Reviews = new List<Review>{ new Review{ NumStars = 1}};
context.SaveChanges();
```

Another good reason not to assign an empty collection to a collection is performance. If you need to use explicit loading of a collection, for example, and you know that it's already loaded because it's not null, you can skip doing the (redundant) explicit loading. Also, in chapter 13, I select the best-performing way to add a new `Review` entity class to a `Book` entity class, depending on whether the `Book`'s `Reviews` collection property is already loaded.

So in my code (and throughout this book), I don't preload any navigational properties with a collection. Instead of failing silently when I leave out the `Include` method, I get a `NullReferenceException` when the code accesses the navigational collection property. To my mind, that result is much better than getting the wrong data.

6.1.6 Using Global Query Filters in real-world situations

Global Query Filters (shortened to Query Filter) were introduced in section 3.5 to implement a soft-delete feature. In this section, you'll look at some of the issues involved in using soft delete in real applications. You will also look at using Query Filters to produce multitenant systems.

SOFT DELETE IN REAL-WORLD APPLICATIONS

The soft-delete feature is useful because the users of the application get a second chance when they delete something. Two of my clients both had applications that used the soft-delete feature on nearly every entity class. Typically, a normal user would delete something, which in fact meant soft-deleting it, and an admin person could undelete the item. Both applications were complex and quite different, so I learned a lot about implementing soft delete.

First, soft delete doesn't work like the normal database delete command. With database deletes, if you delete a `Book`, you would also delete all the `PriceOffer`, `Reviews`, and `AuthorLinks` linked to the `Book` you deleted (see section 3.5.3). That situation doesn't happen with soft delete, which has some interesting issues.

If you soft-delete a `Book`, for example, the `PriceOffer`, `Reviews,` and `AuthorLinks` are still there, which can cause problems if you don't think things through. In section 5.11.1, you built a background process that logged the number of `Reviews` in the database on every hour. If you soft-deleted a `Book` that had ten `Reviews,` you might expect the number of `Reviews` to go down, but with the code in listing 5.14, it wouldn't. You need a way to handle this problem.

A pattern in Domain-Driven Design (DDD) called Root and Aggregates helps you in this situation. In this pattern, the `Book` entity class is the Root, and the `PriceOffer`, `Reviews`, and `AuthorLinks` are Aggregates. (See the principal and dependent descriptions in section 3.1.1.) This pattern goes on to say you should access Aggregates only via the Root. This process works well with soft deletes because if the `Book` (Root) is soft-deleted, you can't access its Aggregates. So the correct code for counting all the `Reviews`, taking the soft delete into account, is

```
var numReviews = context.Books.SelectMany(x => x.Reviews).Count();
```

> **NOTE** Another way to solve the Root/Aggregate problem with soft deletes is to mimic the cascade delete behavior when setting soft deletes, which is quite complex to do. But I have built a library called `EfCore.SoftDeleteServices` that mimics cascade-delete behavior but uses soft deletes; see https://github.com/JonPSmith/EfCore.SoftDeleteServices.

The second thing to consider is that you shouldn't apply soft deletes to a one-to-one relationship. You will have problems if you try to add a new one-to-one entity when an existing but soft-deleted entity is already there. If you had a soft-deleted `PriceOffer`, which has a one-to-one relationship with the `Book`, and tried to add another `PriceOffer` to the `Book`, you would get a database exception. A one-to-one relationship has a unique index on the foreign key `BookId`, and a (soft-deleted) `PriceOffer` was taking that slot.

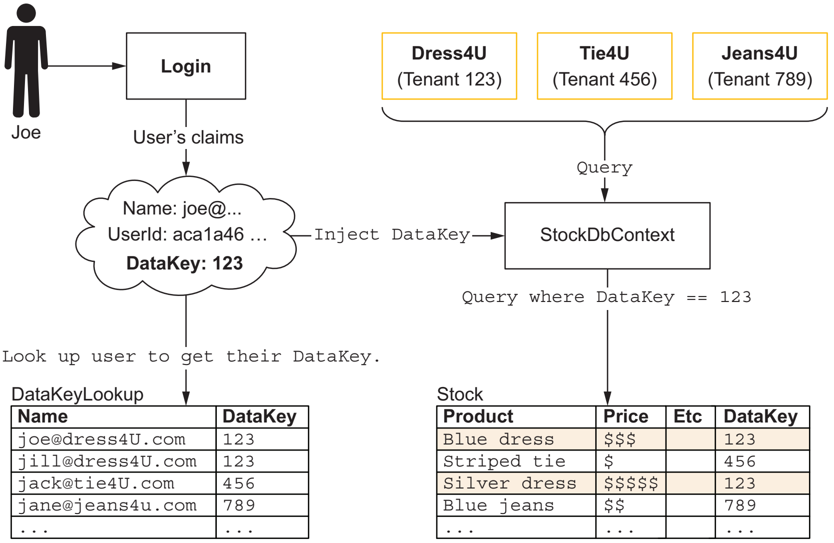
As my clients have found, the soft-delete feature is useful because users can mistakenly delete the wrong data. But being aware of the issues allows you to plan how to handle them in your applications. I usually use the Root/Aggregate approach and don't allow soft deletes of one-to-one dependent entities.

USING QUERY FILTERS TO CREATE MULTITENANT SYSTEMS

A *multitenant system* is one in which different users or groups of users have data that should be accessed only by certain users. You can find many examples, such as Office365 and GitHub. The Query Filters feature isn't enough to build Office365 on its own, but you can use Query Filters to build complex multitenant application.

In the soft-delete use of the Query Filter, you used a Boolean as the filter, but for a multitenant system, you need a more elaborate key, which I refer to as the DataKey. Each tenant has a unique DataKey. A tenant might be an individual user or, more likely, a group of users. Figure 6.5 shows an example Software as a Service (SaaS) application that provides stock control for lots of retail companies. In this case, Joe works for Dress4U and has the DataKey on login.

In the Book App, no one needs to log in, so you can't implement the exact approach shown in figure 6.5, but it does have a basket cookie with a pseudo `UserId` that you can use. When a user selects a book to buy in the Book App, a basket cookie is created



Name	DataKey
joe@dress4U.com	123
jill@dress4U.com	123
jack@tie4U.com	456
jane@jeans4u.com	789
...	...

Product	Price	Etc	DataKey
Blue dress	$$$		123
Striped tie	$		456
Silver dress	$$$$$		123
Blue jeans	$$		789
...	...		...

Figure 6.5 When Joe logs in, his name and `UserId` are looked up in the DataKeyLookup table, and the appropriate DataKey (123) is added to his user claims. When Joe asks for a list of stock, the DataKey from the user's claims is extracted and given to the application's DbContext when it is created. Then the DataKey is used in a Global Query Filter applied to the Stock table. Therefore, Joe sees only the Blue dress and the Silver dress.

to hold each book in the user's basket, plus a `UserId`. This basket cookie is used if the user clicks the My Orders menu item to show only the `Orders` from this user. The following code takes the `UserId` from the basket cookie and uses a Query Filter to return only the `Orders` that the user created. Two main parts make this code work:

- A `UserIdService` gets the `UserId` from the basket cookie.
- The `IUserIdService` is injected via the application's DbContext constructor and used to access the current user.

The following listing shows the `UserIdService` code, which relies on the `IHttpContextAccessor` to access the current HTTP request.

Listing 6.4 `UserIdService` that extracts the `UserId` from the basket cookie

```
public class UserIdService : IUserIdService
{
    private readonly IHttpContextAccessor _httpAccessor;

    public UserIdService(IHttpContextAccessor httpAccessor)
    {
        _httpAccessor = httpAccessor;
    }

    public Guid GetUserId()
    {
        var httpContext = _httpAccessor.HttpContext;
        if (httpContext == null)
            return Guid.Empty;

        var cookie = new BasketCookie(httpContext.Request.Cookies);
        if (!cookie.Exists())
            return Guid.Empty;

        var service = new CheckoutCookieService(cookie.GetValue());
        return service.UserId;
    }
}
```

The IHttpContextAccessor is a way to access the current HTTP context. To use it, you need to register it in the Startup class, using the command services.AddHttpContextAccessor().

In some cases, the HTTPContext could be null, such as a background task. In such a case, you provide an empty GUID.

Uses existing services to look for the basket cookie. If there is no cookie, the code returns an empty GUID.

If there is a basket cookie, creates the CheckoutCookieService, which extracts the UserId and returns it

When you have a value to act as a DataKey, you need to provide it to the application's DbContext. The typical way is via DI constructor injection; the injected service provides a way to get the DataKey. For our example, we are using the `UserId`, taken from the basket cookie, to serve as a DataKey. Then you use that `UserId` in a Query Filter applied to the `CustomerId` property in the `Order` entity class, which contains the `UserId` of the person who created the `Order`. Any query for `Order` entities will return only `Orders` created by the current user. The following listing shows how to inject the `UserIdService` service into the application's DbContext and then use that `UserId` in a Query Filter.

Listing 6.5 Book App's DbContext with injection of `UserId` and Query Filter

```
public class EfCoreContext : DbContext
{
    private readonly Guid _userId;

    public EfCoreContext(DbContextOptions<EfCoreContext> options,
        IUserIdService userIdService = null)
        : base(options)
    {
        _userId = userIdService?.GetUserId()
                   ?? new ReplacementUserIdService().GetUserId();
    }

    public DbSet<Book> Books { get; set; }
    //… rest of DbSet<T> left out

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        //… other configuration left out for clarity

        modelBuilder.Entity<Book>()
            .HasQueryFilter(p => !p.SoftDeleted);
        modelBuilder.Entity<Order>()
            .HasQueryFilter(x => x.CustomerName == _userId);
    }
}
```

This property holds the UserId used in the Query Filter on the Order entity class.

Sets the UserIdService. Note that this parameter is optional, which makes it much easier to use in unit tests that don't use the Query Filter.

Normal options for setting up the application's DbContext

Sets the UserId. If the UserId is null, a simple replacement version provides the default Guid.Empty value.

The method where you configure EF Core and put your Query Filters

Soft-delete Query Filter

Order query filter, which matches the current UserId obtained from the cookie basket with the CustomerId in the Order entity class

To be clear, every instance of the application's DbContext gets the `UserId` of the current user, or an empty GUID if they never "bought" a book. Whereas the DbContext's configuration is set up on first use and cached, the lambda Query Filter is linked to a live field called _userId. The query filter is fixed, but the _userId is dynamic and can change on every instance of the DbContext.

But it's important that the Query Filter not be put in a separate configuration class (see section 7.5.1), because the _userId would become fixed to the `UserId` provided on first use. You must put the lambda query somewhere that it can get the dynamic _userId variable. In this case, I place it in the `OnModelCreating` method inside the application's DbContext, which is fine. In chapter 7, I show you a way to automate the configuration of the Query Filters that keep the _userId dynamic; see section 7.15.4.

If you have an ASP.NET Core application that users log in to, you can use `IHttpContextAccessor` to access the current `ClaimPrincipal`. The `ClaimPrincipal` contains a list of `Claims` for the logged-in user, including their `UserId`, which is stored in a claim with the name defined by the system constant `ClaimTypes.NameIdentifier`. Or, as shown in figure 6.5, you could add a new `Claim` to the user on login to provide a DataKey that is used in the Query Filter.

NOTE For an example of a full multitenant system in which a user's `Id` is used to find a tenant's DataKey at login time and a DataKey `Claim` is added to the user `Claims`, see the article at http://mng.bz/yY7q.

6.1.7 Considering LINQ commands that need special attention

EF Core does a great job of mapping LINQ methods to SQL, the language of most relational databases. But three types of LINQ methods need special handling:

- Some LINQ commands need extra code to make them fit the way that the database works, such as the LINQ `Average`, `Sum`, `Max`, and other aggregate commands needed to handle a return of `null`. Just about the only aggregate that won't return `null` is `Count`.
- Some LINQ commands can work with a database, but only within rigid boundaries because the database doesn't support all the possibilities of the command. An example is the `GroupBy` LINQ command; the database can have only a simple key, and there are significant limitations on the `IGrouping` part.
- Some LINQ commands have a good match to a database feature, but with some limitations on what the database can return. Examples are `Join` and `GroupJoin`.

The EF Core documentation has a great page called Complex Query Operators (see http://mng.bz/MXan) with good descriptions of many of these commands, so I'm not going to go through them all. But I do want to warn you about the feared `InvalidOperationException` exception, with a message containing the words `could not be translated`, and tell you what to do when you get it.

The problem is that if you get your LINQ slightly wrong, you will get the `could not be translated` exception. The message might not be too helpful in diagnosing the problem (but see the following note), other than saying that you should `switch to client evaluation explicitly by inserting a call to AsEnumerable . . .`" Although you could switch to client evaluation, you might take a (big) performance hit.

NOTE The EF Core team is refining the messages returned from a `could not be translated` exception and adding specific messages for common situations, such as trying to use the `String.Equal` method with a `StringComparison` parameter (which can't be converted to SQL).

The following section provides some tips for making the more mainstream complex commands work with a relational database. I also suggest that you test any complex queries, as they are easy to get wrong.

AGGREGATES NEED A NULL (APART FROM COUNT)

You are likely to use the LINQ aggregates `Max`, `Min`, `Sum`, `Average`, `Count`, and `CountLong`, so here are some pointers on what to do to get them to work:

- The `Count` and `CountLong` methods work fine if you count something sensible in the database, such as a row or relational links such as the number of `Reviews` for a `Book`.

- The LINQ aggregates `Max`, `Min`, `Sum`, and `Average` need a nullable result, such as `context.Books.Max(x => (decimal?)x.Price)`. If the source (`Price` in this example) isn't nullable, you must have cast to the nullable version of the source. Also, if you are using Sqlite for unit testing, remember that it doesn't support `decimal`, so you would get an error even if you used the nullable version.
- You can't use the LINQ `Aggregate` method directly on the database because it does a per-row calculation.

GROUPBY LINQ COMMAND

The other LINQ method that can be useful is `GroupBy`. When `GroupBy` is used on an SQL database, the `Key` part needs to be a scalar value (or values) because that's what the SQL `GROUP BY` supports. The `IGrouping` part can be a selection of data, including some LINQ commands. My experience is that you need to follow a `GroupBy` command with an execute command (see section 2.3.3) such as `ToList`. Anything else seems to cause the `could not be translated` exception.

Here is a real example taken from a client's application, with some of the names changed to keep the client's secrets. Notice that the `Key` can be a combination of scalar columns and the `IGrouping` part:

```
var something = await _context.SomeComplexEntity
     .GroupBy(x => new { x.ItemID, x.Item.Name })
     .Select(x => new
      {
           Id = x.Key.ItemID,
           Name = x.Key.Name,
           MaxPrice = x.Max(o => (decimal?)o.Price)
      })
     .ToListAsync();
```

6.1.8 Using AutoMapper to automate building Select queries

In chapter 2, you learned that `Select` queries allow you to build one query that returns exactly the data you need, and these queries are often quite efficient from the performance side too. The problem is that they take a bit more time to write—only a few more lines, but real applications can be thousands of queries, so each `Select` query adds to development time. I'm always looking for ways to automate things, and AutoMapper (https://automapper.org) can help you automate the building of `Select` queries.

I am not going to describe all the features of AutoMapper, which might take a whole book in itself! But I'll give you an overview of how to set up and use AutoMapper, because I don't think those topics are covered well elsewhere. Let's start by comparing a simple `Select` query that is handcoded against an AutoMapper-built `Select` query, as shown in figure 6.6.

Although the example in figure 6.6 is simple, it shows that you can collapse a `Select` query to one line by using AutoMapper's `ProjectTo` method. Figure 6.6 uses AutoMapper's By Convention configuration, where it maps properties in the source—`Book` class, in this case—to the DTO properties by matching them by the type and name of each property. AutoMapper can automatically map some relationships. A property of

Handcoded version

```
var dto = context.Books
   .Select(p => new ChangePubDateDto
   {
      BookId = p.BookId,
      Title = p.Title,
      PublishedOn = p.PublishedOn
   })
   .Single(k => k.BookId == lastBook.BookId);
```

AutoMapper version

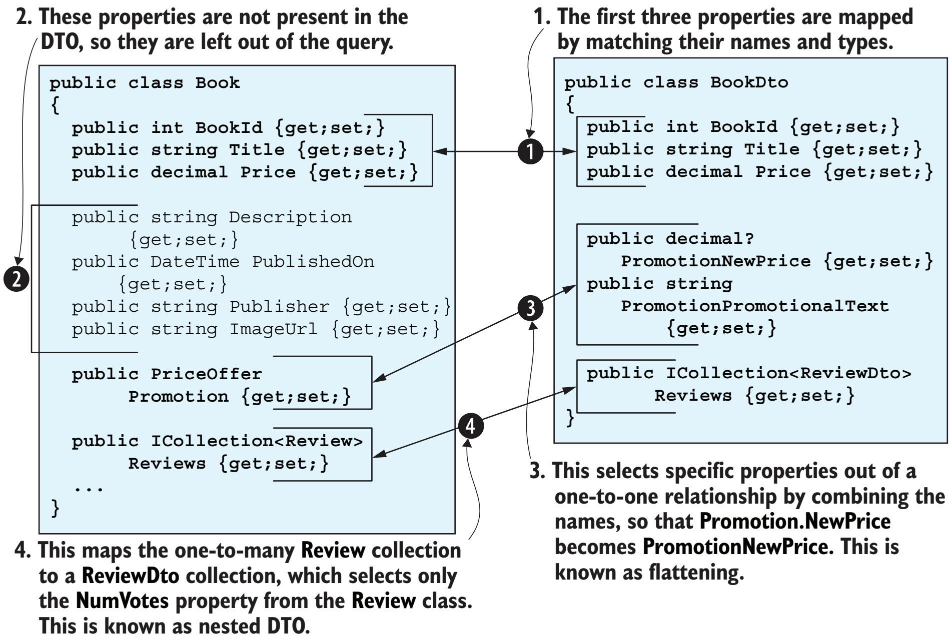
```
var dto = context.Books
   .ProjectTo<ChangePubDateDtoAm>(config)
   .Single(x => x.BookId == lastBook.BookId);
```

Figure 6.6 Both versions of the `Select` query produce the same results and the same SQL code. This query is super simple, with only three properties copied over, but it gives you an idea of how AutoMapper works. In this case, the DTO has properties of the same type and name as the properties we want to copy over, which means AutoMapper will automatically build the LINQ code to copy those three properties.

type `decimal` and called `PromotionNewPrice` would map the Book's `Promotion.NewPrice` relationship, for example. (This AutoMapper feature is called *flattening*; see http://mng.bz/aorB.)

Figure 6.7 shows four by-convention configurations of using AutoMapper:

- *Same type and same name mapping*—Properties are mapped from the entity class to DTO properties by having the same type and same name.



2. These properties are not present in the DTO, so they are left out of the query.

1. The first three properties are mapped by matching their names and types.

```
public class Book
{
  public int BookId {get;set;}
  public string Title {get;set;}
  public decimal Price {get;set;}

  public string Description
       {get;set;}
  public DateTime PublishedOn
      {get;set;}
  public string Publisher {get;set;}
  public string ImageUrl {get;set;}

  public PriceOffer
       Promotion {get;set;}

  public ICollection<Review>
       Reviews {get;set;}
  ...
}
```

```
public class BookDto
{
  public int BookId {get;set;}
  public string Title {get;set;}
  public decimal Price {get;set;}

  public decimal?
     PromotionNewPrice {get;set;}
  public string
     PromotionPromotionalText
         {get;set;}

  public ICollection<ReviewDto>
        Reviews {get;set;}
}
```

3. This selects specific properties out of a one-to-one relationship by combining the names, so that Promotion.NewPrice becomes PromotionNewPrice. This is known as flattening.

4. This maps the one-to-many Review collection to a ReviewDto collection, which selects only the NumVotes property from the Review class. This is known as nested DTO.

Figure 6.7 Four ways that AutoMapper maps the `Book` entity class to the `BookDto` class. The default convention is to map via similar names and types, including handling relationships by having a name equivalent to the property access but without the dot. The DTO property `PromotionNewPrice`, for example, is mapped automatically to the `Promotion.NewPrice` property in the source. Mappings also can be nested; a collection in the entity class can be mapped to a collection with a DTO.

- *Trimming properties*—By leaving out properties that are in the entity class from the DTO, the `Select` query won't load those columns.
- *Flattening relationships*—The name in the DTO is a combination of the navigational property name and the property in the navigational property type. The `Book` entity reference of `Promotion.NewPrice`, for example, is mapped to the DTO's `PromotionNewPrice` property.
- *Nested DTOs*—This configuration allows you to map collections from the entity class to a DTO class, so you can copy specific properties from the entity class in a navigational collection property.

Now that you have an idea of what AutoMapper can do, I want to give you some tips on how to use and configure it.

FOR SIMPLE MAPPINGS, USE THE [AUTOMAP] ATTRIBUTE

Using AutoMapper's `ProjectTo` method is straightforward, but it relies on the configuration of AutoMapper, which is more complex. In release 8.1 of AutoMapper, Jimmy Bogart added the `AutoMap` attribute, which allows by convention configuration of simple mappings. The following code snippet shows the `[AutoMap]` attribute in the first line (in bold), where you define what entity class this DTO should map from:

```
[AutoMap(typeof(Book))]
public class ChangePubDateDtoAm
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public DateTime PublishedOn { get; set; }
}
```

Classes mapped via `AutoMap` attribute use AutoMapper's By Convention configuration, with a few parameters and attributes to allow some tweaking. As you saw in figure 6.7, by convention can do quite a lot, but certainly not all that you might need. For that, you need AutoMapper's `Profile` class.

COMPLEX MAPPINGS NEED A PROFILE CLASS

When AutoMapper's By Convention approach isn't enough, you need to build an AutoMapper `Profile` class, which allows you to define the mapping for properties that aren't covered by the By Convention approach. To map a `Book` to the `BookListDto` described in listings 2.10 and 2.11, for example, three of the nine DTO properties need special handling. You have to create a `MappingConfiguration`. You have a few ways to do this, but typically, you use AutoMapper's `Profile` class, which is easy to find and register. The following listing shows a class that inherits the `Profile` class and sets up the mappings that are too complex for AutoMapper to deduce.

Listing 6.6 AutoMapper `Profile` class configuring special mappings for some properties

```
public class BookListDtoProfile : Profile
{
    public BookListDtoProfile()
```

Your class must inherit the AutoMapper Profile class. You can have multiple classes that inherit Profile.

```
  {
    CreateMap<Book, BookListDto>()
      .ForMember(p => p.ActualPrice,
        m => m.MapFrom(s => s.Promotion == null
          ? s.Price : s.Promotion.NewPrice))
      .ForMember(p => p.AuthorsOrdered,
        m => m.MapFrom(s => string.Join(", ",
            s.AuthorsLink.Select(x => x.Author.Name))))
      .ForMember(p => p.ReviewsAverageVotes,
        m => m.MapFrom(s =>
          s.Reviews.Select(y =>
            (double?)y.NumStars).Average()));
  }
}
```

Sets up the mapping from the Book entity class to the BookListDto

The Actual price depends on whether the Promotion has a PriceOffer.

Gets the list of Author names as a comma-delimited string

Contains the special code needed to make the Average method run in the database

This code sets up three of the nine properties, with the other six properties using AutoMapper's By Convention approach, which is why some of the names of the properties in the `ListBookDto` class are long. The DTO property called `PromotionPromotionalText`, for example, has that name because it maps by convention to the navigational property `Promotion` and then to the `PromotionalText` property in the `PriceOffer` entity class.

You can add lots of `CreateMap` calls in one `Profile`, or you can have multiple `Profiles`. `Profiles` can get complex, and managing them is the main pain point involved in using AutoMapper. One of my clients had a single `Profile` that was 1,000 lines long.

REGISTER AUTOMAPPER CONFIGURATIONS

The last stage is registering all the mapping with dependency injection. Fortunately, AutoMapper has a NuGet package called `AutoMapper.Extensions.Microsoft.DependencyInjection` containing the method `AddAutoMapper`, which scans the assemblies you provide and registers an `IMapper` interface as a service. You use the `IMapper` interface to inject the configuration for all your classes that have the `[AutoMap]` attribute and all the classes that inherit AutoMapper's `Profile` class. In an ASP.NET Core application, the following code snippet would be added to the `Configure` method of the `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    // … other code removed for clarity

    services.AddAutoMapper( MyAssemblyToScan1, MyAssemblyToScan2…);
}
```

6.1.9 Evaluating how EF Core creates an entity class when reading data in

Up until now, the entity classes in this book haven't had user-defined constructors, so if you read in that entity class, EF Core uses the default parameterless constructor and then updates the properties and backing fields directly. (Chapter 7 describes backing fields.) But sometimes, it's useful to have a constructor with parameters, because it makes it easier to create an instance or because you want to make sure that the class is created in the correct way.

> **NOTE** Using constructors to create a class is a good approach, because you can define what parameters you must set to create a valid instance. When you're using the DDD approach with EF Core (see chapter 13), the only way to create an entity class is via some form of constructor or static factory.

Since EF Core 2.1, EF Core has used an entity class's constructor when it needs to create an entity class instance, typically when reading in data. If you use EF Core's By Convention pattern for your constructor—that is, the constructor's parameters match the properties by type and name (with camel/Pascal casing) and don't include navigational properties, as shown in the following listing—EF Core will use it too.

Listing 6.7 An entity class with a constructor that works with EF Core

```
public class ReviewGood
{
    public int Id { get; private set; }
    public string VoterName { get; private set; }
    public int NumStars { get; set; }

    public ReviewGood
         (string voterName)
    {
         VoterName = voterName;
         NumStars = 2;
    }
}
```

You can set your properties to have a private setter. EF Core can still set them.

The constructor doesn't need parameters for all the properties in the class. Also, the constructor can be any type of accessibility: public, private, and so on.

EF Core will look for a parameter with the same type and a name that matches the property (with matching of Pascal/camel case versions of the name).

The assignment should not include any changing of the data; otherwise, you won't get the exact data that was in the database.

Any assignment to a property that doesn't have a parameter is fine. EF Core will set that property after the constructor to the data read back from the database.

I could have added a constructor to the `ReviewGood` class that set all the non-navigational properties, but I wanted to point out that EF Core can use a constructor to create the entity instance and then fill in any properties that weren't in the constructor's parameters. Now, having looked at a constructor that works, let's look at constructors that EF Core can't or won't use and how to handle each problem.

CONSTRUCTORS THAT CAN CAUSE YOU PROBLEMS WITH EF CORE

The first type of constructor that EF Core can't use is one with a parameter whose type or name doesn't match. The following listing shows an example with a parameter called `starRating`, which assigns to the property called `NumStars`. If this constructor is the only one, EF Core will throw an exception the first time you use the application's DbContext.

Listing 6.8 Class with constructor that EF Core can't use, causing an exception

```
public class ReviewBadCtor
{
    public int Id { get; set; }
    public string VoterName { get; set; }
    public int NumStars { get; set; }

    public ReviewBadCtor(
        string voterName,
        int starRating)
    {
        VoterName = voterName;
        NumStars = starRating;
    }
}
```

The only constructor in this class

This parameter's name doesn't match the name of any property in this class, so EF Core can't use it to create an instance of the class when it is reading in data.

Another example of a constructor that EF Core can't use is one with a parameter that sets a navigational property. If the `Book` entity class had a constructor that included a parameter to set the `PriceOffer Promotion` navigational property, for example, EF Core couldn't use it either. A constructor that EF Core can use can have only nonrelational properties.

If your constructor doesn't match EF Core's By Convention pattern, you need to provide a constructor that EF Core can use. The standard solution is to add a private parameterless constructor, which EF Core can use to create the class instance and use its normal parameter/field setting.

> **NOTE** EF Core can use constructors with access modifiers. It uses any level of access from `private` to `public` constructors, for example. As you have already seen, it can also write to a property with a private setter, such as `public int Id {get; private set;}`. EF Core can handle read-only properties (such as instance `public int Id {get;}`), but with some limitations; see http://mng.bz/go2E.

Another, more subtle problem occurs if you alter the parameter data when you assign it to the matching property. The following code snippet would cause problems because the data read in would be altered in the assignment:

```
public ReviewBad(string voterName)
{
    VoterName = "Name: "+voterName; //alter the parameter before assign to
     property
    //… other code left out
}
```

The result of the assignment in the ReviewBad constructor means that if the data in the database was XXX, after the read, it would be Name: XXX, which is not what you want. The solution is to change the name of the parameter so that it doesn't match the property name. In this case, you might call it `voterNameNeedingPrefix`.

Finally, be aware that checks and validations you apply to your parameters in your constructor are going to be applied when EF Core uses the constructor. If you have a test to make sure that a string is not null, then you should configure the database column to be non-null (see chapter 7) to make sure that some rogue data in your database doesn't return a `null` value.

EF CORE CAN INJECT CERTAIN SERVICES VIA THE ENTITY CONSTRUCTOR

While we are talking about entity class constructors, we should look at EF Core's ability to inject some services via the entity class's constructor. EF Core can inject three types of services, the most useful of which injects a method to allow lazy loading of relationships, which I describe in full. The other two uses are advanced features; I summarize what they do and provide a link to the Microsoft EF Core documentation for more information.

In section 2.4.4, you learned how to configure lazy loading of relationships via the `Microsoft.EntityFrameworkCore.Proxies` NuGet package. That package is the simplest way to configure lazy loading, but it has the drawback that all the navigational properties must be set up to use lazy loading—that is, every navigational property must have the keyword `virtual` added to its property definition.

If you want to limit what relationships use lazy loading, you can obtain a lazy loading service via an entity class's constructor. Then you change the navigational properties to use this service in the property's getter method. The following listing shows a `BookLazy` entity class that has two relationships: a `PriceOffer` relationship that doesn't use lazy loading and a `Reviews` relationship that does.

Listing 6.9 Showing how lazy loading works via an injected lazy loader method

```
public class BookLazy
{
    public BookLazy() { }

    private BookLazy(ILazyLoader lazyLoader)
    {
        _lazyLoader = lazyLoader;
    }
    private readonly ILazyLoader _lazyLoader;

    public int Id { get; set; }

    public PriceOffer Promotion { get; set; }

    private ICollection<LazyReview> _reviews;
    public ICollection<LazyReview> Reviews
    {
        get => _lazyLoader.Load(this, ref _reviews);
        set => _reviews = value;
    }
}
```

You need a public constructor so that you can create this book in your code.

This private constructor is used by EF Core to inject the LazyLoader.

A normal relational link that isn't loaded via lazy loading

The actual reviews are held in a backing field (see section 8.7).

The list that you will access

A read of the property will trigger a lazy loading of the data (if not already loaded).

The set simply updates the backing field.

Injecting the service via the `ILazyLoader` interface requires the NuGet package `Microsoft.EntityFrameworkCore.Abstractions` to be added to the project. This package has a minimal set of types and no dependencies, so it doesn't "pollute" the project with references to the DbContext and other data-access types.

But if you are enforcing an architecture that doesn't allow any external packages in it, you can add a parameter by using the type `Action<object, string>` in the entity's constructor. EF Core will fill the parameter of type `Action<object, string>` with an action that takes the entity instance as its first parameter and the name of the field as the second parameter. When this action is invoked, it loads the relationship data into the named field in the given entity class instance.

> **NOTE** By providing a small extension method, you can make the `Action <object, string>` option work similarly to `ILazyLoader`. You can see this effect in the extension method at the end of the "Lazy loading without proxies" section of the EF Core documentation page at http://mng.bz/e5Zv in the class `LazyBook2` in the `Test` project in the GitHub repo associated with this book.

The other two ways of injecting a service into the entity class via a constructor are as follows:

- Injecting the DbContext instance that the entity class is linked to is useful if you want to run database accesses inside your entity class. In chapter 13, I cover the pros and cons of executing database accesses inside your entity class. In a nutshell, you shouldn't use this technique unless you have a serious performance or business logic problem that can't be solved any other way.
- The `IEntityType` for this entity class instance gives you access to the configuration, `State`, EF Core information about this entity, and so on associated with this entity type.

These two techniques are advanced features, and I won't cover these in detail. The EF Core documentation on entity class constructors has more information on this topic; see http://mng.bz/pV78.

6.2 Writing to the database with EF Core

The first part of this chapter was about querying the database. Now you'll turn your mind to writing to the database: creating, updating, and deleting entity classes. As in section 6.1, the aim is to expose you to how EF Core works inside when writing to the database. Some subsections of section 6.1 are about learning what is happening when you write to the database, and some are neat techniques for copying or deleting data quickly. Here is the list of topics that I will cover:

- Evaluating how EF Core writes entities with relationships to the database
- Evaluating how DbContext handles writing out entities with relationships
- Copying data with relationships quickly
- Deleting an entity quickly

6.2.1 *Evaluating how EF Core writes entities/relationships to the database*

When you are creating a new entity with new relationship(s), navigational properties are your friends because EF Core takes on the problem of filling the foreign key for you. The next listing shows a simple example: adding a new `Book` that has a new `Review`.

Listing 6.10 Adding a new `Book` entity with a new `Review`

```
var book = new Book
{
    Title = "Test",
    Reviews = new List<Review>()
};
book.Reviews.Add(
    new Review { NumStars = 1 });
context.Add(book);
context.SaveChanges();
```

Creates a new Book

Adds a new Review to the Book's Reviews navigational property

The Add method says that the entity instance should be added to the appropriate row, with any relationships added or updated.

SaveChanges carries out the database update.

To add these two linked entities to the database, EF Core has to do the following:

- *Work out the order in which it should create these new rows*—In this case, it has to create a row in the Books table so that it has the primary key of the `Book`.
- *Copy any primary keys into the foreign key of any relationships*—In this case, it copies the Books row's primary key, `BookId`, into the foreign key in the new Review row.
- *Copy back any new data created in the database so that the entity classes properly represent the database*—In this case, it must copy back the `BookId` and update the `BookId` property in both the `Book` and `Review` entity classes and the `ReviewId` for the `Review` entity class.

The following listing shows the SQL for this create.

Listing 6.11 The SQL commands to create the two rows, with return of primary keys

Because EF Core wants to return the primary key, it turns off the return of the database changes.

```
-- first database access
SET NOCOUNT ON;
INSERT INTO [Books] ([Description], [Title], ...)
VALUES (@p0, @p1, @p2, @p3, @p4, @p5, @p6);

SELECT [BookId] FROM [Books]
WHERE @@ROWCOUNT = 1 AND [BookId] = scope_identity();

-- second database access
SET NOCOUNT ON;
INSERT INTO [Review] ([BookId], [Comment], ...)
VALUES (@p7, @p8, @p9, @p10);
```

Inserts a new row into the Books table. The database generates the Book's primary key.

Returns the primary key, with checks to ensure that the new row was added

Inserts a new row into the Review table. The database generates the Review's primary key.

```
SELECT [ReviewId] FROM [Review]
WHERE @@ROWCOUNT = 1 AND [ReviewId] = scope_identity();
```

Returns the primary key, with checks to ensure that the new row was added

This example is simple, but it covers all the main parts. What you need to understand is that you can create complex data with relationships and relationships of those relationships, and EF Core will work out how to add them to the database.

I have seen EF Core code in which the developer used multiple calls to the `SaveChanges` method to obtain the primary key from the first create to set the foreign key for the related entity. You don't need to do that if you have navigational properties that link the different entities. So if you think that you need to call `SaveChanges` twice, normally you haven't set up the right navigational properties to handle that case.

> **WARNING** Calling `SaveChanges` multiple times to create an entity with relationships isn't recommended because if the second `SaveChanges` fails for some reason, you have an incomplete set of data in your database, which could cause problems. See the sidebar called "Why you should call SaveChanges only once at the end of your changes" in section 3.2.2 for more information.

6.2.2 Evaluating how DbContext handles writing out entities/relationships

In section 6.2.1, you saw what EF Core does at the database end, but now we are going to look at what happens inside EF Core. Most of the time, you don't need this information, but sometimes, knowing it is important. If you are catching changes during a call to `SaveChanges`, for example, you get its `State` only before `SaveChanges` is called, but you have the primary key of a newly created entity only after the call to `SaveChanges`.

> **NOTE** I bumped into the before/after `SaveChanges` problem when I wrote the first edition of this book. I needed to detect changes to a `Book` entity class and changes to any of its related entity classes, such as `Review`, `BookAuthor`, and `PriceOffer`. At that point, I needed to catch the `State` of each entity at the start, but I might not have had the right foreign key until `SaveChanges` had finished.

Even if you aren't trying something as complex as the before/after `SaveChanges` issue, it is good to understand how EF Core works. This example is a little more complex than the last one because I want to show you the different ways that EF Core handles new instances of an entity class over an instance of an entity that has been read from the database. The code in the next listing creates a new `Book`, but with an `Author` that is already in the database. The code has comments `STAGE 1`, `STAGE 2`, and `STAGE 3`, and I describe what happens after each stage.

Listing 6.12 Creating a new `Book` with a new many-to-many link to an existing `Author`

```
//STAGE1
var author = context.Authors.First();
var bookAuthor = new BookAuthor { Author = author };
var book = new Book
{
    Title = "Test Book",
    AuthorsLink = new List<BookAuthor> { bookAuthor }
};

//STAGE2
context.Add(book);

//STAGE3
context.SaveChanges();
```

Each of the three stages starts with a comment.

Reads in an existing Author for the new book

Creates a new BookAuthor linking row, ready to link to Book to the Author

Creates a Book and fills in the AuthorsLink navigational property with a single entry, linking it to the existing Author

Calls the Add method, which tells EF Core that the Book needs to be added to the database

SaveChanges looks at all the tracked entities and works out how to update the database to achieve what you have asked it to do.

Figures 6.8, 6.9, and 6.10 show you what is happening inside the entity classes and their tracked data at each stage. Each of the three figures shows the following data at the end of its stage:

- The `State` of each entity instance at each stage of the process (shown above each entity class)
- The primary and foreign keys with the current value in brackets. If a key is `(0)`, it hasn't been set yet.
- The navigational links are shown as connections from the navigational property to the appropriate entity class that it is linked to.
- Changes between each stage, shown by bold text or thicker lines for the navigational links.

Figure 6.8 shows the situation after Stage 1 has finished. This initial code sets up a new `Book` entity class (left) with a new `BookAuthor` entity class (middle) that links the Book to an existing `Author` entity class (right).

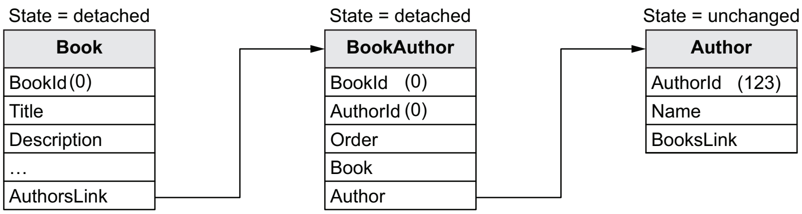


Figure 6.8 End of stage 1. This figure shows that the new `Book` with a new `BookAuthor` linking to the `Book` has a `State` of `Detached`, and the existing `Author`, which was read in from the database, has a `State` of `Unchanged`. The figure also shows the two navigational links that the code set up to link the `Book` entity to the `Author` entity. Finally, the primary and foreign keys of the `Book` and `BookAuthor` are unset—that is, zero—whereas the `Author` entity has an existing primary key (123) because it is already in the database.

Figure 6.8 is a pictorial version of the three entity classes after Stage 1 has finished in listing 6.12. This figure is the starting point before you call any EF Core methods. Figure 6.9 shows the situation after the line `context.Add(book)` is executed. The changes are shown in bold and with thick lines for the added navigational links.

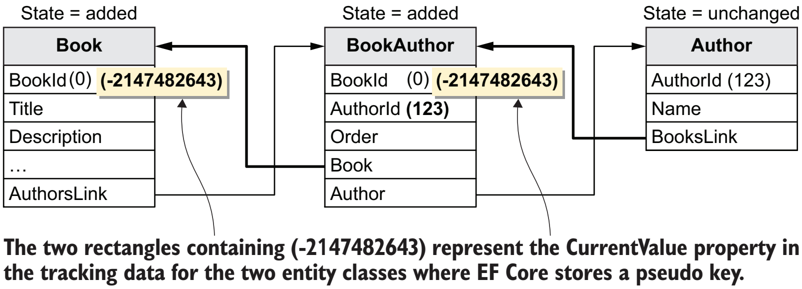


Figure 6.9 End of Stage 2. Lots of things have happened here. The `State` of the two new entities, `Book` and `BookAuthor`, has changed to `Added`. At the same time, the `Add` method tries to set up the foreign keys: It knows the `Author`'s primary key, so it can set the `AuthorId` in the `BookAuthor entity`. It doesn't know the `Book`'s primary key (`BookId`), so it puts a unique negative number in the hidden tracking values, acting as a pseudo key. The `Add` also has a relational fixup stage that fills in any other navigational properties.

You may be surprised by how much happened when the `Add` method was executed. (I was!) It seems to be getting the entities as close as possible to the positions they will be after `SaveChanges` is called. Here are the things that happen when the `Add` method is called in Stage 2.

The `Add` method sets the `State` of the entity provided as a parameter to `Added`—in this example, the `Book` entity. Then it looks at all entities linked to the entity provided as a parameter, either by navigational properties or by foreign-key values. For each linked entity, it does the following:

- If the entity is not tracked—that is, its current `State` is `Detached`—it sets its `State` to `Added`. In this example, that entity is `BookAuthor`. The `Author`'s `State` isn't updated because that entity is tracked.
- It fills in any foreign keys for the correct primary keys. If the linked primary key isn't yet available, it puts a unique negative number in the `CurrentValue` properties of the tracking data for the primary key and the foreign key, as you see in figure 6.9.
- It fills in any navigational properties that aren't currently set up by running a version of the relational fixup described in section 6.1.1. The relationships are shown as thick lines in figure 6.9.

In this example, the only entities to link to are set by your code, but `Add`'s relational fixup stage can link to any tracked entity. The call to the `Add` method can take some time to execute if you have a lot of relationships and/or lots of tracked entity classes in the current DbContext. I cover this performance issue in detail in chapter 14.

The final stage, Stage 3, is what happens when the `SaveChanges` method is called, as shown in figure 6.10.

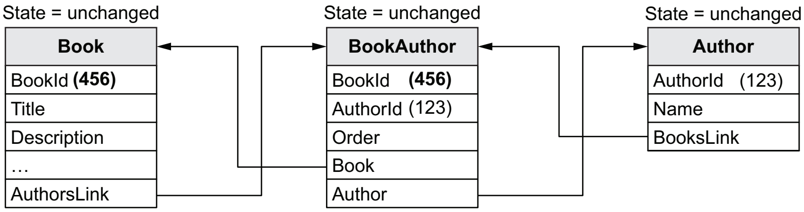


Figure 6.10 End of Stage 3. After `SaveChanges` has finished, the `Book` and `BookAuthor` entities have been added to the database: two new rows have been inserted into the Books and BookAuthors tables. Creating the Book row means that its primary key is generated by the database, which is copied back into the `Book`'s `BookId` and also into the `BookAuthor`'s `BookId` foreign key. On return, the `State` of the `Book` and `BookAuthor` are set to `Unchanged`.

You saw in section 6.2.1 that any columns set or changed by the database are copied back into the entity class so that the entity matches the database. In this example, the `Book`'s `BookId` and the `BookAuthor`'s `BookId` were updated to have the key value created in the database. Also, now that all the entities involved in this database write match the database, their `States` are set to `Unchanged`.

That example may have seemed to be a long explanation of something that "just works," and many times, you don't need to know why. But when something doesn't work correctly, or when you want to do something complex, such as logging entity class changes, this information is useful.

Which wins if they are different: navigational links or foreign key values?

I stated in Stage 2 of section 6.2.2 that the `add` method "looks at all entities linked to the entity provided as a parameter, either by navigational properties or by foreign key values." Which wins if a navigational link links to one entity and the foreign key links to a different entity? My tests say that the navigational link wins. But that result is not defined in the EF Core documentation. I have asked for clarification (see https://github.com/dotnet/efcore/issues/21105), but until there is an answer to this issue, you must test your code to ensure the "navigational properties win over foreign key values" feature hasn't changed.

6.2.3 A quick way to copy data with relationships

Sometimes, you want to copy an entity class with all its relationships. One of my clients needed different versions of a custom-designed structure to send to a customer so they could pick the version they liked. These designs had many common parts, and the designers didn't want to type that data for each design; they wanted to build the first design and copy it as a starting point for the next design.

One solution would be to clone each entity class and its relationships, but that's hard work. (My client's designs could have hundreds of items, each with ~25 relationships.) But knowing how EF Core works allowed me to write code to copy a design by using EF Core itself.

As an example, you are going to use your knowledge of EF Core to copy a user's Book App `Order`, which has a collection of `LineItems`, which in turn links to `Books`. You want to copy the `Order` only with the `LineItems`, but you do *not* want to copy the `Books` that the `LineItems` links to; two copies of a `Book` would cause all sorts of problems. Let's start by looking at the `Order` that we want to copy, shown in the following listing.

Listing 6.13 Creating an `Order` with two `LineItems` ready to be copied

For this test, add four books to use as test data.

Creates an Order with two LineItems to copy

Sets CustomerId to the default value so that the query filter reads the order back

```
var books = context.SeedDatabaseFourBooks();
var order = new Order
{
    CustomerId = Guid.Empty,
    LineItems = new List<LineItem>
    {
        new LineItem
        {
            LineNum = 1, ChosenBook = books[0], NumBooks = 1
        },
        new LineItem
        {
            LineNum = 2, ChosenBook = books[1], NumBooks = 2
        },
    }
};
context.Add(order);
context.SaveChanges();
```

Adds the first LineNum linked to the first book

Adds the second LineNum linked to the second book

Writes this Order to the database

To copy that `Order` properly, you need to know three things (and you know the first two from section 6.2.2):

- If you `Add` an entity that has linked entities that are not tracked—that is, with a `State` of `Detached`—they will be set to the `State Added`.
- EF Core can find linked entities via the navigational links.
- If you try to `Add` an entity class to the database, and the primary key is already in the database, you will get a database exception because the primary key must be unique.

When you know those three things, you can get EF Core to copy the `Order` with its `LineItems`, but not the `Books` that the `LineItems` link to. Here is the code that copies the `Order` and its `LineItems` but doesn't copy the `Book` linked to the `LineItems`.

Listing 6.14 Copying an `Order` with its `LineItems`

```
var order = context.Orders
    .AsNoTracking()
    .Include(x => x.LineItems)

    .Single(x => x.OrderId == id);

order.OrderId = default;
order.LineItems.First().LineItemId = default;
order.LineItems.Last().LineItemId = default;
context.Add(order);
context.SaveChanges();
```

This code is going to query the Orders table.

AsNoTracking means that the entities are read-only; their State will be Detached.

Include the LineItems, as you want to copy them too.

You do not add .ThenInclude(x => x.ChosenBook) to the query. If you did, the query would copy the Book entities, which is not what you want.

Takes the Order that you want to copy

Resets the primary keys (Order and LineItem) to their default value, telling the database to generate new primary keys

Writes out the order and creates a copy

Note that you haven't reset the foreign keys because you are relying on the fact that the navigational properties override any foreign key values. (See the earlier sidebar "Which wins if they are different: navigational links or foreign key values?") But because you are careful, you build a unit test to check that the relationships are copied properly.

6.2.4 A quick way to delete an entity

Now you can copy an entity with its relationships. What about deleting an entity quickly? It turns out that there is a quick way to delete an entity that works well for a disconnected state delete when you're working with a web application.

Chapter 3 covered deleting an entity by reading in the entity you want to delete and then calling EF Core's `Remove` method with that entity instance. That approach works, but it requires two database accesses—one to read in the entity you want to delete and another when the `SaveChanges` is called to delete the entity. As it turns out, though, all that the `Remove` method needs is the appropriate entity class with its primary key(s) set. The following listing shows the deletion of a `Book` entity by providing the `Book`'s primary key value, `BookId`.

Listing 6.15 Deleting an entity from the database by setting its primary key

```
var book = new Book
{
    BookId = bookId
};
context.Remove(book);
context.SaveChanges();
```

Creates the entity class that you want to delete (in this case, a Book)

Sets the primary key of the entity instance

The call to Remove tells EF Core that you want this entity/row to be deleted.

SaveChanges sends the command to the database to delete that row.

In a disconnected situation, such as some form of web application, the command to delete returns only the type and primary key value(s), making the delete code simpler and quicker. Some minor things are different from the read/remove approach to relationships:

- If there is no row for the primary key you gave, EF Core throws a `DbUpdateConcurrencyException`, saying that nothing was deleted.
- The database is in command of which other linked entities are deleted; EF Core has no say in that. (See the discussion of `OnDelete` in chapter 8 for more information.)

Summary

- When reading in entity classes as tracked entities, EF Core uses a process called relational fixup that sets up all the navigational properties to any other tracked entities.
- The normal tracking query uses identity resolution, producing the best representation of the database structure with one entity class instance for each unique primary key.
- The `AsNoTracking` query is quicker than a normal tracking query because it doesn't use identity resolution, but it can create duplicate entity classes with the same data.
- If your query loads multiple collections of relationships by using the `Include` method, it creates one big database query, which can be slow in some circumstances.
- If your query is missing an `Include` method, you will get the wrong result, but there is a way to set up your navigational collections so that your code will fail instead of returning incorrect data.
- Using Global Query Filters to implement a soft-delete feature works well, but watch how you handle relationships that rely on the soft-deleted entity.
- Select queries are efficient from the database side but can take more lines of code to write. The AutoMapper library can automate the building of `Select` queries.
- EF Core creates an entity class when reading in data. It does this via the default parameterless constructor or any other constructors you write if you follow the normal pattern.
- When EF Core creates an entity in the database, it reads back any data generated by the database, such as a primary key provided by the database, so that it can update the entity class instance to match the database.

Part 2

Entity Framework in depth

Part 1 showed how you might build an application by using EF Core. Part 2 covers how to configure EF Core exactly the way you need it and the different ways you can change (EF Core term: *migrate*) your database. It also introduces you to advanced features that can make your software more efficient in both development and performance terms. Part 2 is more of a reference section that covers each part of EF Core in detail, but (I hope) not in a boring way.

Chapter 7 introduces the way that EF Core configures itself when it's first used so that you know where and how to apply any of your own EF Core configurations. The chapter focuses on nonrelational properties, with types such as `int`, `string`, and `DateTime`.

Chapter 8 shows how EF Core finds and configures relationships. EF Core does a good job of configuring most relationships for you, but it does need help on some, and you'll want to configure others because EF Core's default settings don't suit your needs.

Chapter 9 covers the important issue of matching the database to your EF Core configuration, with either the software or the database SQL in control. It deals with different ways to safely alter—that is, migrate—a database as your application evolves.

Chapter 10 covers more-advanced configurable features, such as defining computed columns in your database and catching and handling concurrent updates of the database. You'll use these features only in certain circumstances, but you should know that they're there in case you need them.

Chapter 11 looks at methods inside the EF Core's DbContext class, especially how `SaveChanges` works out what to write to the database and how you can influence that. This chapter covers other diverse topics, such as raw SQL access to the database, database connection resiliency, and the DbContext's `Model` property.

7

Configuring nonrelational properties

This chapter covers

- Configuring EF Core three ways
- Focusing on nonrelational properties
- Defining the database structure
- Introducing value converters, shadow properties, and backing fields
- Deciding which type of configuration works best in different situations

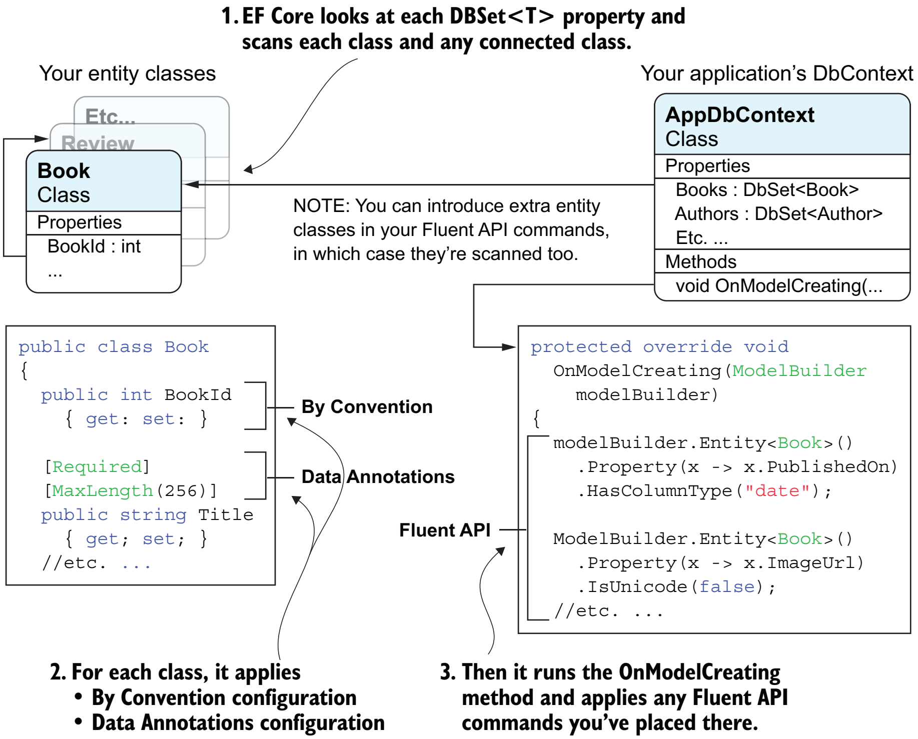
This chapter introduces configuring EF Core in general but concentrates on configuring the nonrelational properties in an entity class; these properties are known as *scalar properties*. Chapter 8 covers configuring relational properties, and chapter 10 covers configuring more-advanced features, such as DbFunctions, computed columns, and so on.

This chapter starts with an overview of the configuration process that EF Core runs when the application's DbContext is used for the first time. Then you'll learn how to configure the mapping between the .NET classes and their associated database tables, with features such as setting the name, SQL type, and nullability of the columns in a table.

This chapter also introduces three EF Core features—*value converters*, *shadow properties*, and *backing fields*—that enable you to control how the data is stored and controlled by the rest of your non-EF Core code. Value converters, for example, allow you to transform data when it is written/read from the database, allowing you to make the database representation easier to understand and debug; shadow properties and backing fields allow you to "hide," or control access to, database data at the software level. These features can help you write better, less fragile applications that are easier to debug and refactor.

7.1 Three ways of configuring EF Core

Chapter 1 covered how EF Core models the database and presented a figure to show what EF Core is doing, with the focus on the database. Figure 7.1 has a more detailed depiction of the configuration process that happens the first time you use the application's DbContext. This figure shows the entire process, with the three configuration approaches: By Convention, Data Annotations, and the Fluent API. This example focuses on the configuration of scalar properties, but the process is the same for all configurations of EF Core.



NOTE: You can introduce extra entity classes in your Fluent API commands, in which case they're scanned too.

```
public class Book
{
  public int BookId
    { get: set: }

  [Required]
  [MaxLength(256)]
  public string Title
    { get; set; }
  //etc. ...
```

```
protected override void
  OnModelCreating(ModelBuilder
    modelBuilder)
{
  modelBuilder.Entity<Book>()
    .Property(x -> x.PublishedOn)
    .HasColumnType("date");

  ModelBuilder.Entity<Book>()
    .Property(x -> x.ImageUrl)
    .IsUnicode(false);
  //etc. ...
```

Figure 7.1 When the application's DbContext is first used, EF Core sets off a process to configure itself and build a model of the database it's supposed to access. You can use three approaches to configure EF Core: By Convention, Data Annotations, and Fluent API. Most real applications need a mixture of all three approaches to configure EF Core in exactly the way your application needs.

This list summarizes the three approaches to configuring EF Core:

- *By Convention*—When you follow simple rules on property types and names, EF Core will autoconfigure many of the software and database features. The By Convention approach is quick and easy, but it can't handle every eventuality.
- *Data Annotations*—A range of .NET attributes known as *Data Annotations* can be added to entity classes and/or properties to provide extra configuration information. These attributes can also be useful for data validation, covered in chapter 4.
- *Fluent API*—EF Core has a method called `OnModelCreating` that's run when the EF context is first used. You can override this method and add commands, known as the *Fluent API*, to provide extra information to EF Core in its modeling stage. The Fluent API is the most comprehensive form of configuration information, and some features are available only via that API.

NOTE Most real applications need to use all three approaches to configure EF Core and the database in exactly the way they need. Some configuration features are available via two or even all three approaches (such as defining the primary key in an entity class). Section 7.16 gives you my recommendations on which approach to use for certain features, plus a way to automate some of your configurations.

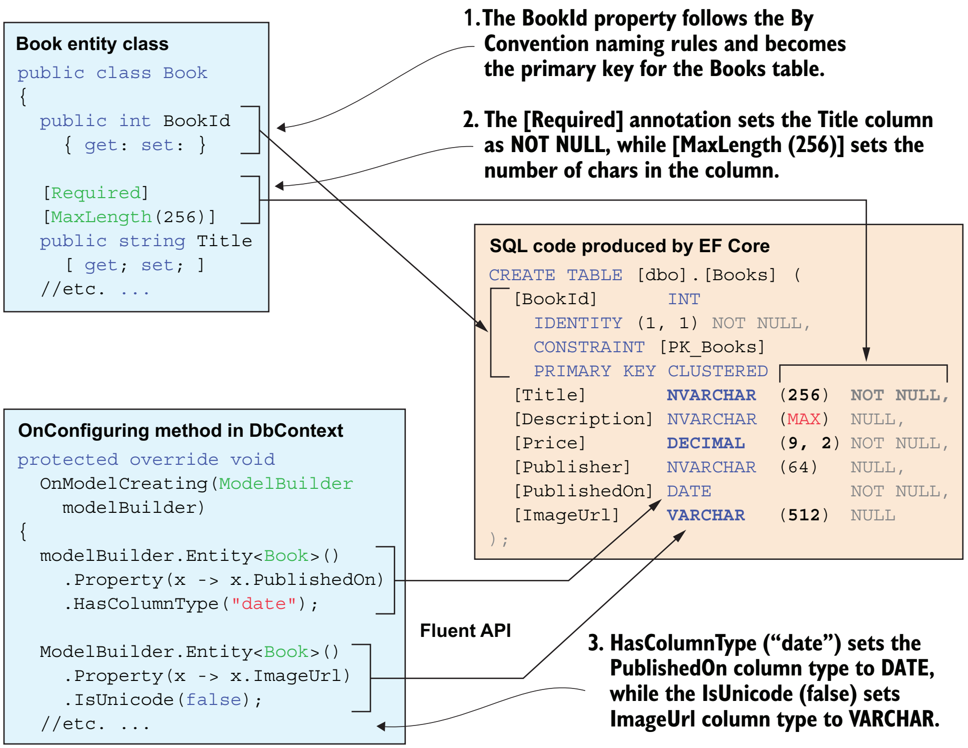
7.2 A worked example of configuring EF Core

For anything beyond a Hello World version of using EF Core, you're likely to need some form of Data Annotations or Fluent API configuration. In part 1, you needed to set up the key for the many-to-many link table. In this chapter, you'll see an example of applying the three configuration approaches introduced in section 7.1 to better match the database to the needs of our Book App.

In this example, you're going to remodel the `Book` entity class used in chapters 2–5 and change the size and type of some of the columns from the defaults that EF Core uses via a EF Core migration. These changes make your database smaller, make sorting or searching on some columns faster, and check that some columns aren't `null`. It's always good practice to define the correct size, type, and nullability for your database columns based on the business needs.

To do this, you'll use a combination of all three configuration approaches. The By Convention configuration has a major part to play, as it defines the table and column names, but you'll add specific Data Annotations and Fluent API configuration methods to change a few of the columns from the default By Convention settings. Figure 7.2 shows how each configuration approach affects EF Core's internal model of database table structure. Because of space limitations, the figure doesn't show all the Data Annotations and Fluent API configuration methods applied to the table, but you can see them in listings 7.1 and 7.2, respectively.

NOTE Figure 7.2 uses arrows to link different EF Core configuration code to the parts of the database table's columns. To be completely clear, changing



Book entity class

```
public class Book
{
  public int BookId
    { get: set: }

   [Required]
   [MaxLength(256)]
  public string Title
    [ get; set; ]
  //etc. ...
```

1. The BookId property follows the By Convention naming rules and becomes the primary key for the Books table.

2. The [Required] annotation sets the Title column as NOT NULL, while [MaxLength (256)] sets the number of chars in the column.

SQL code produced by EF Core

```
CREATE TABLE [dbo].[Books] (
  [BookId]        INT
    IDENTITY (1, 1) NOT NULL,
    CONSTRAINT [PK_Books]
    PRIMARY KEY CLUSTERED
  [Title]         NVARCHAR  (256)  NOT NULL,
  [Description]   NVARCHAR  (MAX)  NULL,
  [Price]         DECIMAL   (9, 2) NOT NULL,
  [Publisher]     NVARCHAR  (64)   NULL,
  [PublishedOn]   DATE             NOT NULL,
  [ImageUrl]      VARCHAR   (512)  NULL
);
```

OnConfiguring method in DbContext

```
protected override void
  OnModelCreating(ModelBuilder
    modelBuilder)
{
  modelBuilder.Entity<Book>()
    .Property(x -> x.PublishedOn)
    .HasColumnType("date");

  ModelBuilder.Entity<Book>()
    .Property(x -> x.ImageUrl)
    .IsUnicode(false);
  //etc. ...
```

3. HasColumnType ("date") sets the PublishedOn column type to DATE, while the IsUnicode (false) sets ImageUrl column type to VARCHAR.

Figure 7.2 To configure the Books table in the exact format you want, you must use all three configuration approaches. A large part is done with By Convention (all the parts not in bold), but then you use Data Annotations to set the size and nullability of the Title column and the Fluent API to change the type of the `PublishedOn` and `ImageUrl` columns.

> EF Core configurations doesn't magically change the database. Chapter 9, which is about changing the database structure (known as the schema) covers several ways in which the EF Core configurations alter the database or the database alters the EF Core configurations in your code.

You will see more detailed explanations of these settings as you read this chapter, but this part gives you an overall view of different ways you can configure your application's DbContext. It's also interesting to think about how some of these configurations could be useful in your own projects. Here are a few EF Core configurations that I use in most projects I work on:

- `[Required]` *attribute*—This attribute tells EF Core that the Title column can't be SQL NULL, which means that the database will return an error if you try to insert/update a book with a `null` `Title` property.
- `[MaxLength(256)]` *attribute*—This attribute tells EF Core that the number of characters stored in the database should 256 rather than defaulting to the database's

maximum size (2 GB in SQL Server). Having fixed-length strings of the right type, 2-byte Unicode or 1-byte ASCII, makes the database access slightly more efficient and allows an SQL index to be applied to these fixed-size columns.

DEFINITION An *SQL index* is a feature that improves the performance of sorting and searching. Section 7.10 covers this topic in more detail.

- `HasColumnType("date")` *Fluent API*—By making the PublishedOn column hold only the date (which is all you need) rather than the default `datetime2`, you reduce the column size from 8 bytes to 3 bytes, which makes searching and sorting on the `PublishedOn` column faster.
- `IsUnicode(false)` *Fluent API*—The `ImageUrl` property contains only 8-bit ASCII characters, so you tell EF Core so, which means that the string will be stored that way. So if the `ImageUrl` property has a `[MaxLength(512)]` attribute (as shown in listing 7.1), the `IsUnicode(false)method` would reduce the size of the ImageUrl column from 1024 bytes (Unicode takes 2 bytes per character) to 512 bytes (ASCII takes 1 byte per character).

This listing shows you the updated `Book` entity class code, with the new Data Annotations in bold. (The Fluent API commands are described in section 7.5.)

Listing 7.1 The `Book` entity class with added Data Annotations

```
public class Book
{
    public int BookId { get; set; }

    [Required]
    [MaxLength(256)]
    public string Title { get; set; }
    public string Description { get; set; }
    public DateTime PublishedOn { get; set; }
    [MaxLength(64)]
    public string Publisher { get; set; }
    public decimal Price { get; set; }

    [MaxLength(512)]
    public string ImageUrl { get; set; }
    public bool SoftDeleted { get; set; }

    //-----------------------------------------------
    //relationships

    public PriceOffer Promotion { get; set; }
    public IList<Review> Reviews { get; set; }
    public IList<BookAuthor> AuthorsLink { get; set; }
}
```

Tells EF Core that the string is non-nullable

Defines the size of the string column in the database

TIP You'd normally set the size parameter in the `[MaxLength(nn)]` attribute by using a constant so that if you create a DTO, it will use the same constant. If you change the size of one property, you change all the associated properties.

Now that you've seen an example that uses all three configuration approaches, let's explore each approach in detail.

7.3 Configuring by convention

By Convention is the default configuration, which can be overridden by the other two approaches, Data Annotations and the Fluent API. The By Convention approach relies on the developer to use the By Convention naming standards and type mappings, which allow EF Core to find and configure entity classes and their relationships, as well as define much of the database model. This approach provides a quick way to configure much of your database mapping, so it's worth learning.

7.3.1 Conventions for entity classes

Classes that EF Core maps to the database are called *entity classes*. As stated in chapter 2, entity classes are normal .NET classes, sometimes referred to as POCOs (plain old CLR objects). EF Core requires entity classes to have the following features:

- The class must be of public access: the keyword `public` should be before the class.
- The class can't be a `static` class, as EF Core must be able to create a new instance of the class.
- The class must have a constructor that EF Core can use. The default, parameterless constructor works, and other constructors with parameters can work. See section 6.1.10 for the detailed rules on how EF Core uses constructors.

7.3.2 Conventions for parameters in an entity class

By convention, EF Core will look for `public` properties in an entity class that have a `public` getter and a setter of any access mode (`public`, `internal`, `protected`, or `private`). The typical, all-public property is

```
public int MyProp { get; set; }
```

Although the all-public property is the norm, in some places having a property with a more localized access setting (such as `public int MyProp { get; private set; }`) gives you more control of how it's set. One example would be a method in the entity class that also does some checks before setting the property; see chapter 13 for more information.

NOTE EF Core can handle read-only properties—properties with only a getter, such as `public int MyProp { get; }`. But in that case, the By Convention approach won't work; you need to use Fluent API to tell EF Core that those properties are mapped to the database.

7.3.3 Conventions for name, type, and size

Here are the rules for the name, type, and size of a relational column:

- The name of the property is used as the name of the column in the table.
- The .NET type is translated by the database provider to the corresponding SQL type. Many basic .NET types have a one-to-one mapping to a corresponding database type. These basic .NET types are mostly .NET *primitive* types (`int`, `bool`, and so on), with some special cases (such as `string`, `DateTime`, and `Guid`).
- The size is defined by the .NET type; for instance, the 32-bit `int` type is stored in the corresponding SQL's 32-bit INT type. `String` and `byte[]` types take on a size of max, which will be different for each database type.

EF6 One change in the default mapping conventions is that EF Core maps a .NET `DateTime` type to SQL `datetime2(7)`, whereas EF6 maps .NET `DateTime` to SQL `datetime`. Microsoft recommends using `datetime2(7)` because it follows the ANSI and ISO SQL standard. Also, `datetime2(7)` is more accurate: SQL `datetime's` resolution is about 0.004 seconds, whereas `datetime2(7)` has a resolution of 100 nanoseconds.

7.3.4 By convention, the nullability of a property is based on .NET type

In relational databases, `NULL` represents missing or unknown data. Whether a column can be `NULL` is defined by the .NET type:

- If the type is `string`, the column can be `NULL`, because a string can be null.
- Primitive types (such as `int`) or struct types (such as `DateTime`) are non-null by default.
- Primitive or struct types can be made nullable by using either the `?` suffix (such as `int?`) or the generic `Nullable<T>` (such as `Nullable<int>`). In these cases, the column can be `NULL`.

Figure 7.3 shows the name, type, size, and nullability conventions applied to a property.

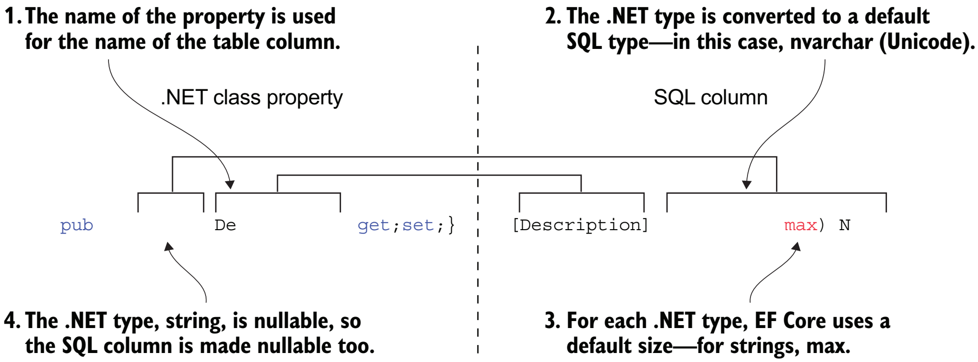


Figure 7.3 The application of the By Convention rules to define an SQL column. The type of the property is converted by the database provider to the equivalent SQL type, whereas the name of the property is used for the name of the column.

7.3.5 An EF Core naming convention identifies primary keys

The other rule is about defining the database table's primary key. The EF Core conventions for designating a primary key are as follows:

- EF Core expects one primary-key property. (The By Convention approach doesn't handle keys made up of multiple properties/columns, called *composite keys.*)
- The property is called `Id` or `<class name>id` (such as `BookId`).
- The type of the property defines what assigns a unique value to the key. Chapter 8 covers key generation.

Figure 7.4 shows an example of a database-generated primary key with By Convention mapping for the `Book`'s `BookId` property and the Books table's SQL column BookId.

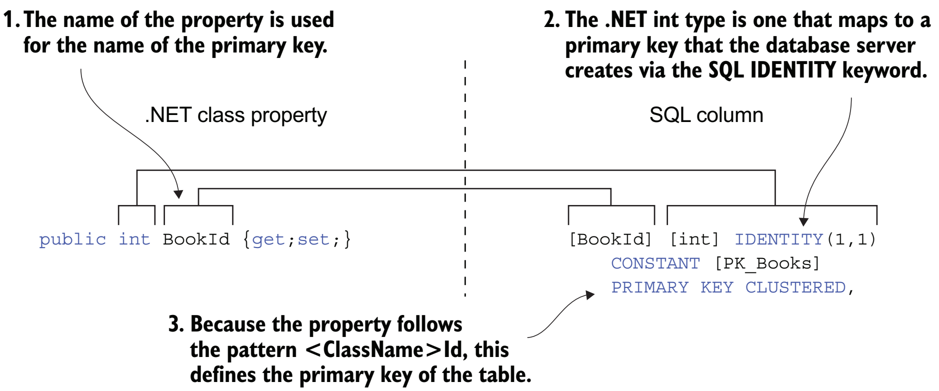


Figure 7.4 The mapping between the .NET class property `BookId` and the SQL primary column `BookId`, using the By Convention approach. The name of the property tells EF Core that this property is the primary key. Also, the database provider knows that a type of `int` means that it should create a unique value for each row added to the table.

TIP Although you have the option of using the short name, `Id`, for a primary key, I recommend that you use the longer name: `<class name>` followed by `Id` (`BookId`, for example). Understanding what's going on in your code is easier if you use `Where(p => BookId == 1)` rather than the shorter `Where(p => Id == 1)`, especially when you have lots of entity classes.

7.4 Configuring via Data Annotations

Data Annotations are a specific type of .NET attribute used for validation and database features. These attributes can be applied to an entity class or property and provide configuration information to EF Core. This section introduces where you can find them and how they're typically applied. The Data Annotation attributes that are relevant to EF Core configuration come from two namespaces.

7.4.1 Using annotations from System.ComponentModel.DataAnnotations

The attributes in the `System.ComponentModel.DataAnnotations` namespace are used mainly for data validation at the frontend, such as ASP.NET, but EF Core uses some of them for creating the mapping model. Attributes such as `[Required]` and `[MaxLength]` are the main ones, with many of the other Data Annotations having no effect on EF Core. Figure 7.5 shows how the main attributes, `[Required]` and `[MaxLength]`, affect the database column definition.

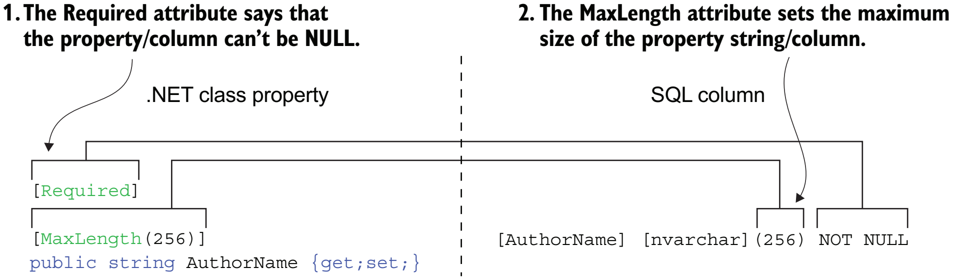


Figure 7.5 The `[Required]` and `[MaxLength]` attributes affect the mapping to a database column. The `[Required]` attribute indicates that the column shouldn't be `null`, and the `[MaxLength]` attribute sets the size of the `nvarchar`.

7.4.2 Using annotations from System.ComponentModel.DataAnnotations.Schema

The attributes in the `System.ComponentModel.DataAnnotations.Schema` namespace are more specific to database configuration. This namespace was added in NET Framework 4.5, well before EF Core was written, but EF Core uses its attributes, such as `[Table]`, `[Column]`, and so on, to set the table name and column name/type, as described in section 7.11.

7.5 Configuring via the Fluent API

The third approach to configuring EF Core, called the *Fluent API*, is a set of methods that works on the `ModelBuilder` class that's available in the `OnModelCreating` method inside your application's DbContext. As you will see, the Fluent API works by extension methods that can be chained together, as LINQ commands are chained together, to set a configuration setting. The Fluent API provides the most comprehensive list of configuration commands, with many configurations available only via that API.

But before defining the Fluent API relationship commands, I want to introduce a different approach that segregates your Fluent API commands into per-entity class sized groups. This approach is useful because as your application grows, putting all Fluent API commands in the `OnModelCreating` method (as shown in figure 2.6) makes finding a specific Fluent API hard work. The solution is to move the Fluent API for an entity class into a separate configuration class that's then called from the `OnModelCreating` method.

EF Core provides a method to facilitate this process in the shape of the `IEntityTypeConfiguration<T>` interface. Listing 7.2 shows your new application DbContext, `EfCoreContext`, where you move the Fluent API setup of the various classes into separate configuration classes. The benefit of this approach is that the Fluent API for an entity class is all in one place, not mixed with Fluent API commands for other entity classes.

EF6 EF6.x has an `EntityTypeConfiguration<T>` class that you can inherit to encapsulate the Fluent API configuration for a given entity class. EF Core's implementation achieves the same result but uses an `IEntityTypeConfiguration<T>` interface that you apply to your configuration class.

Listing 7.2 Application's DbContext for database with relationships

```
public class EfCoreContext : DbContext
{
    public EfCoreContext(DbContextOptions<EfCoreContext> options)
        : base(options)
    { }

    public DbSet<Book> Books { get; set; }
    public DbSet<Author> Authors { get; set; }
    public DbSet<PriceOffer> PriceOffers { get; set; }
    public DbSet<Order> Orders { get; set; }

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.ApplyConfiguration(new BookConfig());
        modelBuilder.ApplyConfiguration(new BookAuthorConfig());
        modelBuilder.ApplyConfiguration(new PriceOfferConfig());
        modelBuilder.ApplyConfiguration(new LineItemConfig());
    }
}
```

UserId of the user who has bought some books

Creates the DbContext, using the options set up when you registered the DbContext

The entity classes that your code will access

The method in which your Fluent API commands run

Run each of the separate configurations for each entity class that needs configuration.

Let's look at the `BookConfig` class used in listing 7.2 to see how you would construct a per-type configuration class. Listing 7.3 shows a configuration class that implements the `IEntityTypeConfiguration<T>` interface and contains the Fluent API methods for the `Book` entity class.

NOTE I am not describing the Fluent APIs in listing 7.3 because it is an example of the use of the `IEntityTypeConfiguration<T>` interface. The Fluent APIs are covered in section 7.7 (database type) and section 7.10 (indexes).

Listing 7.3 `BookConfig` extension class configures `Book` entity class

```
internal class BookConfig : IEntityTypeConfiguration<Book>
{
    public void Configure
        (EntityTypeBuilder<Book> entity)
    {
        entity.Property(p => p.PublishedOn)
            .HasColumnType("date");

        entity.Property(p => p.Price)
            . HasPrecision(9,2);

        entity.Property(x => x.ImageUrl)
            .IsUnicode(false);

        entity.HasIndex(x => x.PublishedOn);
    }
}
```

Convention-based mapping for .NET DateTime is SQL datetime2. This command changes the SQL column type to date, which holds only the date, not the time.

The precision of (9,2) sets a max price of 9,999,999.99 (9 digits, 2 after decimal point), which takes up the smallest size in the database.

The convention-based mapping for .NET string is SQL nvarchar (16 bit Unicode). This command changes the SQL column type to varchar (8-bit ASCII).

Adds an index to the PublishedOn property because you sort and filter on this property

In listing 7.2, I list each of the separate `modelBuilder.ApplyConfiguration` calls so that you can see them in action. But a time-saving method called `ApplyConfigurationsFromAssembly` can find all your configuration classes that inherit `IEntityTypeConfiguration<T>` and run them all for you. See the following code snippet, which finds and runs all your configuration classes in the same assembly as the DbContext:

```
modelBuilder.ApplyConfigurationsFromAssembly(
     Assembly.GetExecutingAssembly());
```

Listing 7.3 shows a typical use of the Fluent API, but please remember that the fluent nature of the API allows chaining of multiple commands, as shown in this code snippet:

```
modelBuilder.Entity<Book>()
    .Property(x => x.ImageUrl)
    .IsUnicode(false)
    .HasColumnName("DifferentName")
    .HasMaxLength(123)
    .IsRequired(false);
```

> **EF6** The Fluent API works the same in EF6.x, but with lots of new features and substantial changes in setting up relationships (covered in chapter 8) and subtle changes in data types.

`OnModelCreating` is called when the application first accesses the application's DbContext. At that stage, EF Core configures itself by using all three approaches: By Convention, Data Annotations, and any Fluent API you've added in the `OnModelCreating` method.

What if Data Annotations and the Fluent API say different things?

The Data Annotations and the Fluent API modeling methods always override convention-based modeling. But what happens if a Data Annotation and the Fluent API both provide a mapping of the same property and setting?

I tried setting the SQL type and length of the `WebUrl` property to different values via Data Annotations and via the Fluent API. The Fluent API values were used. That test wasn't a definitive one, but it makes sense that the Fluent API was the final arbitrator.

Now that you've learned about the Data Annotations and Fluent API configuration approaches, let's detail the configuration of specific parts of the database model.

7.6 Excluding properties and classes from the database

Section 7.3.2 described how EF Core finds properties. But at times, you'll want to exclude data in your entity classes from being in the database. You might want to have local data for a calculation used during the lifetime of the class instance, for example, but you don't want it saved to the database. You can exclude a class or a property in two ways: via Data Annotations or via the Fluent API.

7.6.1 Excluding a class or property via Data Annotations

EF Core will exclude a property or a class that has a `[NotMapped]` data attribute applied to it. The following listing shows the application of the `[NotMapped]` data attribute to both a property and a class.

Listing 7.4 Excluding three properties, two by using `[NotMapped]`

```
public class MyEntityClass
{
    public int MyEntityClassId { get; set; }

    public string NormalProp{ get; set; }

     [NotMapped]
    public string LocalString { get; set; }

    public ExcludeClass LocalClass { get; set; }
}

[NotMapped]
public class ExcludeClass
{
    public int LocalInt { get; set; }
}
```

Included: A normal public property, with public getter and setter

Excluded: Placing a [NotMapped] attribute tells EF Core to not map this property to a column in the database.

Excluded: This class won't be included in the database because the class definition has a [NotMapped] attribute on it.

Excluded: This class will be excluded because the class definition has a [NotMapped] attribute on it.

7.6.2 Excluding a class or property via the Fluent API

In addition, you can exclude properties and classes by using the Fluent API configuration command `Ignore`, as shown in listing 7.5.

> **NOTE** For simplicity, I show the Fluent API inside the `OnModelCreating` method rather than in a separate configuration class.

Listing 7.5 Excluding a property and a class by using the Fluent API

```
public class ExcludeDbContext : DbContext
{
    public DbSet<MyEntityClass> MyEntities { get; set; }

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<MyEntityClass>()
            .Ignore(b => b.LocalString);

        modelBuilder.Ignore<ExcludeClass>();
    }
}
```

The Ignore method is used to exclude the LocalString property in the entity class, MyEntityClass, from being added to the database.

A different Ignore method can exclude a class such that if you have a property in an entity class of the Ignored type, that property isn't added to the database.

As I said in section 7.3.2, by default, EF Core will ignore read-only properties—that is, a property with only a getter (such as `public int MyProp { get; })`.

7.7 Setting database column type, size, and nullability

As described earlier, the convention-based modeling uses default values for the SQL type, size/precision, and nullability based on the .NET type. A common requirement is to set one or more of these attributes manually, either because you're using an existing database or because you have performance or business reasons to do so.

In the introduction to configuring (section 7.3), you worked through an example that changed the type and size of various columns. Table 7.1 provides a full list of the commands that are available to perform this task.

Table 7.1 Setting nullability and SQL type/size for a column

Setting	Data Annotations	Fluent API

Set not null
(Default is nullable.)
	`[Required]`
	`public string MyProp`
	`{ get; set; }`
		`modelBuilder.Entity<MyClass>()`
		`.Property(p => p.MyProp)`
		`.IsRequired();`

Set size (string)
(Default is `MAX` length.)
	`[MaxLength(123)]`
	`public string MyProp`
	`{ get; set; }`
		`modelBuilder.Entity<MyClass>()`
		`.Property(p => p.MyProp)`
		`.HasMaxLength(123);`

Set SQL type/size
(Each type has a default precision and size.)
	`[Column(TypeName =`
	`"date")]`
	`public DateTime`
	`PublishedOn`
	`{ get; set; }`
		`modelBuilder.Entity<MyClass>(`
		`.Property(p =>`
		`p.PublishedOn)`
		`.HasColumnType("date");`

Some specific SQL types have their own Fluent API commands, which are shown in the following list. You can see the first Fluent API commands in use in listing 7.3:

- `IsUnicode(false)`—Sets the SQL type to `varchar(nnn)` (1-byte character, known as ASCII) rather than the default of `nvarchar(nnn)` (2-byte character, known as Unicode).
- `HasPrecision(precision, scale)`—Sets the number of digits (`precision` parameter) and how many of the digits are after the decimal point (`scale` parameter). This Fluent command is new in EF Core 5. The default setting of the SQL `decimal` is (18,2).
- `HasCollation("collation name")`—Another EF Core 5 feature that allows you to define the collation on a property—that is, the sorting rules, case, and accent sensitivity properties of `char` and `string` types. (See section 2.8.3 for more about collations.)

I recommend using the `IsUnicode(false)` method to tell EF Core that a string property contains only single-byte ASCII-format characters, because using the `IsUnicode` method allows you to set the string size separately.

EF6 EF Core has a slightly different approach to setting the SQL data type of a column. If you provide the data type, you need to give the whole definition, both type and length/precision—as in `[Column(TypeName = "varchar(nnn)")]`, where `nnn` is an integer number. In EF6, you can use `[Column(TypeName = "varchar")]` and then define the length by using `[MaxLength(nnn)]`, but that technique doesn't work in EF Core. See https://github.com/dotnet/efcore/issues/3985 for more information.

7.8 Value conversions: Changing data to/from the database

EF Core's value conversions feature allows you to change data when reading and writing a property to the database. Typical uses are

- Saving `Enum` type properties as a string (instead of a number) so that it's easier to understand when you're looking at the data in the database
- Fixing the problem of `DateTime` losing its UTC (Coordinated Universal Time) setting when read back from the database
- (Advanced) Encrypting a property written to the database and decrypting on reading back

The value conversions have two parts:

- Code that transforms the data as it is written out to the database
- Code that transforms the database column back to the original type when read back

The first example of value conversions deals with a limitation of the SQL database in storing `DateTime` types, in that it doesn't save the `DateTimeKind` part of the `DateTime` struct that tells us whether the `DateTime` is local time or UTC. This situation can cause problems. If you send that `DateTime` to your frontend using JSON, for example, the DateTime won't contain the Z suffix character that tells JavaScript that the time is UTC, so your frontend code may display the wrong time. The following listing shows how to configure a property to have a value conversion that sets the `DateTimeKind` on the return from the database.

Listing 7.6 Configuring a `DateTime` property to replace the lost `DateTimeKind` setting

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
    var utcConverter = new ValueConverter<DateTime, DateTime>(
        toDb => toDb,
        fromDb =>
            DateTime.SpecifyKind(fromDb, DateTimeKind.Utc));

    modelBuilder.Entity<ValueConversionExample>()
        .Property(e => e.DateTimeUtcUtcOnReturn)
        .HasConversion(utcConverter);
    //… other configurations left out
}
```

Creates a ValueConverter from DateTime to DateTime

Saves the DateTime to the database in the normal way (such as no conversion)

On reading from the database, you add the UTC setting to the DateTime.

Selects the property you want to configure

Adds the utcConverter to that property

In this case, you had to create your own value converter, but about 20 built-in value converters are available. (See http://mng.bz/mgYP.) In fact, one value converter is so popular that it has a predefined Fluent API method or an attribute—a conversion to store an `Enum` as a string in the database. Let me explain.

`Enums` are normally stored in the database as numbers, which is an efficient format, but it does make things harder if you need to delve into the database to work out what happened. So some developers like to save `Enums` in the database as a string. You can configure a conversion of an `Enum` type to a string by using the `HasConversion<string>()` command, as in the following code snippet:

```
modelBuilder.Entity<ValueConversionExample>()
    .Property(e => e.Stage)
    .HasConversion<string>();
```

Following are some rules and limitations on using value conversions:

- A `null` value will never be passed to a value converter. You need to write a value converter to handle only the non-null value, as your converter will be called only if the value isn't a `null`.
- Watch out for queries that contain sorting on a converted value. If you converted your `Enums` to a string, for example, the sorting will sort by the `Enum` name, not by the `Enum` value.
- The converter can only map a single property to a single column in the database.
- You can create some complex value converters, such as serializing a list of `ints` to a JSON string. At this point, EF Core cannot compare the `List<int>` property

with the JSON in the database, so it won't update the database. To solve this problem, you need to add what is called a *value comparer*. See the EF Core doc at http://mng.bz/5j5z for more information on this topic.

Later, in section 7.16.4, you will learn a way to automatically apply value converters to certain property types/names to make your life easier.

7.9 The different ways of configuring the primary key

You've already seen the By Convention approach of setting up the primary key of an entity. This section covers the normal primary-key setting—one key for which the .NET property defines the name and type. You need to configure the primary key explicitly in two situations:

- When the key name doesn't fit the By Convention naming rules
- When the primary key is made up of more than one property/column, called a *composite key*

A many-to-many relationship-linking table is an example of where the By Convention approach doesn't work. You can use two alternative approaches to define primary keys.

NOTE Chapter 8 deals with configuring foreign keys, because they define relationships even though they're of a scalar type.

7.9.1 Configuring a primary key via Data Annotations

The `[Key]` attribute allows you to designate one property as the primary key in a class. Use this annotation when you don't use the By Convention primary key name, as shown in the following listing. This code is simple and clearly marks the primary key.

Listing 7.7 Defining a property as the primary key bu using the `[Key]` annotation

```
private class SomeEntity
{
    [Key]
    public int NonStandardKeyName { get; set; }

    public string MyString { get; set; }
}
```

[Key] attribute tells EF Core that the property is a primary key.

Note that the `[Key]` attribute can't be used for composite keys. In earlier versions of EF Core, you could define composite keys by using `[Key]` and `[Column]` attributes, but that feature has been removed.

7.9.2 Configuring a primary key via the Fluent API

You can also configure a primary key via the Fluent API, which is useful for primary keys that don't fit the By Convention patterns. The following listing shows two primary keys being configured by the Fluent API's `HasKey` method. The first primary key is a single primary key with a nonstandard name in the `SomeEntity` entity class, and the second is a composite primary key, consisting of two columns, in the BookAuthor linking table.

Listing 7.8 Using the Fluent API to configure primary keys on two entity classes

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<SomeEntity>()
        .HasKey(x => x.NonStandardKeyName);

    modelBuilder.Entity<BookAuthor>()
        .HasKey(x => new {x.BookId, x.AuthorId});

    //… other configuration settings removed
}
```

Defines a normal, single-column primary key. Use HasKey when your key name doesn't match the By Convention defaults.

Uses an anonymous object to define two (or more) properties to form a composite key. The order in which the properties appear in the anonymous object defines their order.

There is no By Convention version for composite keys, so you must use the Fluent API's `HasKey` method.

7.9.3 Configuring an entity as read-only

In some advanced situations, your entity class might not have a primary key. Here are three examples:

- *You want to define an entity class as read-only.* If an entity class hasn't got a primary key, then EF Core will treat it as read-only.
- *You want to map an entity class to a read-only SQL View. SQL Views* are SQL queries that work like SQL tables. See this article for more information: http://mng.bz/6g6y.
- *You want to map an entity class to an SQL query by using the* `ToSqlQuery` *Fluent API command.* The `ToSqlQuery` method allows you to define an SQL command string that will be executed when you read in that entity class.

To set an entity class explicitly as read-only, you can use the fluent API `HasNoKey()` command or apply the attribute `[Keyless]` to the entity class. And if your entity class doesn't have a primary key, you must mark it as read-only, using either of the two approaches. Any attempt to change the database via an entity class with no primary key will fail with an exception. EF Core does this because it can't execute the update without a key, which is one way you can define an entity class as read-only. The other way to mark an entity as read-only is to map an entity to an SQL View by using the fluent API method `ToView("ViewNameString")` command, as shown in the following code snippet:

```
modelBuilder.Entity<MyEntityClass>()
    .ToView("MyView");
```

EF Core will throw an exception if you try to change the database via an entity class that is mapped to a View. If you want to map an entity class to an updatable view—an SQL View that can be updated—you should use the `ToTable` command instead.

7.10 Adding indexes to database columns

Relational databases have a feature called an *index,* which provides quicker searching and sorting of rows based on the column, or columns, in the index. In addition, an index may have a constraint, which ensures that each entry in the index is unique. A primary key is given a unique index, for example, to ensure that the primary key is different for each row in the table.

You can add an index to a column via Fluent API and attributes, as shown in table 7.2. An index will speed quick searching and sorting, and if you add the unique constraint, the database will ensure that the column value in each row will be different.

Table 7.2 Adding an index to a column

Action

Fluent API

Add index, Fluent

```
modelBuilder.Entity<MyClass>()
    .HasIndex(p => p.MyProp);
```

Add index, Attribute

```
[Index(nameof(MyProp))]
public class MyClass …
```

Add index, multiple columns

```
modelBuilder.Entity<Person>()
     .HasIndex(p => new {p.First, p.Surname});
```

Add index, multiple columns, Attribute

```
[Index(nameof(First), nameof(Surname)]
public class MyClass …
```

Add unique index, Fluent

```
modelBuilder.Entity<MyClass>()
     .HasIndex(p => p.BookISBN)
     .IsUnique();
```

Add unique index, Attribute

```
[Index(nameof(MyProp), IsUnique = true)]
public class MyClass …
```

Add named index, Fluent

```
modelBuilder.Entity<MyClass>()
     .HasIndex(p => p.MyProp)
     .HasDatabaseName("Index_MyProp");
```

TIP Don't forget that you can chain the Fluent API commands together to mix and match these methods.

Some databases allow you to specify a filtered or partial index to ignore certain situations by using a `WHERE` clause. You could set a unique filtered index that ignored any soft-deleted items, for example. To set up a filtered index, you use the `HasFilter` Fluent API method containing an SQL expression to define whether the index should be updated with the value. The following code snippet gives an example of enforcing that the property `MyProp` will contain a unique value unless the SoftDeleted column of the table is `true`:

```
modelBuilder.Entity<MyClass>()
    .HasIndex(p => p.MyProp)
    .IsUnique()
    .HasFilter(“NOT SoftDeleted");
```

NOTE When you're using the SQL Server provider, EF adds an `IS NOT NULL` filter for all nullable columns that are part of a unique index. You can override this convention by providing `null` to the `HasFilter` parameter—that is `HasFilter(null)`.

7.11 Configuring the naming on the database side

If you're building a new database, using the default names for the various parts of the database is fine. But if you have an existing database, or if your database needs to be accessed by an existing system you can't change, you most likely need to use specific names for the *schema* name, the table names, and the column names of the database.

DEFINITION *Schema* refers to the organization of data inside a database—the way the data is organized as tables, columns, constraints, and so on. In some databases, such as SQL Server, *schema* is also used to give a namespace to a particular grouping of data that the database designer uses to partition the database into logical groups.

7.11.1 Configuring table names

By convention, the name of a table is set by the name of the `DbSet<T>` property in the application's DbContext, or if no `DbSet<T>` property is defined, the table uses the class name. In the application's DbContext of our Book App, for example, you defined a `DbSet<Book>` Books property, so the database table name is set to Books. Conversely, you haven't defined a `DbSet<T>` property for the `Review` entity class in the application's DbContext, so its table name used the class name and is, therefore, Review.

If your database has specific table names that don't fit the By Convention naming rules—for example, if the table name can't be converted to a valid .NET variable name because it has a space in it—you can use either Data Annotations or the Fluent API to set the table name specifically. Table 7.3 summarizes the two approaches to setting the table name.

Table 7.3 Two ways to configure a table name explicitly for an entity class

Configuration method	Example: Setting the table name of the `Book` class to "XXX"
Data Annotations	`[Table("XXX")]` `public class Book … etc.`
Fluent API	`modelBuilder.Entity<Book>().ToTable("XXX");`

7.11.2 Configuring the schema name and schema groupings

Some databases, such as SQL Server, allow you to group your tables by using what is called a schema name. You could have two tables with the same name but different schema names: a table called Books with a schema name Display, for example, would be different from a table called Books with a schema name Order.

By convention, the schema name is set by the database provider because some databases, such as SQLite and MySQL, don't support schemas. In the case of SQL Server, which does support schemas, the default schema name is *dbo*, which is the SQL Server default name. You can change the default schema name only via the Fluent API, using the following snippet in the `OnModelCreating` method of your application's DbContext:

```
modelBuilder.HasDefaultSchema("NewSchemaName");
```

Table 7.4 shows how to set the schema name for a table. You use this approach if your database is split into logical groups such as sales, production, accounts, and so on, and a table needs to be specifically assigned to a schema.

Table 7.4 Setting the schema name on a specific table

Configuration method	Example: Setting the schema name "sales" on a table
Data Annotations	`[Table("SpecialOrder", Schema = "sales")]` `class MyClass … etc.`
Fluent API	

```
modelBuilder.Entity<MyClass>()
    .ToTable("SpecialOrder", schema: "sales");
```

7.11.3 Configuring the database column names in a table

By convention, the column in a table has the same name as the property name. If your database has a name that can't be represented as a valid .NET variable name or doesn't fit the software use, you can set the column names by using Data Annotations or the Fluent API. Table 7.5 shows the two approaches.

Table 7.5 The two ways to configure a column name

Configuration method	Setting the column name of the `BookId` property to `SpecialCol`
Data Annotations	`[Column("SpecialCol")]` `public int BookId { get; set; }`
Fluent API	

```
modelBuilder.Entity<MyClass>()
    .Property(b => b.BookId)
    .HasColumnName("SpecialCol");
```

7.12 Configuring Global Query Filters

Many applications, such as ASP.NET Core, have security features that control what views and controls the user can access. EF Core has a similar security feature called *Global Query Filters* (shortened to *Query Filters*). You can use Query Filters to build a multitenant application. This type of application holds data for different users in one database, but each user can see only the data they are allowed to access. Another use is to implement a soft-delete feature; instead of deleting data in the database, you might use a Query Filter to make the soft-deleted row disappear, but the data will still be there if you need to undelete it later.

I have found Query Filters to be useful in many client jobs, so I included a detailed section called "Using Global Query Filters in real-world situations" in chapter 6 (section 6.1.6). That section contains information on how to configure Query Filters, so please look there for that information. In section 7.16.4 of this chapter, I show how you can automate the configuration of Query Filters, which ensures that you won't forget to add an important Query Filter to one of your entity classes.

7.13 Applying Fluent API commands based on the database provider type

The EF Core database providers provide a way to detect what database provider is being used when an instance of an application DbContext is created. This approach is useful for situations such as using, say, an SQLite database for your unit tests, but the production database is on an SQL Server, and you want to change some things to make your unit tests work.

SQLite, for example, doesn't fully support a few NET types, such as `decimal`, so if you try to sort on a decimal property in an SQLite database, you'll get an exception saying that you won't get the right result from an SQLite database. One way to get around this issue is to convert the `decimal` type to a `double` type when using SQLite; it won't be accurate, but it might be OK for a controlled set of unit tests.

Each database provider provides an extension method to return `true` if the database matches that provider. The SQL Server database provider, for example, has a method called `IsSqlServer()`; the SQLite database provider has a method called `IsSqlite()`; and so on. Another approach is to use the `ActiveProvider` property in the `ModelBuilder` class, which returns a string that is the NuGet package name of the database provider, such as `"Microsoft.EntityFrameworkCore.SqlServer"`.

The following listing is an example of applying the `decimal` to `double` type change if the database is SQLite. This code allows the Book App's `OrderBooksBy` query object method to use an in-memory SQLite database.

Listing 7.9 Using database-provider commands to set a column name

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
    //… put your normal configration here
```

The IsSqlite will return true if the database provided in the options is SQLite.

```
if (Database.IsSqlite())
{
    modelBuilder.Entity<Book>()
        .Property(e => e.Price)
        .HasConversion<double>();
    modelBuilder.Entity<PriceOffer>()
        .Property(e => e.NewPrice)
        .HasConversion<double>();
}
}
```

You set the two decimal values to double so that a unit test that sorts on these values doesn't throw an exception.

EF Core 5 added the `IsRelational()` method, which returns `false` for database providers that aren't relational, such as Cosmos Db. You can find a few database-specific Fluent API commands, such as the SQL Server provider method `IsMemoryOptimized`, in the EF Core documentation for each database provider.

> **NOTE** Although you could use this approach to create migrations for different production database types, it's not recommended. The EF Core team suggests that you create a migration for each database type and store each migration in separate directories. For more information, see chapter 9.

7.14 Shadow properties: Hiding column data inside EF Core

> **EF6** EF6.x had the concept of shadow properties, but they were used only internally to handle missing foreign keys. In EF Core, shadow properties become a proper feature that you can use.

Shadow properties allow you to access database columns without having them appear in the entity class as a property. Shadow properties allow you to "hide" data that you consider not to be part of the normal use of the entity class. This is all about good software practice: you let upper layers access only the data they need, and you hide anything that those layers don't need to know about. Let me give you two examples that show when you might use shadow properties:

- A common need is to track by whom and when data was changed, maybe for auditing purposes or to understand customer behavior. The tracking data you receive is separate from the primary use of the class, so you may decide to implement that data by using shadow properties, which can be picked up outside the entity class.
- When you're setting up relationships in which you don't define the foreign-key properties in your entity class, EF Core must add those properties to make the relationship work, and it does this via shadow properties. Chapter 8 covers this topic.

7.14.1 Configuring shadow properties

There's a By Convention approach to configuring shadow properties, but because it relates only to relationships, I explain it in chapter 8. The other method is to use the Fluent API. You can introduce a new property by using the Fluent API method

`Property<T>`. Because you're setting up a shadow property, there won't be a property of that name in the entity class, so you need to use the Fluent API's `Property<T>` method, which takes a .NET Type and the name of the shadow property. The following listing shows the setup of a shadow property called `UpdatedOn` that's of type `DateTime`.

Listing 7.10 Creating the `UpdatedOn` shadow property by using the Fluent API

```
public class Chapter06DbContext : DbContext
{
    …

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<MyEntityClass>()
            .Property<DateTime>("UpdatedOn");
    …
    }
}
```

Uses the Property<T> method to define the shadow property type

Under By Convention, the name of the table column the shadow property is mapped to is the same as the name of the shadow property. You can override this setting by adding the `HasColumnName` method on to the end of the `property` method.

> **WARNING** If a property of that name already exists in the entity class, the configuration will use that property instead of creating a shadow property.

7.14.2 Accessing shadow properties

Because the shadow properties don't map to a class property, you need to access them directly via EF Core. For this purpose, you have to use the EF Core command `Entry(myEntity).Property("MyPropertyName").CurrentValue`, which is a read/write property, as shown in the following listing.

Listing 7.11 Using `Entry(inst).Property(name)` to set the shadow property

```
var entity = new SomeEntityClass();
context.Add(entity);
context.Entry(entity)
    .Property("UpdatedOn").CurrentValue
        = DateTime.Now;
context.SaveChanges();
```

Creates an entity class ...

... and adds it to the context, so it's now tracked

Gets the EntityEntry from the tracked entity data

Uses the Property method to get the shadow property with read/write access

Sets that property to the value you want

Calls SaveChanges to save the MyEntityClass instance, with its normal and shadow property values, to the database

If you want to read a shadow property in an entity that has been loaded, use the `context.Entry(entityInstance).Property("PropertyName").CurrentValue` command. But you must read the entity as a tracked entity; you should read the entity without the `AsNoTracking` method being used in the query. The `Entry(<entityInstance>).Property` method uses the tracked entity data inside EF Core to hold the value, as it's not held in the entity class instance.

In LINQ queries, you use another technique to access a shadow property: the `EF.Property` command. You could sort by the `UpdatedOn` shadow property, for example, by using the following query snippet, with the `EF.Property` method in bold:

```
context.MyEntities
     .OrderBy(b => EF.Property<DateTime>(b, "UpdatedOn"))
     .ToList();
```

7.15 Backing fields: Controlling access to data in an entity class

EF6 Backing fields aren't available in EF6. This EF Core feature provides a level of control over access to data that EF6.x users have been after for some time.

As you saw earlier, columns in a database table are normally mapped to an entity class property with normal getters and setters—`public int MyProp { get ; set; }`. But you can also map a private field to your database. This feature is called a *backing field*, and it gives you more control of the way that database data is read or set by the software.

Like shadow properties, backing fields hide data, but they do the hiding in another way. For shadow properties, the data is hidden inside EF Core's data, but backing fields hide the data inside the entity class, so it's easier for the entity class to access the backing field inside the class. Here are some examples of situations in which you might use backing fields:

- *Hiding sensitive data*—Hiding a person's date of birth in a private field and making their age in years available to the rest of the software.
- *Catching changes*—Detecting an update of a property by storing the data in a private field and adding code in the setter to detect the update of a property. You will use this technique in chapter 12, when you use property change to trigger an event.
- *Creating Domain-Driven Design (DDD) entity classes*—Creating DDD entity classes in which all the entity classes' properties need to be read-only. Backing fields allow you to lock down navigational collection properties, as described in section 8.7.

But before you get into the complex versions, let's start with the simplest form of backing fields, in which a property getter/setter accesses the field.

7.15.1 Creating a simple backing field accessed by a read/write property

The following code snippet shows you a string property called `MyProperty`, in which the string data is stored in a private field. This form of backing field doesn't do anything particularly different from using a normal property, but this example shows the concept of a property linked to a private field:

```
public class MyClass
{
    private string _myProperty;
    public string MyProperty
    {
        get { return _myProperty; }
        set { _myProperty = value; }
    }
}
```

EF Core's By Convention configuration will find the type of backing field and configure it as a backing field (see section 7.15.4 for backing-field configuration options), and by default, EF Core will read/write the database data to this private field.

7.15.2 Creating a read-only column

Creating a read-only column is the most obvious use, although it can also be implemented via a private setting property (see section 7.3.2). If you have a column in the database that you need to read but don't want the software to write, a backing field is a great solution. In this case, you can create a private field and use a public property, with a getter only, to retrieve the value. The following code snippet gives you an example:

```
public class MyClass
{
    private string _readOnlyCol;
    public string ReadOnlyCol => _readOnlyCol;
}
```

Something must set the column property, such as setting a default value in the database column (covered in chapter 9) or through some sort of internal database method.

7.15.3 Concealing a person's date of birth: Hiding data inside a class

Hiding a person's date of birth is a possible use of backing fields. In this case, you deem for security reasons that a person's date of birth can be set, but only their age can be read from the entity class. The following listing shows how to do this in the `Person` class by using a private `_dateOfBirth` field and then providing a method to set it and a property to calculate the person's age.

Listing 7.12 Using a backing field to hide sensitive data from normal access

```
public class Person
{
    private DateTime _dateOfBirth;

    public void SetDateOfBirth(DateTime dateOfBirth)
    {
        _dateOfBirth = dateOfBirth;
    }

    public int AgeYears =>
        Years(_dateOfBirth, DateTime.Today);

    //Thanks to dana on stackoverflow
    //see
    private static int Years(DateTime start, DateTime end)
    {
        return (end.Year - start.Year - 1) +
               (((end.Month > start.Month) ||
                 ((end.Month == start.Month)
                  && (end.Day >= start.Day)))
                   ? 1 : 0);
    }}
```

The private backing field, which can't be accessed directly via normal .NET software

Allows the backing field to be set

You can access the person's age but not their exact date of birth.

NOTE In the preceding example, you need to use the Fluent API to create a backing-field-only variable (covered in section 7.15.2), because EF Core can't find this backing field by using the By Convention approach.

From the class point of view, the `_dateOfBirth` field is hidden, but you can still access the table column via various EF Core commands in the same way that you accessed the shadow properties: by using the `EF.Property<DateTime>(entity, "_dateOfBirth")` method.

The backing field, `_dateOfBirth`, isn't totally secure from the developer, but that's not the aim. The idea is to remove the date-of-birth data from the normal properties so that it doesn't get displayed unintentionally in any user-visible view.

7.15.4 Configuring backing fields

Having seen backing fields in action, you can configure them By Convention, via Fluent API, and now in EF Core 5 via Data Annotations. The By Convention approach works well but relies on the class to have a property that matches a field by type and a naming convention. If a field doesn't match the property name/type or doesn't have a matching property such as in the `_dateOfBirth` example, you need to configure your backing fields with Data Annotations or by using the Fluent API. The following sections describe the various configuration approaches.

CONFIGURING BACKING FIELDS BY CONVENTION

If your backing field is linked to a valid property (see section 7.3.2), the field can be configured by convention. The rules for By Convention configuration state that the private field must have one of the following names that match a property in the same class:

- `_<property name>` (for example, `_MyProperty`)
- `_<camel-cased property name >` (for example, `_myProperty`)
- `m_<property name>` (for example, `m_MyProperty`)
- `m_<camel-cased property name>` (for example, `m_myProperty`)

DEFINITION *Camel case* is a convention in which a variable name starts with a lowercase letter but uses an uppercase letter to start each subsequent word in the name—as in `thisIsCamelCase`.

CONFIGURING BACKING FIELDS VIA DATA ANNOTATIONS

New in EF Core 5 is the `BackingField` attribute, which allows you to link a property to a private field in the entity class. This attribute is useful if you aren't using the By Convention backing field naming style, as in this example:

```
private string _fieldName;
[BackingField(nameof(_fieldName))]
public string PropertyName
{
    get { return _fieldName; }
}

public void SetPropertyNameValue(string someString)
{
    _fieldName = someString;
}
```

CONFIGURING BACKING FIELDS VIA THE FLUENT API

You have several ways of configuring backing fields via the Fluent API. We'll start with the simplest and work up to the more complex. Each example shows you the `OnModelCreating` method inside the application's DbContext, with only the field part being configured:

- *Setting the name of the backing field*—If your backing field name doesn't follow EF Core's conventions, you need to specify the field name via the Fluent API. Here's an example:

  ```
  protected override void OnModelCreating
      (ModelBuilder modelBuilder)
  {
      modelBuilder.Entity<Person>()
          .Property(b => b.MyProperty)
          .HasField("_differentName");
      …
  }
  ```

- *Supplying only the field name*—In this case, if there's a property with the correct name, by convention EF Core will refer to the property, and the property name will be used for the database column. Here's an example:

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .Property("_dateOfBirth")
        .HasColumnName("DateOfBirth");
    …
}
```

  If no property getter or setter is found, the field will still be mapped to the column, using its name, which in this example is `_dateOfBirth`, but that's most likely not the name you want for the column. So you add the `HasColumnName` Fluent API method to get a better column name. The downside is that you'd still need to refer to the data in a query by its field name (in this case, `_dateOfBirth`), which isn't too friendly or obvious.

ADVANCED: CONFIGURING HOW DATA IS READ/WRITTEN TO THE BACKING FIELD

Since the release of EF Core 3, the default database access mode for backing fields is for EF Core to read and write to the field. This mode works in nearly all cases, but if you want to change the database access mode, you can do so via the Fluent API `UsePropertyAccessMode` method. The following code snippet tells EF Core to try to use the property for read/write, but if the property is missing a setter, EF Core will fill in the field on a database read:

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .Property(b => b.MyProperty)
                .HasField("_differentName")
        .UsePropertyAccessMode(PropertyAccessMode.PreferProperty);
    …
}
```

TIP To see the various access modes for a backing field, use Visual Studio's intellisense feature to look at the comments on each of the `PropertyAccessMode` `Enum` values.

7.16 Recommendations for using EF Core's configuration

You have so many ways to configure EF Core, some of which duplicate each other, that it isn't always obvious which of the three approaches you should use for each part of the configuration. Here are suggested approaches to use for each part of EF Core configuration:

- Start by using the By Convention approach wherever possible, because it's quick and easy.
- Use the validation attributes—`MaxLength`, `Required`, and so on—from the Data Annotations approach, as they're useful for validation.
- For everything else, use the Fluent API approach, because it has the most comprehensive set of commands. But consider writing code to automate common settings, such as applying the `DateTime` "UTC fix" to all `DateTime` properties whose `Name` ends with `"Utc"`.

The following sections provide more-detailed explanations of my recommendations for configuring EF Core.

7.16.1 Use By Convention configuration first

EF Core does a respectable job of configuring most standard properties, so always start with that approach. In part 1, you built the whole of this initial database by using the By Convention approach, apart from the composite key in the `BookAuthor` many-to-many linking entity class.

The By Convention approach is quick and easy. You'll see in chapter 8 that most relationships can be set up purely by using the By Convention naming rules, which can save you a lot of time. Learning what By Convention can configure will dramatically reduce the amount of configuration code you need to write.

7.16.2 Use validation Data Annotations wherever possible

Although you can do things such as limit the size of a string property with either Data Annotations or the Fluent API, I recommend using Data Annotations for the following reasons:

- *Frontend validation can use them.* Although EF Core doesn't validate the entity class before saving it to the database, other parts of the system may use Data Annotations for validation. ASP.NET Core uses Data Annotations to validate input, for example, so if you input directly into an entity class, the validation attributes will be useful. Or if you use separate ASP.NET ViewModel or DTO classes, you can cut and paste the properties with their validation attributes.
- *You may want to add validation to EF Core's* `SaveChanges`. Using data validation to move checks out of your business logic can make your business logic simpler. Chapter 4 showed you how to add validation of entity classes when `SaveChanges` is called.
- *Data Annotations make great comments.* Attributes, which include Data Annotations, are compile-time constants; they're easy to see and easy to understand.

7.16.3 Use the Fluent API for anything else

Typically, I use the Fluent API for setting up the database column mapping (column name, column data type, and so on) when it differs from the conventional values. You could use the schema Data Annotations to do that, but I try to hide things like these inside the `OnModelCreating` method because they're database implementation issues rather than software structure issues. That practice is more a preference than a rule, though, so make your own decision. Section 7.16.4 describes how to automate some of your Fluent API configurations, which saves you time and also ensures that all your configuration rules are applied to every matching class/property.

7.16.4 Automate adding Fluent API commands by class/property signatures

One useful feature of the Fluent API commands allows you to write code to find and configure certain configurations based on the class/property type, name, and so on. In a real application, you might have hundreds of `DateTime` properties that need the UTC fix you used in listing 7.6. Rather than add the configuration for each property by hand, wouldn't it be nice to find each property that needs the UTC fix and apply it automatically? You're going to do exactly that.

Automating finding/adding configurations relies on a type called `IMutableModel`, which you can access in the `OnModelCreating` method. This type gives you access to all the classes mapped by EF Core to the database, and each `IMutableEntityType` allows you to access the properties. Most configuration options can be applied via methods in these two interfaces, but a few, such as Query Filters, need a bit more work.

To start, you will build the code that will iterate through each entity class and its properties, and add one configuration, as shown in listing 7.13. This iteration approach defines the way to automate configurations, and in later examples, you will add extra commands to do more configurations.

The following example adds a value converter to a `DateTime` that applies the UTC fix shown in listing 7.6. But in the following listing, the UTC fix value converter is applied to every property that is a `DateTime` with a `Name` that ends with `"Utc"`.

Listing 7.13 Applying value converter to any `DateTime` property ending in `"Utc"`

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    var utcConverter = new ValueConverter<DateTime, DateTime>(
        toDb => toDb,
        fromDb =>
            DateTime.SpecifyKind(fromDb, DateTimeKind.Utc));

    foreach (var entityType in modelBuilder.Model.GetEntityTypes())
    {
```

The Fluent API commands are applied in the OnModelCreating method.

Defines a value converter to set the UTC setting to the returned DateTime

Loops through all the classes that EF Core has currently found mapped to the database

Loops through all the properties in an entity class that are mapped to the database

```
        foreach (var entityProperty in entityType.GetProperties())
        {
            if (entityProperty.ClrType == typeof(DateTime)
                && entityProperty.Name.EndsWith("Utc"))
            {
                entityProperty.SetValueConverter(utcConverter);
            }
            //… other examples left out for clarity
        }
    }
    //… rest of configration code left out
```

Adds the UTC value converter to properties of type DateTime and Name ending in "Utc"

Listing 7.13 showed the setup of only one `Type/Named` property, but normally, you would have lots of Fluent API settings. In this example, you are going to do the following:

1. Add the UTC fix value converter to properties of type `DateTime` whose `Names` end with `"Utc"`.
2. Set the decimal precision/scale where the property's `Name` contains `"Price"`.
3. Set any string properties whose `Name` ends in `"Url"` to be stored as ASCII—that is, `varchar(nnn)`.

The following code snippet shows the code inside the `OnModelCreating` method in the Book App DbContext to add these three configuration settings:

```
foreach (var entityType in modelBuilder.Model.GetEntityTypes())
{
    foreach (var entityProperty in entityType.GetProperties())
    {
        if (entityProperty.ClrType == typeof(DateTime)
            && entityProperty.Name.EndsWith("Utc"))
        {
            entityProperty.SetValueConverter(utcConverter);
        }

        if (entityProperty.ClrType == typeof(decimal)
            && entityProperty.Name.Contains("Price"))
        {
            entityProperty.SetPrecision(9);
            entityProperty.SetScale(2);
        }

        if (entityProperty.ClrType == typeof(string)
            && entityProperty.Name.EndsWith("Url"))
        {
            entityProperty.SetIsUnicode(false);
        }
    }
}
```

A few Fluent APIs configurations need class-specific code, however. The Query Filters, for example, need a query that accesses entity classes. For this case, you need to add an interface to the entity class you want to add a Query Filter to and create the correct filter query dynamically.

As an example, you are going to build code that allows you to add automatically the `SoftDelete` Query Filter described in section 3.5.1 and the `UserId` Query Filter shown in section 6.1.7. Of these two Query Filters, `UserId` is more complex because it needs to get the current `UserId`, which changes on every instance of the Book App's DbContext. You can do this in a couple of ways, but you decide to provide the current instance of the DbContext to the query. The following listing shows the extension class, called `SoftDeleteQueryExtensions`, with its `MyQueryFilterTypes` `enum`.

Listing 7.14 The `enum`/class to use to set up Query Filters on every compatible class

Third optional property holds a copy of the current DbContext instance so that the UserId will be the current one

Second parameter allows you to pick which type of query filter to add

Defines the different type of LINQ query to put in the Query Filter

A static extension class

Call this method to set up the query filter.

First parameter comes from EF Core and allows you to add a query filter

```
public enum MyQueryFilterTypes { SoftDelete, UserId }

public static class SoftDeleteQueryExtensions
{
    public static void AddSoftDeleteQueryFilter(
        this IMutableEntityType entityData,
        MyQueryFilterTypes queryFilterType,
        IUserId userIdProvider = null)
    {
        var methodName = $"Get{queryFilterType}Filter";
        var methodToCall = typeof(SoftDeleteQueryExtensions)
            .GetMethod(methodName,
                BindingFlags.NonPublic | BindingFlags.Static)
            .MakeGenericMethod(entityData.ClrType);
        var filter = methodToCall
            .Invoke(null, new object[] { userIdProvider });
        entityData.SetQueryFilter((LambdaExpression)filter);
        if (queryFilterType == MyQueryFilterTypes.SoftDelete)
            entityData.AddIndex(entityData.FindProperty(
                nameof(ISoftDelete.SoftDeleted)));
        if (queryFilterType == MyQueryFilterTypes.UserId)
            entityData.AddIndex(entityData.FindProperty(
                nameof(IUserId.UserId)));
    }

    private static LambdaExpression GetUserIdFilter<TEntity>(
        IUserId userIdProvider)
        where TEntity : class, IUserId
    {
        Expression<Func<TEntity, bool>> filter =
            x => x.UserId == userIdProvider.UserId;
        return filter;
    }
```

Creates the correctly typed method to create the Where LINQ expression to use in the Query Filter

Uses the filter returned by the created type method in the SetQueryFilter method

Adds an index on the SoftDeleted property for better performance

Adds an index on the UserId property for better performance

Creates a query that is true only if the _userId matches the UserID in the entity class

Creates a query that is true only if the SoftDeleted property is false

```
    private static LambdaExpression GetSoftDeleteFilter<TEntity>(
        IUserId userIdProvider)
        where TEntity : class, ISoftDelete
    {
        Expression<Func<TEntity, bool>> filter =
            x => !x.SoftDeleted;
        return filter;
    }
}
```

Because every query of an entity that has a Query Filter will contain a filter on that property, the code automatically adds an index on every property that is used in a Query Filter. That technique improves performance on that entity. Finally, the following listing shows how to use the code shown in listing 7.14 within the Book App's DbContext to automate the configuration of the Query Filters.

Listing 7.15 Adding code to the DbContext to automate setting up Query Filters

Holds the UserId, which is used in the Query Filter that uses the IUserId interface

Adding the IUserId to the DbContext means that we can pass the DbContext to the UserId query filter.

```
public class EfCoreContext : DbContext, IUserId
{
    public Guid UserId { get; private set; }
```

Sets up the UserId. If the userIdService is null, or if it returns null for the UserId, we set a replacement UserId.

```
    public EfCoreContext(DbContextOptions<EfCoreContext> options,
        IUserIdService userIdService = null)
        : base(options)
    {
        UserId = userIdService?.GetUserId()
                 ?? new ReplacementUserIdService().GetUserId();
    }

    //DbSets removed for clarity
```

The automate code goes in the OnModelCreating method.

Loops through all the classes that EF Core has currently found mapped to the database

```
    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
        //other configration code removed for clarity

        foreach (var entityType in modelBuilder.Model.GetEntityTypes()
        {
            //other property code removed for clarity
```

If the class inherits the ISoftDelete interface, it needs the SoftDelete Query Filter.

Adds a Query Filter to this class, with a query suitable for SoftDelete

```
            if (typeof(ISoftDelete)
                .IsAssignableFrom(entityType.ClrType))
            {
                entityType.AddSoftDeleteQueryFilter(
                    MyQueryFilterTypes.SoftDelete);
            }
```

If the class inherits the IUserId interface, it needs the IUserId Query Filter.

```
            if (typeof(IUserId)
                .IsAssignableFrom(entityType.ClrType))
            {
```

```
                entityType.AddSoftDeleteQueryFilter(
                    MyQueryFilterTypes.UserId, this);
            }
        }
}
```

Adds the UserId Query Filter to this class. Passing 'this' allows access to the current UserId.

For the Book App, all this automation is overkill, but in bigger applications, it can save you a great deal of time; more important, it ensures that you have set everything up correctly. To end this section, here are some recommendations and limitations that you should know about if you are going to use this approach:

- If you run the automatic Fluent API code before your handcoded configurations, your handcoded configurations will override any of the automatic Fluent API settings. But be aware that if there is an entity class that is registered only via manually written Fluent API, that entity class won't be seen by the automatic Fluent API code.
- The configuration commands must apply the same configurations every time because the EF Core configures the application's DbContext only once—on first use—and then works from a cache version.

Summary

- The first time you create the application's DbContext, EF Core configures itself by using a combination of three approaches: By Convention, Data Annotations, and the Fluent API.
- Value converters allow you to transform the software type/value when writing and reading back from the database.
- Two EF Core features, shadow properties and backing fields, allow you to hide data from higher levels of your code and/or control access to data in an entity class. Use the By Convention approach to set up as much as you can, because it's simple and quick to code.
- When the By Convention approach doesn't fit your needs, Data Annotations and/or EF Core's Fluent API can provide extra commands to configure both the way EF Core maps the entity classes to the database and the way EF Core will handle that data.
- In addition to writing configuration code manually, you can also add code to configure entity classes and/or properties automatically based on the class/properties signature.

For readers who are familiar with EF6:

- The basic process of configuring EF Core is, on the surface, similar to the way EF6 works, but there is a significant number of changed or new commands.
- EF Core can use configuration classes to hold the Fluent API commands for a given entity class. The Fluent API commands provide a feature similar to the

EF6.x `EntityType-Configuration<T>` class, but EF Core uses an `IEntityType-Configuration<T>` interface instead.

- EF Core has introduced many extra features that are not available in EF6, such as value converters, shadow properties, and backing fields, all of which are welcome additions to EF.

8

Configuring relationships

This chapter covers

- Configuring relationships with By Convention
- Configuring relationships with Data Annotations
- Configuring relationships with the Fluent API
- Mapping entities to database tables in five other ways

Chapter 7 described how to configure scalar (nonrelational) properties. This chapter covers how to configure database relationships. I assume that you've read at least the first part of chapter 7, because configuring relationships uses the same three approaches—By Convention, Data Annotations, and the Fluent API—to map the database relationships.

This chapter covers how EF Core finds and configures relationships between entity classes, with pointers and examples showing how to configure each type of relationship: one-to-one, one-to-many, and many-to-many. EF Core's By Convention relationship rules can configure many relationships quickly, but you'll also learn about all the Data Annotations and Fluent API configuration options, which allow you to define precisely the way you want a relationship to behave. You'll also look at features that allow you to enhance your relationships with extra keys and alternative table-mapping approaches. Finally, you'll consider five ways to map your classes to the database.

8.1 Defining some relationship terms

This chapter refers to the various parts of a relationship, and you need clear terms so that you know exactly what part of the relationship we're talking about. Figure 8.1 shows those terms, using the `Book` and `Review` entity classes from our Book App. I follow this figure with a more detailed description so the terms will make sense to you when I use them in this chapter.

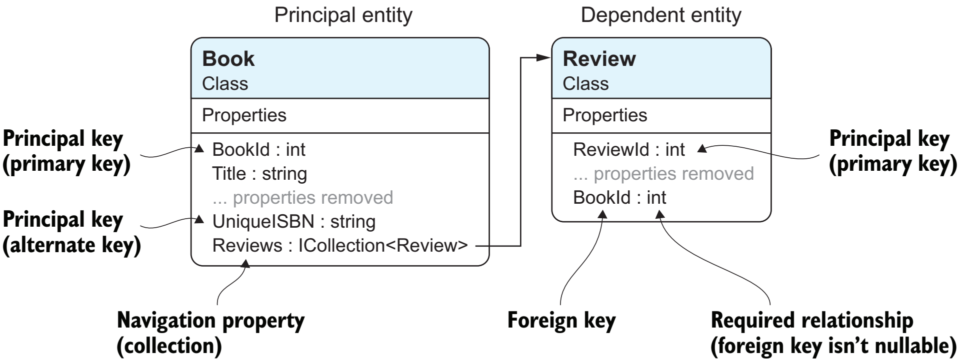


Figure 8.1 The `Book` and `Review` entity classes show six of the terms used in this chapter to discuss relationships: *principal entity*, *dependent entity*, *principal key*, *navigational property*, *foreign key*, and *required relationship*. Not shown is *optional relationship*, which is described in section 2.1.1.

To ensure that these terms are clear, here are detailed descriptions:

- *Principal key*—A new term, taken from EF Core's documentation, that refers to either the primary key, defined in part 1, or the new *alternate key*, which has a unique value per row and isn't the primary key (see section 8.8.3)

NOTE Figure 8.1 provides an example of an alternate key called `UniqueISBN`, which represents a unique value per entity. (*ISBN* stands for *International Standard Book Number*, which is unique for every book.)

- *Principal entity*—The entity that contains the principal-key properties, which the dependent relationship refer to via a foreign key(s) (covered in chapter 3)
- *Dependent entity*—The entity that contains the foreign-key properties that refer to the principal entity (covered in chapter 3)
- *Principal key*—The entity has a principal key, also known as the *primary key*, which is unique for each entity stored in the database

- *Navigational property*—A term taken from EF Core's documentation that refers to the property containing a single entity class, or a collection of entity classes, that EF Core uses to link entity classes
- *Foreign key*—Defined in section 2.1.3, holds the principal key value(s) of the database row it's linked to (or could be `null`)
- *Required relationship*—A relationship in which the foreign key is non-nullable (and principal entity must exist)
- *Optional relationship*—A relationship in which the foreign key is nullable (and principal entity can be missing)

NOTE A principal key and a foreign key can consist of more than one property/column. These keys are called *composite keys*. You've already seen one of these keys in section 3.4.4, as the `BookAuthor` many-to-many linking entity class has a composite primary key consisting of the `BookId` and the `AuthorId`.

You'll see in section 8.4 that EF Core can find and configure most relationships By Convention. In some cases, EF Core needs help, but generally, it can find and configure your navigational properties for you if you use the By Convention naming rules.

8.2 What navigational properties do you need?

The configuring of relationships between entity classes should be guided by the business needs of your project. You could add navigational properties at both ends of a relationship, but that suggests that every navigational property is useful, and some navigational properties aren't. It is good practice to provide only navigational properties that make sense from the business or software design point of view.

In our Book App, for example, the `Book` entity class has many `Review` entity classes, and each `Review` class is linked, via a foreign key, to one `Book`. Therefore, you could have a navigational property of type `ICollection<Review>` in the `Book` class and a navigational property of type `Book` in the `Review` class. In that case, you'd have a *fully defined relationship*: a relationship with navigational properties at both ends.

But do you need a fully defined relationship? From the software design point of view, there are two questions about the `Book`/`Review` navigational relationships. The answers to these questions define which navigational relationship you need to include:

- Does the `Book` entity class need to know about the `Review` entity classes? I say yes, because we want to calculate the average review score.
- Does the `Review` entity class need to know about the `Book` entity class? I say no, because in this example application, we don't do anything with that relationship.

Our solution, therefore, is to have only the `ICollection<Review>` navigational property in the `Book` class, which is what figure 8.1 portrays.

My experience is you should add a navigational property only when it makes sense from a business point of view or when you need a navigational property to create (EF Core's `Add`) an entity class with a relationship (see section 6.2.1). Minimizing navigational properties will help make the entity classes easier to understand, and more-junior developers won't be tempted to use relationships that aren't right for your project.

8.3 Configuring relationships

In the same way as in chapter 7, which covered configuring nonrelational properties, EF Core has three ways to configure relationships. Here are the three approaches for configuring properties, but focused on relationships:

- *By Convention*—EF Core finds and configures relationships by looking for references to classes that have a primary key in them.
- *Data Annotations*—These annotations can be used to mark foreign keys and relationship references.
- *Fluent API*—This API provides the richest set of commands to configure any relationship fully.

The next three sections detail each of these approaches in turn. As you'll see, the By Convention approach can autoconfigure many relationships for you if you follow its naming standards. At the other end of the scale, the Fluent API allows you to define every part of a relationship manually, which can be useful if you have a relationship that falls outside the By Convention approach.

8.4 Configuring relationships By Convention

The By Convention approach is a real time-saver when it comes to configuring relationships. In EF6.x, I used to define my relationships laboriously because I didn't fully understand the power of the By Convention approach to relationships. Now that I understand the conventions, I let EF Core set up most of my relationships, other than in the few cases in which By Convention doesn't work. (Section 8.4.6 lists those exceptions.)

The rules are straightforward, but the ways that the property name, type, and nullability work together to define a relationship take a bit of time to absorb. I hope that reading this section will save you time when you're developing your next application that uses EF Core.

8.4.1 What makes a class an entity class?

Chapter 2 defined the term *entity class* as a normal .NET class that has been mapped by EF Core to the database. Here, you want to define how EF Core finds and identifies a class as an entity class by using the By Convention approach.

Figure 7.1 showed the three ways that EF Core configures itself. Following is a recap of that process, now focused on finding the relationships and navigational properties:

1. EF Core scans the application's DbContext, looking for any public `DbSet<T>` properties. It assumes that the classes, `T`, in the `DbSet<T>` properties are entity classes.

2. EF Core also looks at every public property in the classes found in step 1 and looks at properties that could be navigational properties. The properties whose type contains a class that isn't defined as being scalar properties (`string` is a class, but it's defined as a scalar property) are assumed to be navigational properties. These properties may appear as a single link (such as `public PriceOffer Promotion ( get; set; }`) or a type that implements the `IEnumerable<T>` interface (such as `public ICollection<Review> Reviews { get; set; }`).
3. EF Core checks whether each of these entity classes has a primary key (see section 7.9). If the class doesn't have a primary key and hasn't been configured as not having a key (see section 7.9.3), or if the class isn't excluded, EF Core will throw an exception.

8.4.2 An example of an entity class with navigational properties

Listing 8.1 shows the entity class `Book`, which is defined in the application's DbContext. In this case, you have a public property of type `DbSet<Book>`, which passed the "must have a valid primary key" test in that it has a public property called `BookId`.

What you're interested in is how EF Core's By Convention configuration handles the three navigational properties at the bottom of the class. As you'll see in this section, EF Core can work out which sort of relationship it is by the type of the navigational property and the foreign key in the class that the navigational property refers to.

Listing 8.1 The `Book` entity class, with the relationships at the bottom

```
public class Book
{
    public int BookId { get; set; }
    //other scalar properties removed as not relevant…

    public PriceOffer Promotion { get; set; }

    public ICollection<Tag> Tags { get; set; }

    public ICollection<BookAuthor> AuthorsLink { get; set; }

    public ICollection<Review> Reviews { get; set; }
}
```

Links to a PriceOffer, which is one-to-zero-or-one relationship

Links directly to a list of Tag entities, using EF Core 5's automatic many-to-many relationship

Links to one side of the many-to-many relationship of authors via a linking table

Links to any reviews for this book: one-to-many relationship

If two navigational properties exist between the two entity classes, the relationship is known as *fully defined*, and EF Core can work out By Convention whether it's a one-to-one or a one-to-many relationship. If only one navigational property exists, EF Core can't be sure, so it assumes a one-to-many relationship.

Certain one-to-one relationships may need configuration via the Fluent API if you have only one navigational property or if you want to change the default By Convention setting, such as when you're deleting an entity class with a relationship.

8.4.3 How EF Core finds foreign keys By Convention

A foreign key must match the principal key (defined in section 8.1) in type and in name, but to handle a few scenarios, foreign-key name matching has three options, shown in figure 8.2. The figure shows all three options for a foreign-key name using the entity class `Review` that references the primary key, `BookId`, in the entity class `Book`.

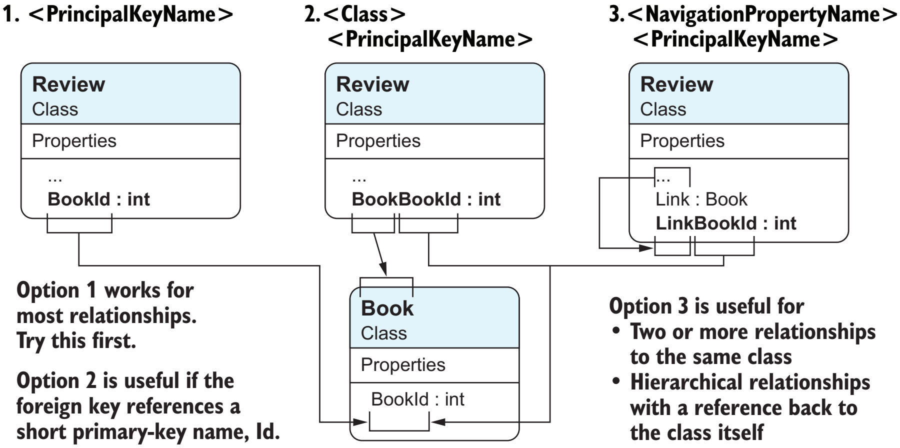


Figure 8.2 Three By Convention options for a foreign key referring to the `Book` entity class's primary key. These options allow you to use a unique name for your foreign key, from which EF Core can work out which primary key this relationship refers to.

Option 1 is the one I use most; it's depicted in figure 8.1. Option 2 is for developers who use the short, By Convention primary-key name, `Id`, as it makes the foreign key unique to the class it's linking to. Option 3 helps with specific cases in which you'd get duplicate named properties if you used option 1. The following listing shows an example of using option 3 to handle a hierarchical relationship.

Listing 8.2 A hierarchical relationship with an option-3 foreign key

```
public class Employee
{
    public int EmployeeId { get; set; }

    public string Name { get; set; }

    //------------------------------
    //Relationships

    public int? ManagerEmployeeId { get; set; }
    public Employee Manager { get; set; }
}
```

Foreign key uses the <NavigationalPropertyName><PrimaryKeyName> pattern

The entity class called `Employee` has a navigational property called `Manager` that links to the employee's manager, who is an employee as well. You can't use a foreign key of `EmployeeId` (option 1), because it's already used for the primary key. Therefore, you use option 3 and call the foreign key `ManagerEmployeeId` by using the navigational property name at the start.

8.4.4 Nullability of foreign keys: Required or optional dependent relationships

The nullability of the foreign key defines whether the relationship is required (non-nullable foreign key) or optional (nullable foreign key). A *required relationship* ensures that relationships exist by ensuring that the foreign key is linked to a valid principal key. Section 8.6.1 describes an `Attendee` entity that has a required relationship to a `Ticket` entity class.

An *optional* relationship allows there to be no link between the principal entity and the dependent entity by having the foreign-key value(s) set to `null`. The `Manager` navigational property in the `Employee` entity class, shown in listing 8.2, is an example of an optional relationship, as someone at the top of the business hierarchy won't have a boss.

The required or optional status of the relationship also affects what happens to dependent entities when the principal entity is deleted. The default setting of the `OnDelete` action for each relationship type is as follows:

- For a *required relationship,* EF Core sets the `OnDelete` action to `Cascade`. If the principal entity is deleted, the dependent entity will be deleted too.
- For a *optional relationship,* EF Core sets the `OnDelete` action to `ClientSetNull`. If the dependent entity is being tracked, the foreign key will be set to `null` when the principal entity is deleted. But if the dependent entity *isn't* being tracked, the database constraint delete setting takes over, and the `ClientSetNull` setting sets the database rules as though the `Restrict` setting were in place. The result is that the delete fails at the database level, and an exception is thrown.

NOTE The `ClientSetNull` delete behavior is rather unusual, and section 8.8.1 explains why. That section also describes how to configure the delete behavior of a relationship.

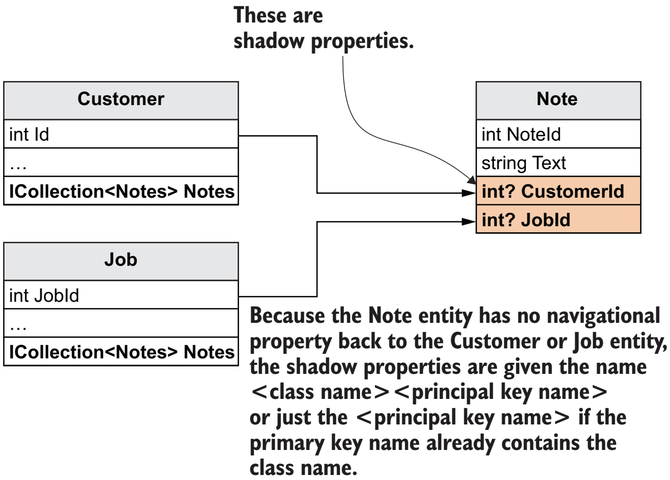
8.4.5 Foreign keys: What happens if you leave them out?

If EF Core finds a relationship via a navigational property or through a relationship you configured via the Fluent API, it needs a foreign key to set up the relationship in the relational database. Including foreign keys in your entity classes is good practice, giving you better control of the nullability of the foreign key. Also, access to foreign keys can be useful when you're handling relationships in a disconnected update (see section 3.3.1).

But if you do leave out a foreign key (on purpose or by accident), EF Core configuration will add a foreign key as a shadow property. *Shadow properties,* which were introduced in chapter 7, are hidden properties that can be accessed only via specific EF

Core commands. Having foreign keys added automatically as shadow properties can be useful. One of my clients, for example, had a general `Note` entity class that was added to a `Notes` collection in many entities.

Figure 8.3 shows a one-to-many relationship in which the `Note` entity class is used in a collection navigational property in two entity classes: `Customer` and `Job`. Note that the primary-key names of the `Customer` and `Job` entity classes use different By Convention naming approaches to show how the shadow properties are named.



These are shadow properties.

Because the Note entity has no navigational property back to the Customer or Job entity, the shadow properties are given the name <class name><principal key name> or just the <principal key name> if the primary key name already contains the class name.

Figure 8.3 EF Core's By Convention configuration will add nullable (that is, optional relationship) foreign keys as shadow properties if you don't provide your own foreign keys in the `Notes` entity class.

If the entity class that gains a shadow property foreign key has a navigational link to the other end of the relationship, the name of that shadow property would be `<navigation property name><principal key property name>`. If the `Note` entity in figure 8.3 has a navigational link back to the Customer entity called `LinkBack`, the shadow property foreign key's name would be `LinkBackId`.

> **NOTE** My unit tests show that one-to-one relationships are rejected if there is no foreign key to link the two entities. Therefore, EF Core's By Convention won't set up shadow property foreign keys on one-to-one relationships automatically.

If you want to add a foreign key as a shadow property, you can do that via the Fluent API `HasForeignKey`, shown in section 8.6, but with the name of the shadow property name provided via a string. Be careful not to use the name of an existing property, as that will not add a shadow property but will use the existing property.

The shadow foreign-key property will be nullable, which has the effect described in section 8.4.4 on nullability of foreign keys. If this effect isn't what you want, you can alter the shadow property's nullability by using the Fluent API `IsRequired` method, as described in section 8.8.2.

> **EF6** EF6.x uses a similar approach to adding foreign keys if you left them out of your entity classes, but in EF6.x, you can't configure the nullability or access the content. EF Core's shadow properties make leaving out foreign keys more controllable.

8.4.6 When does By Convention configuration not work?

If you're going to use the By Convention configuration approach, you need to know when it's not going to work so that you can use other means to configure your relationship. Here's my list of scenarios that won't work, with the most common listed first:

- You have composite foreign keys (see section 8.6 or section 8.5.1).
- You want to create a one-to-one relationship without navigational links going both ways (see section 8.6.1).
- You want to override the default delete-behavior setting (see section 8.8.1).
- You have two navigational properties going to the same class (see section 8.5.2).
- You want to define a specific database constraint (see section 8.8.4).

8.5 Configuring relationships by using Data Annotations

Only two Data Annotations relate to relationships, as most of the navigational configuration is done via the Fluent API: the `ForeignKey` and `InverseProperty` annotations.

8.5.1 The ForeignKey Data Annotation

The `ForeignKey` Data Annotation allows you to define the foreign key for a navigational property in the class. Taking the hierarchical example of the `Employee` class, you can use this annotation to define the foreign key for the `Manager` navigational property. The following listing shows an updated `Employee` entity class with a new, shorter foreign-key name for the `Manager` navigational property that doesn't fit By Convention naming: `ManagerEmployeeId`.

Listing 8.3 Using the `ForeignKey` data annotation to set the foreign-key name

```
public class Employee
{
    public int EmployeeId { get; set; }
    public string Name { get; set; }

    public int? ManagerId { get; set; }
    [ForeignKey(nameof(ManagerId))]
    public Employee Manager { get; set; }
}
```

Defines which property is the foreign key for the Manager navigational property

NOTE You've applied the `ForeignKey` data annotation to the `Manager` navigational property, giving the name of the foreign key, `ManagerId`. But the `ForeignKey` data annotation also works the other way around. You could've applied the `ForeignKey` data annotation to the foreign-key property, `ManagerId`, giving the name of the navigational property, `Manager`—such as `[ForeignKey(nameof(Manager))]`.

The `ForeignKey` data annotation takes one parameter, which is a string. This string should hold the name of the foreign-key property. If the foreign key is a composite key (has more than one property), it should be comma-delimited—as in `[ForeignKey("Property1, Property2")]`.

TIP I suggest that you use the `nameof` keyword to provide the property name string. That's safer, because if you change the name of the foreign-key property, `nameof` will either be updated at the same time or throw a compile error if you forgot to change all the references.

8.5.2 The InverseProperty Data Annotation

The `InverseProperty` Data Annotation is a rather specialized Data Annotation for use when you have two navigational properties going to the same class. At that point, EF Core can't work out which foreign keys relate to which navigational property. This situation is best shown in code. The following listing shows an example `Person` entity class with two lists: one for books owned by the librarian and one for `Books` out on loan to a specific person.

Listing 8.4 `LibraryBook` entity class with two relationships to `Person` class

```
public class LibraryBook
{
    public int LibraryBookId { get; set; }
    public string Title { get; set; }

    public int LibrarianPersonId { get; set; }
    public Person Librarian { get; set; }

    public int? OnLoanToPersonId { get; set; }
    public Person OnLoanTo { get; set; }
}
```

The `Librarian` and the borrower of the book (`OnLoanTo` navigational property) are both represented by the `Person` entity class. The `Librarian` navigational property and the `OnLoanTo` navigational property both link to the same class, and EF Core can't set up the navigational linking without help. The `InverseProperty` Data Annotation shown in the following listing provides the information to EF Core when it's configuring the navigational links.

Listing 8.5 The `Person` entity class, which uses the `InverseProperty` annotation

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }

    [InverseProperty("Librarian")]
    public ICollection<LibraryBook>
        LibrarianBooks { get; set; }

    [InverseProperty("OnLoanTo")]
    public ICollection<LibraryBook>
        BooksBorrowedByMe { get; set; }
}
```

Links LibrarianBooks to the Librarian navigational property in the LibraryBook class

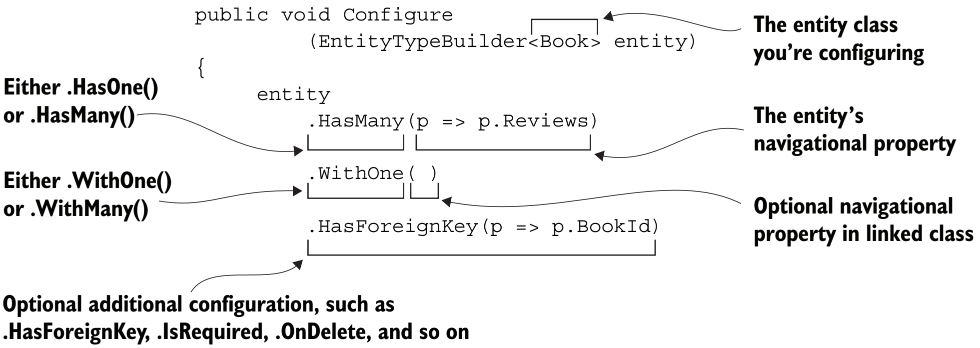
Links the BooksBorrowedByMe list to the OnLoanTo navigational property in the LibraryBook class

This code is one of those configuration options that you rarely use, but if you have this situation, you must either use it or define the relationship with the Fluent API. Otherwise, EF Core will throw an exception when it starts, as it can't work out how to configure the relationships.

8.6 Fluent API relationship configuration commands

As I said in section 8.4, you can configure most of your relationships by using EF Core's By Convention approach. But if you want to configure a relationship, the Fluent API has a well-designed set of commands that cover all the possible combinations of relationships. It also has extra commands that allow you to define other database constraints. Figure 8.4 shows the format for defining a relationship with the Fluent API. All Fluent API relationship configuration commands follow this pattern.

> **EF6** EF Core's Fluent API command names have changed from EF6, and for me, they're much clearer. I found EF6's `WithRequired` and `WithRequiredPrincipal`/`WithRequiredDependent` commands to be a bit confusing, whereas



```
public void Configure
        (EntityTypeBuilder<Book> entity)
{
    entity
        .HasMany(p => p.Reviews)
        .WithOne( )
        .HasForeignKey(p => p.BookId)
```

The entity class you're configuring

Either .HasOne() or .HasMany()

The entity's navigational property

Either .WithOne() or .WithMany()

Optional navigational property in linked class

Optional additional configuration, such as .HasForeignKey, .IsRequired, .OnDelete, and so on

Figure 8.4 The Fluent API allows you to define a relationship between two entity classes. `HasOne`/`HasMany` and `WithOne`/`WithMany` are the two main parts, followed by other commands to specify other parts or set certain features.

the EF Core Fluent API commands have a clearer `HasOne`/`HasMany` followed by `WithOne`/`WithMany` syntax.

Next, we'll define one-to-one, one-to-many, and many-to-many relationships to illustrate the use of these Fluent API relationships.

8.6.1 Creating a one-to-one relationship

One-to-one relationships can get a little complicated because there are three ways to build them in a relational database. To understand these options, you'll look at an example in which you have attendees (entity class `Attendee`) at a software convention, and each attendee has a unique ticket (entity class `Ticket`).

Chapter 3 showed how to create, update, and delete relationships. To recap, here's a code snippet showing how to create a one-to-one relationship:

```
var attendee = new Attendee
{
    Name = "Person1",
    Ticket = new Ticket{ TicketType = TicketTypes.VIP}
};
context.Add(attendee);
context.SaveChanges();
```

Figure 8.5 shows the three options for building this sort of one-to-one relationship. The principal entities are at the top of the diagram, and the dependent entities are at the bottom. Note that option 1 has the `Attendee` as the dependent entity, whereas options 2 and 3 have the `Ticket` as the dependent entity.

Each option has advantages and disadvantages. You should use the one that's right for your business needs.

Option 1 is the standard approach to building one-to-one relationships, because it allows you to define that the one-to-one dependent entity is required (must be present). In our example, an exception will be thrown if you try to save an `Attendee` entity instance without a unique `Ticket` attached to it. Figure 8.6 shows option 1 in more detail.

With the option-1 one-to-one arrangement, you can make the dependent entity optional by making the foreign key nullable. Also, in figure 8.6, you can see that the `WithOne` method has a parameter that picks out the `Attendee` navigational property in the `Ticket` entity class that links back to the `Attendee` entity class. Because the `Attendee` class is the dependent part of the relationship, if you delete the `Attendee` entity, the linked `Ticket` won't be deleted, because the `Ticket` is the principal entity in the relationship. The downside of option 1 in this example is that it allows one `Ticket` to be used for multiple `Attendees`, which doesn't match the business rules I stated at the start. Finally, this option allows you to replace `Ticket` with another `Ticket` instance by assigning a new `Ticket` to the Attendee's `Ticket` navigational property.

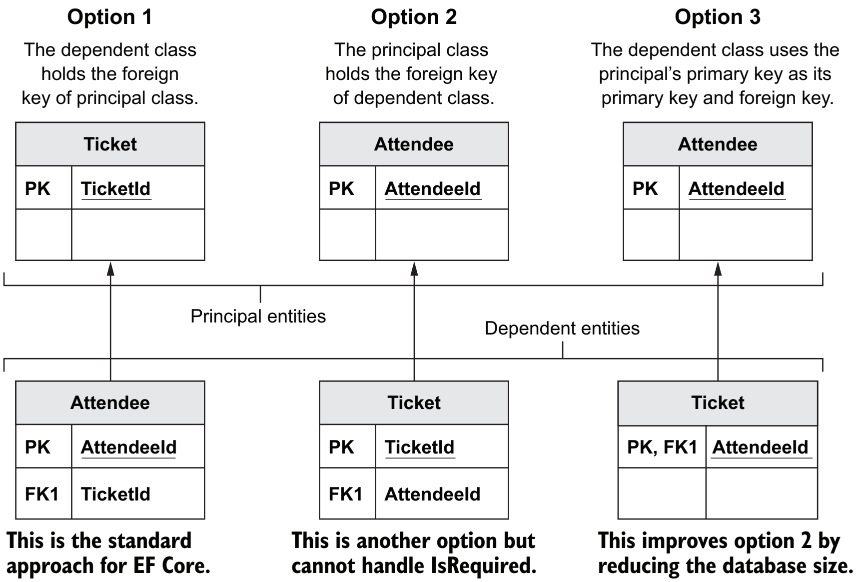
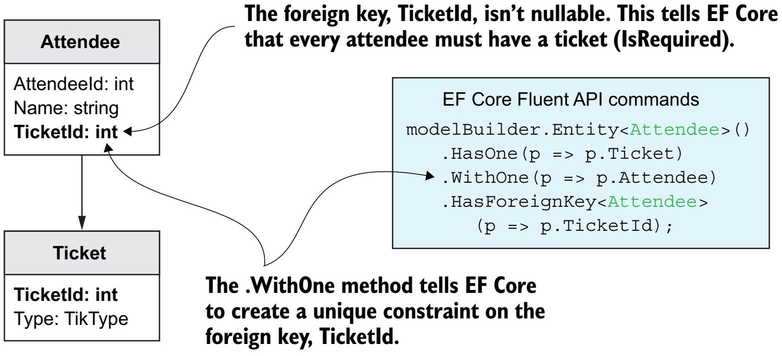


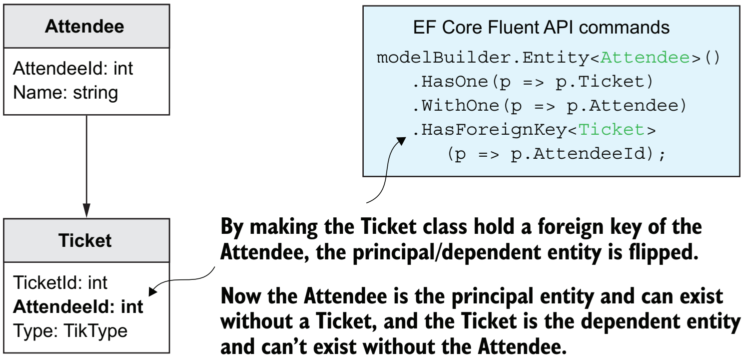
Figure 8.5 The three ways of defining a one-to-one relationship in a relational database; comments at the bottom indicate EF Core's handling of each approach. Option 1 is different from options 2 and 3 in that the order of the two ends of the one-to-one relationship are swapped, which changes which part can be forced to exist. In option 1, the `Attendee` must have a `Ticket`, whereas in options 2 and 3, the `Ticket` is optional for the `Attendee`. Also, if the principal entity (top row) is deleted, the dependent entity (bottom row) will be deleted too.



```
modelBuilder.Entity<Attendee>()
    .HasOne(p => p.Ticket)
    .WithOne(p => p.Attendee)
    .HasForeignKey<Attendee>
        (p => p.TicketId);
```

Figure 8.6 The non-nullable foreign key ensures that the principal entity (in this case, `Attendee`) must have a dependent, one-to-one entity, `Ticket`. Also, configuring the relationship as one-to-one ensures that each dependent entity, `Ticket`, is unique. Notice that the Fluent API on the right has navigational properties going both ways; each entity has a navigational property going to the other.

Options 2 and 3 in figure 8.5 turn the principal/dependent relationship around, with the `Attendee` becoming the principal entity in the relationship. This situation swaps the required/optional nature of the relationship. Now the `Attendee` can exist without the `Ticket`, but the `Ticket` can't exist without the `Attendee`. Options 2 and 3 do enforce the assignment of a `Ticket` to only one `Attendee`, but replacing `Ticket` with another `Ticket` instance requires you to delete the old ticket first. Figure 8.7 shows this relationship.



```
modelBuilder.Entity<Attendee>()
   .HasOne(p => p.Ticket)
   .WithOne(p => p.Attendee)
   .HasForeignKey<Ticket>
      (p => p.AttendeeId);
```

By making the Ticket class hold a foreign key of the Attendee, the principal/dependent entity is flipped.

Now the Attendee is the principal entity and can exist without a Ticket, and the Ticket is the dependent entity and can't exist without the Attendee.

Figure 8.7 Option 2: The `Ticket` entity holds the foreign key of the `Attendee` entity, changing which entity is the principal and which is the dependent entity. In this case, the `Attendee` is now the principal entity, and the `Ticket` is the dependent entity.

Option 2 and 3 are useful because they form optional one-to-one relationships, often referred to as *one-to-zero-or-one relationships*. Option 3 is a more efficient way to define option 2, with the primary key and the foreign key combined. I would've used option 3 for the `PriceOffer` entity class in the Book App, but I wanted to start with the simpler option-2 approach. Another, even better version uses an `Owned` type (see section 8.9.1) because it is automatically loaded from the same table, which is safer (I can't forget to add the `Include`) and more efficient.

8.6.2 Creating a one-to-many relationship

One-to-many relationships are simpler, because there's one format: the many entities contain the foreign-key value. You can define most one-to-many relationships with the By Convention approach simply by giving the foreign key in the many entities a name that follows the By Convention approach (see section 8.4.3). But if you want to define a relationship, you can use the Fluent API, which gives you complete control of how the relationship is set up. Figure 8.8 provides an example of the Fluent API code to create a "one `Book` has many `Reviews`" relationship in the Book App.

In this case, the `Review` entity class doesn't have a navigational link back to the `Book`, so the `WithOne` method has no parameter.

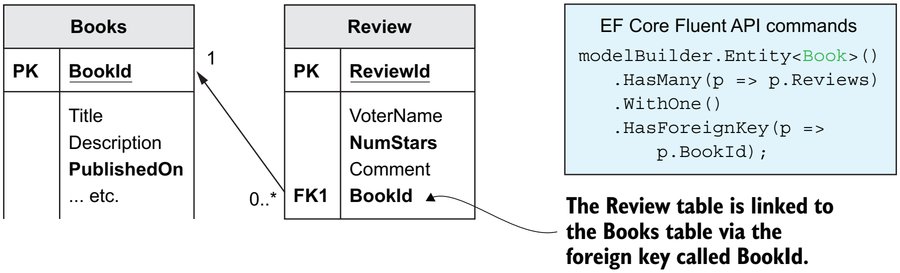


Figure 8.8 A one-to-many relationship, in which the foreign key must be in the dependent entity—in this case, the `Review` entity class. You can see in the Fluent API on the right that the `Book` has a collection navigational property, `Reviews`, linked to the `Review` entity classes, but `Review` doesn't have a navigational property back to `Book`.

> **NOTE** Listing 3.16 shows how to add a `Review` to the `Book`'s one-to-many collection navigational property, `Reviews`.

Collections have a couple of features that are worth knowing about. First, you can use any generic type for a collection that implements the `IEnumerable<T>` interface, such as `IList<T>`, `Collection<T>`, `HashSet<T>`, `List<T>`, and so on. `IEnumerable<T>` on its own is a special case, as you can't add to that collection.

For performance reasons, you should use `HashSet<T>` for navigational collections, because it improves certain parts of EF Core's query and update processes. (See chapter 14 for more on this topic.) But `HashSet` doesn't guarantee the order of entries, which could cause problems if you add sorting to your `Includes` (see section 2.4.1, listing 2.5). That's why I recommend in part 1 and 2 using `ICollection<T>` if you might sort your `Include` methods, as `ICollection` preserves the order in which entries are added. But in part 3, which is about performance, you don't use sort in `Includes` so that you can use `HashSet<T>` for better performance.

Second, although you typically define a collection navigational property with a getter and a setter (such as `public ICollection<Review> Reviews { get; set; }`), doing so isn't necessary. You can provide a getter only if you initialize the backing field with an empty collection. The following is also valid:

```
public ICollection<Review> Reviews { get; } = new List<Review>();
```

Although initializing the collection might make things easier in this case, I don't recommend initializing a navigational collection property. I have given my reasons for not initializing collection navigational properties in section 6.1.6.

8.6.3 Creating a many-to-many relationship

Many-to-many relationships are described in chapters 2 and 3; in this section, you learn how to configure them. In those chapters, you learned about the two types of many-to-many relationships:

- *Your linking table contains information that you want to access when reading in the data on the other side of the many-to-many relationship.* An example is the `Book` to `Author` many-to-many relationship, in which the linking table contains the order in which the `Author Names` should be shown.
- *You directly access the other side of the many-to-many relationship.* An example is the `Book` to `Tag` many-to-many relationship, in which you can directly access the `Tags` collection in the `Book` entity class without ever needing to access the linking table.

CONFIGURING A MANY-TO-MANY RELATIONSHIP USING A LINKING ENTITY CLASS

You start with the many-to-many relationship in which you access the other end of the relationship via the linking table. This relationship takes more work but allows you to add extra data to the linking table, which you can sort/filter on. You saw how to do this in section 3.4.4. Figure 8.9 looks at the configuration parts of this many-to-many relationship.

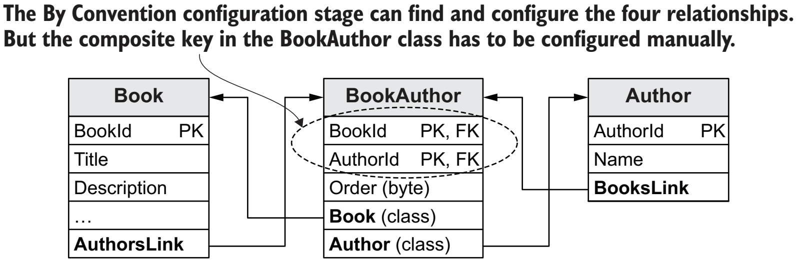


Figure 8.9 The three entity classes involved in a many-to-many relationship, using a linking table. This type of many-to-many relationship is used only if you have extra data in the linking table entity class. In this case, the `BookAuthor` class contains an `Order` property that defines the order in which the `Author Names` should be displayed alongside a `Book`.

In the `Book/Author` example, the By Convention configuration can find and link all the scalar and navigational properties so that the only configuration required is setting up the primary key. The following code snippet uses Fluent API in the application's DbContext's `OnModelCreating` method:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
     modelBuilder.Entity<BookAuthor>()
           .HasKey(x => new {x.BookId, x.AuthorId});
}
```

You can configure the four relationships in the many-to-many relationship by using the Fluent API with the code in the following listing. Note that the `HasOne/WithMany` Fluent API commands in the listing aren't needed because the `BookAuthor` entity class follows the By Convention naming and typing rules.

Listing 8.6 Configuring a many-to-many relationship via two one-to-many relationships

```
public static void Configure
    (this EntityTypeBuilder<BookAuthor> entity)
{
    entity.HasKey(p =>
         new { p.BookId, p.AuthorId });

    //-----------------------------
    //Relationships

    entity.HasOne(p => p.Book)
        .WithMany(p => p.AuthorsLink)
        .HasForeignKey(p => p.BookId);

    entity.HasOne(p => p.Author)
        .WithMany(p => p.BooksLink)
        .HasForeignKey(p => p.AuthorId);
}
```

Uses the names of the Book and Author primary keys to form its own composite key

Configures the one-to-many relationship from the Book to BookAuthor entity class

Configures the one-to-many relationship from the Author to the BookAuthor entity class

CONFIGURING A MANY-TO-MANY RELATIONSHIP WITH DIRECT ACCESS TO THE OTHER ENTITY

With the release of EF Core 5, you can reference the other end of a many-to-many relationship directly. The example shown in chapter 2 and 3 was the `Book` entity class, which has an `ICollection<Tag> Tags` navigation property that holds a series of `Tag` entity classes. The `Tag` entity class contains a category (`Microsoft .NET`, `Web`, and so on), which helps the customer find the book they are looking for.

The By Convention configuration works well for a direct many-to-many relationship. If the entity classes at the two ends are valid, the By Convention configuration will set up the relationships and keys for you, as shown in figure 8.10. By Convention will also create the linking entity for you by using a property bag (see section 8.9.5).

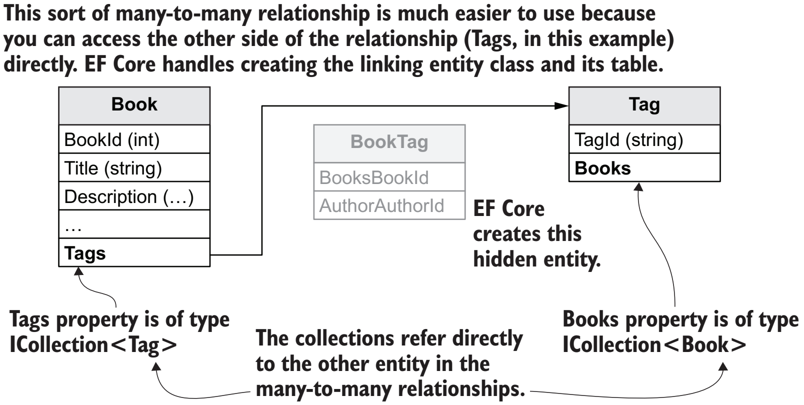


Figure 8.10 EF Core 5's direct many-to-many relationship works because (a) EF Core creates the linking entity class for you and (b) when it sees a query containing a direct many-to-many relationship, it adds the SQL commands to use the hidden linking entity class. Not having to create the linking entity class or perform configuration makes these sorts of many-to-many relationships much easier to set up.

But if you want to add your own linking table and configuration, you can do that via Fluent API configuration. The entity class for the linking table is similar to the `BookAuthor` linked entity class shown in figure 8.9. The difference is that the `Author` key/relationship is replaced by the `Tag` key/relationship. The following listing shows the `Book` configuration class setting up the `BookTag` entity class to link the two parts.

Listing 8.7 Configuring direct many-to-many relationships using Fluent API

```
public void Configure
    (EntityTypeBuilder<Book> entity)
{
    //… other configrations left out for clarity

    entity.HasMany(x => x.Tags)
        .WithMany(x => x.Books)
        .UsingEntity<BookTag>(
            bookTag => bookTag.HasOne(x => x.Tag)
                .WithMany().HasForeignKey(x => x.TagId),
            bookTag => bookTag.HasOne(x => x.Book)
                .WithMany().HasForeignKey(x => x.BookId));
}
```

The HasMany/WithMany sets up a direct many-to-many relationship.

The UsingEntity<T> method allows you to define an entity class for the linking table.

Defined Tag side of the many-to-many relationship

Defined Book side of the many-to-many relationship

The code shown in listing 8.7 does nothing but replace the linking entity that EF Core would have added, so it isn't worth doing. But it would be useful if you wanted to add extra properties to the `BookTag` entity class, such as a `SoftDeleted` property that uses a Query Filter to soft-delete a link.

> **MORE INFO** When EF Core 5 was released, a useful video offered good coverage of direct-access many-to-many relationships, including adding your own linking table (and TPH and TPT). See http://mng.bz/opzM.

8.7 Controlling updates to collection navigational properties

Sometimes, you need to control access to collection navigational properties. Although you can control access to a one-to-one navigational by making the setter private, that approach doesn't work for a collection, as most collection types allow you to add or remove entries. To control collection navigational properties fully, you need to use EF Core backing fields, described in section 7.14.

> **EF6** EF6.x didn't have a way to control access to collection navigational properties, which meant that some patterns, such as DDD, were hard to implement successfully. EF Core's backing fields allow you to build entity classes that follow DDD principles.

Storing the collection of linked entities classes in a field allows you to intercept any attempt to update a collection. Here are some business/software design reasons why this feature is useful:

- Triggering some business logic on a change, such as calling a method if a collection contains more than ten entries.
- Building a local cached value for performance reasons, such as holding a cached `ReviewsAverageVotes` property whenever a `Review` is added to or removed from your `Book` entity class.
- Applying a DDD to your entity classes. Any change to data should be done via a method (see chapter 13).

For the example of controlling collection navigational properties, you are going to add a cached `ReviewsAverageVotes` property to the `Book` class. This property will hold the average of the votes in all the `Reviews` linked to this `Book`. To do so, you need to

- Add a backing field called `_reviews` to hold the `Reviews` collection and change the property to return a read-only copy of the collection held in the `_reviews` backing field.
- Add a read-only property called `ReviewsAverageVotes` to hold the cached average votes from the `Reviews` linked to this `Book`.
- Add methods to `Add Reviews` to and `Remove Reviews` from the `_reviews` backing field. Each method recalculates the average votes, using the current list of `Reviews`.

The following listing shows the updated `Book` class showing the code related to the `Reviews` and the cached `ReviewsAverageVotes` property.

Listing 8.8 `Book` class with a read-only `Reviews` collection navigational property

```
public class Book
{
    private readonly ICollection<Review> _reviews
        = new List<Review>();

    public int BookId { get; set; }
    public string Title { get; set; }
    //… other properties/relationships left out

    public double? ReviewsAverageVotes { get; private set; }

    public IReadOnlyCollection<Review> Reviews =>
        _reviews.ToList();

    public void AddReview(Review review)
    {
        _reviews.Add(review);
        ReviewsAverageVotes =
            _reviews.Average(x => x.NumStars);
    }
```

You add a backing field, which is a list. By default, EF Core will read and write to this field.

A read-only collection so that no one can change the collection

Holds a precalculated average of the reviews and is read-only

Returns a copy of the reviews in the _reviews backing field

Adds a method to allow a new Review to be added to the _reviews collection

Adds the new review to the backing field _reviews and updates the database on the call to SaveChanges

Recalculates the average votes for the book

```
    public void RemoveReview(Review review)
    {
        _reviews.Remove(review);
        ReviewsAverageVotes = _reviews.Any()
            ? _reviews.Average(x => x.NumStars)
            : (double?)null;
    }
}
```

Adds a method to remove a review from the _reviews collection

Removes the review from the list and updates the database on the call to SaveChanges

If there are any reviews, recalculates the average votes for the book

If there are no reviews, sets the value to null

You didn't have to configure the backing field because you were using By Convention naming, and by default, EF Core reads and writes data to the `_reviews` field.

This example shows how to make your collection navigational properties read-only, but it's not perfect because concurrent updates could make the `ReviewsAverageVotes` cache property out of date. In part 3, you will build an application using DDD throughout and implement a robust caching approach that handles concurrency issues.

8.8 Additional methods available in Fluent API relationships

We have covered all the ways to configure standard relationships, but some of the most detailed parts of a relationship require adding extra commands to your Fluent API configuration of a relationship. In this section, we'll go through four methods that define some of the deeper parts of a relationship:

- `OnDelete`—Changes the delete action of a dependent entity
- `IsRequired`—Defines the nullability of the foreign key
- `HasPrincipalKey`—Uses an alternate unique key
- `HasConstraintName`—Sets the foreign-key constraint name and `MetaData` access to the relationship data

8.8.1 OnDelete: Changing the delete action of a dependent entity

Section 8.4.4 described the default action on the deletion of a principal entity, which is based on the nullability of the dependent's foreign key(s). The `OnDelete` Fluent API method allows you to alter what EF Core does when a deletion that affects a dependent entity occurs.

You can add the `OnDelete` method to the end of a Fluent API relationship configuration. This listing shows the code added in chapter 4 to stop a `Book` entity from being deleted if it was referred to in a customer order, via the `LineItem` entity class.

Listing 8.9 Changing the default `OnDelete` action on a dependent entity

```
public static void Configure
    (this EntityTypeBuilder<LineItem> entity)
{
    entity.HasOne(p => p.ChosenBook)
```

```
            .WithMany()
            .OnDelete(DeleteBehavior.Restrict);
}
```

Adds the OnDelete method to the end of defining a relationship

This code causes an exception to be thrown if someone tries to delete a `Book` entity that a `LineItem`'s foreign key links to that `Book`. You do this because you want a customer's order to not be changed. Table 8.1 explains the possible `DeleteBehavior` settings.

Table 8.1 Delete behaviors available in EF Core. The middle column highlights the delete behavior that will be used if you don't apply the `OnDelete` option.

Name	Effect of the delete behavior on the dependent entity	Default for
`Restrict`	The delete operation isn't applied to dependent entities. The dependent entities remain unchanged, which may cause the delete to fail, either in EF Core or in the relational database.	
`SetNull`	The dependent entity isn't deleted, but its foreign-key property is set to `null`. If any of the dependent entity foreign-key properties isn't nullable, an exception is thrown when `SaveChanges` is called.	
`ClientSetNull`	If EF Core is tracking the dependent entity, its foreign key is set to `null`, and the dependent entity isn't deleted. But if EF Core isn't tracking the dependent entity, the database rules apply. In a database created by EF Core, this `DeleteBehavior` will set the SQL DELETE constraint to NO ACTION, which causes the delete to fail with an exception.	Optional relationships
`Cascade`	The dependent entity is deleted.	Required relationships
`ClientCascade`	For entities being tracked by the DbContext, dependent entities will be deleted when the related principal is deleted. But if EF Core isn't tracking the dependent entity, the database rules apply. In a database created by EF Core, this will be set to `Restrict`, which causes the delete to fail with an exception.	

Two delete behaviors whose names start with `Client` are `ClientSetNull` (added in EF Core 2.0) and `ClientCascade` (added in EF Core 3.0). These two delete behaviors move some of the handling of deletion actions from the database to the client—that is, the EF Core code. I believe that these two settings have been added to prevent the problems you can get in some databases, such as SQL Server, when your entities have navigational links that loop back to themselves. In these cases, you would get an error from the database server when you try to create your database, which can be hard to diagnose and fix.

In both cases, these commands execute code inside EF Core that does the same job that the database would do with the `SetNull` and `Cascade` delete behaviors, respectively. But—and it's a big *but*—EF Core can apply these changes only if you have loaded all the relevant dependent entities linked to the principal entity that you are going to delete. If you don't, the database applies its delete rules, which normally will throw an exception.

The `ClientSetNull` delete setting is the default for optional relationships, and EF Core will set the foreign key of the loaded dependent entity class to `null`. If you use EF Core to create/migrate the database, EF Core sets the database delete rules to `ON DELETE NO ACTION` (SQL Server). The database server won't throw an exception if your entities have a circular loop (referred to as possible cyclic delete paths by SQL Server). The `SetNull` delete setting would set the database delete rules to `ON DELETE SET NULL` (SQL Server), which would cause the database server to throw a `possible cyclic delete paths` exception.

The `ClientCascade` delete setting does the same thing for the database's cascade-delete feature, in that it will delete any loaded dependent entity class(es). Again, if you use EF Core to create/migrate the database, EF Core sets the database delete rules to `ON DELETE NO ACTION` (SQL Server). The `Cascade` delete setting would set the database delete rules to `ON DELETE CASCADE` (SQL Server), which would cause the database server to throw a `possible cyclic delete paths` exception.

> **NOTE** The EF Core documentation has a page on cascade delete with some worked examples; see http://mng.bz/nMGK. Also, the Part2 branch of the associated GitHub repo has a unit test called `Ch08_DeleteBehaviour`, with tests of each of the settings.

Listing 8.10 shows the correct way to use the `ClientSetNull` and `ClientCascade` when you delete a principal entity. The entity in this listing is loaded with an optional dependent entity, which (by default) has the default delete behavior of `ClientSetNull`. But the same code would work for the `ClientCascade` as long as you load the correct dependent entity or entities.

Listing 8.10 Deleting a principal entity with an optional dependent entity

```
var entity = context.DeletePrincipals
    .Include(p => p.DependentDefault)
    .Single(p => p.DeletePrincipalId == 1);

context.Remove(entity);
context.SaveChanges();
```

Reads in the principal entity

Includes the dependent entity that has the default delete behavior of ClientSetNull

Sets the principal entity for deletion

Calls SaveChanges, which sets its foreign key to null

Note that if you don't include the `Include` method or another way of loading the optional dependent entity, `SaveChanges` will throw a `DbUpdateException` because the database server will have reported a foreign-key constraint violation. One way to align EF Core's approach to an optional relationship with the database server's approach is to set the delete behavior to `SetNull` instead of the default `ClientSetNull`, making the foreign-key constraint in the database `ON DELETE SET NULL` (SQL Server) and putting the database in charge of setting the foreign key to `null`. Whether or not you load the optional dependent entity, the outcome of the called `SaveChanges` will be the same: the foreign key on the optional dependent entity will be set to `null`.

But be aware that some database servers may return an error on database creation if you have a delete-behavior setting of `SetNull` or `Cascade` and the servers detect a possible circular relationship, such as hierarchical data. That's why EF Core has the `ClientSetNull` and `ClientCascade` delete behaviors.

NOTE If you're managing the database creation/migration outside EF Core, it's important to ensure that the relational database foreign-key constraint is in line with EF Core's `OnDelete` setting. Otherwise, you'll get inconsistent behavior, depending on whether the dependent entity is being tracked.

8.8.2 IsRequired: Defining the nullability of the foreign key

Chapter 6 describes how the Fluent API method `IsRequired` allows you to set the nullability of a scalar property, such as a string. In a relationship, the same command sets the nullability of the foreign key—which, as I've already said, defines whether the relationship is required or optional.

The `IsRequired` method is most useful in shadow properties because EF Core makes shadow properties nullable by default, and the `IsRequired` method can change them to non-nullable. The next listing depicts the `Attendee` entity class, used previously to show a one-to-one relationship, but showing two other one-to-one relationships that use shadow properties for their foreign keys.

Listing 8.11 The `Attendee` entity class showing all its relationships

```
public class Attendee
{
    public int AttendeeId { get; set; }
    public string Name { get; set; }

    public int TicketId { get; set; }
    public Ticket Ticket { get; set; }

    public MyOptionalTrack Optional { get; set; }
    public MyRequiredTrack Required { get; set; }
}
```

Foreign key for the one-to-one relationship, Ticket

One-to-one navigational property that accesses the Ticket entity

One-to-one navigational property using a shadow property for the foreign key. By default, the foreign key is nullable, so the relationship is optional.

One-to-one navigational property using a shadow property for the foreign key. You use Fluent API commands to say that the foreign key isn't nullable, so the relationship is required.

The `Optional` navigational property, which uses a shadow property for its foreign key, is configured by convention, which means that the shadow property is left as a nullable value. Therefore, it's optional, and if the `Attendee` entity is deleted, the `MyOptionalTrack` entity isn't deleted.

For the `Required` navigational property, the following listing presents the Fluent API configuration. Here, you use the `IsRequired` method to make the `Required` one-to-one navigational property as required. Each `Attendee` entity must have a `MyRequiredTrack` entity assigned to the `Required` property.

Listing 8.12 The Fluent API configuration of the `Attendee` entity class

```
public void Configure
    (EntityTypeBuilder<Attendee> entity)
{
    entity.HasOne(attendee => attendee.Ticket)
        .WithOne(attendee => attendee.Attendee)
        .HasForeignKey<Attendee>
            (attendee => attendee.TicketId)
        .IsRequired();

    entity.HasOne(attendee => attendee.Required)
        .WithOne(attendee => attend.Attend)
        .HasForeignKey<Attendee>(
            "MyShadowFk")
        .IsRequired();
}
```

Sets up the one-to-one navigational relationship, Ticket, which has a foreign key defined in the Attendee class

Specifies the property that's the foreign key. You need to provide the class type, as the foreign key could be in the principal or dependent entity class.

Sets up the one-to-one navigational relationship, Required, which doesn't have a foreign key defined

Uses the HasForeignKey<T> method, which takes a string because it's a shadow property and can be referred to only via a name. Note that you use your own name.

Uses IsRequired to say the foreign key should not be nullable

You could've left out the configuration of the `Ticket` navigational property, as it would be configured correctly under the By Convention rules, but you leave it in so that you can compare it with the configuration of the `Required` navigational property, which uses a shadow property for its foreign key. The configuration of the `Required` navigational property is necessary because the `IsRequired` method changes the shadow foreign-key property from nullable to non-nullable, which in turn makes the relationship required.

TYPE AND NAMING CONVENTIONS FOR SHADOW PROPERTY FOREIGN KEYS

Notice how listing 8.12 refers to the shadow foreign-key property: you need to use the `HasForeignKey<T>(string)` method. The `<T>` class tells EF Core where to place the shadow foreign-key property, which can be either end of the relationship for one-to-one relationships or the many entity class of a one-to-many relationship.

The string parameter of the `HasForeignKey<T>(string)` method allows you to define the shadow foreign-key property name. You can use any name; you don't need to stick with the By Convention name listed in figure 8.3. But you need to be careful not to use a name of any existing property in the entity class you're targeting, because that approach can lead to strange behaviors. (There's no warning if you do select an existing property, as you might be trying to define a nonshadow foreign key.)

8.8.3 HasPrincipalKey: Using an alternate unique key

I mentioned the term *alternate key* at the beginning of this chapter, saying that it is a unique value but not the primary key. I gave an example of an alternate key called `UniqueISBN`, which represents a unique key that isn't the primary key. (Remember that *ISBN* stands for *International Standard Book Number*, which is a unique number for every book.)

Now let's look at a different example. The following listing creates a `Person` entity class, which uses a normal `int` primary key, but you'll use the `UserId` as an alternate key when linking to the person's contact information, shown in listing 8.14.

Listing 8.13 `Person` class, with `Name` taken from ASP.NET authorization

```
public class Person
{
    public int PersonId { get; set; }

    public string Name { get; set; }

    public Guid UserId { get; set; }

    public ContactInfo ContactInfo { get; set; }
}
```

Holds the person's unique Id

Navigational property linking to the ContactInfo

Listing 8.14 `ContactInfo` class with `EmailAddress` as a foreign key

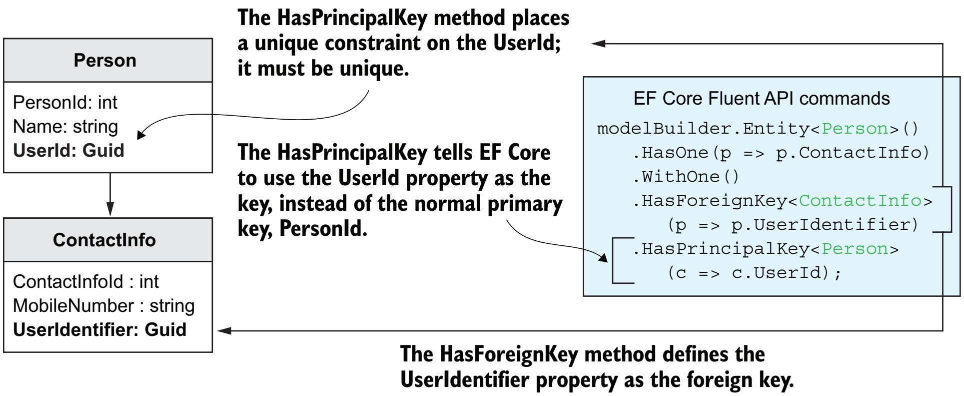
```
public class ContactInfo
{
    public int ContactInfoId { get; set; }

    public string MobileNumber { get; set; }
    public string LandlineNumber { get; set; }

    public Guid UserIdentifier { get; set; }
}
```

The UserIdentifier is used as a foreign key for the Person entity to link to this contact info.

Figure 8.11 shows the Fluent API configuration commands, which use the alternate key in the `Person` entity class as a foreign key in the `ContactInfo` entity class.

Here are a few notes on alternate keys:

- You can have composite alternate keys, which are made up of two or more properties. You handle them in the same way that you do composite keys: by using an anonymous Type, such as `HasPrincipalKey<MyClass>(c => new {c.Part1, c.Part2})`.
- Unique keys (see section 7.10) and alternate keys are different, and you should choose the correct one for your business case. Here are some of the differences:
  - Unique keys ensure that each entry is unique; they can't be used in a foreign key.
  - Unique keys can be `null`, but alternate keys can't.
  - Unique key values can be updated, but alternate keys can't. (See EF Core issue #4073 at http://mng.bz/vzEM).
- You can define a property as a standalone alternate key by using the Fluent API command `modelBuilder.Entity<Car>().HasAlternateKey(c => c.LicensePlate)`, but you don't need to do that, because using the `HasPrincipalKey`



```
modelBuilder.Entity<Person>()
    .HasOne(p => p.ContactInfo)
    .WithOne()
    .HasForeignKey<ContactInfo>
        (p => p.UserIdentifier)
    .HasPrincipalKey<Person>
        (c => c.UserId);
```

The HasPrincipalKey method places a unique constraint on the UserId; it must be unique.

The HasPrincipalKey tells EF Core to use the UserId property as the key, instead of the normal primary key, PersonId.

The HasForeignKey method defines the UserIdentifier property as the foreign key.

Figure 8.11 The Fluent API sets up a one-to-one relationship by using the `UserId` property, which contains the person's unique `Id`, as the foreign key to link to the `ContactInfo`. The command `HasPrincipalKey` both defines the `UserId` property as an alternate key and creates the foreign-key constraint link between the `UserIdentifier` property in the `ContactInfo` entity and the `UserId` in the `Person` entity.

method to set up a relationship automatically registers the property as an alternate key.

8.8.4 Less-used options in Fluent API relationships

This section briefly mentions—but doesn't cover in detail—two Fluent API commands that can be used for setting up relationships.

HASCONSTRAINTNAME: SETTING THE FOREIGN-KEY CONSTRAINT NAME

The method `HasConstraintName` allows you to set the name of the foreign-key constraint, which can be useful if you want to catch the exception on foreign-key errors and use the constraint name to form a more user-friendly error message. This article shows how: http://mng.bz/4ZwV.

METADATA: ACCESS TO THE RELATIONSHIP INFORMATION

The `MetaData` property provides access to the relationship data, some of which is read/write. Much of what the `MetaData` property exposes can be accessed via specific commands, such as `IsRequired`, but if you need something out of the ordinary, look through the various methods/properties supported by the `MetaData` property.

8.9 Alternative ways of mapping entities to database tables

Sometimes, it's useful to not have a one-to-one mapping from an entity class to a database table. Instead of having a relationship between two classes, you might want to combine both classes into one table. This approach allows you to load only part of the table when you use one of the entities, which will improve the query's performance.

This section describes five alternative ways to map classes to the database, each with advantages in certain situations:

- *Owned types*—Allows a class to be merged into the entity class's table and is useful for using normal classes to group data.
- *Table per hierarchy (TPH)*—Allows a set of inherited classes to be saved in one table, such as classes called `Dog`, `Cat`, and `Rabbit` that inherit from the `Animal` class.
- *Table per type (TPT)*—Maps each class to a different table. This approach works like TPH except that each class is mapped to a separate table.
- *Table splitting*—Allows multiple entity classes to be mapped to the same table and is useful when some columns in a table are read more often than all the table columns.
- *Property bags*—Allows you to create an entity class via a `Dictionary`, which gives you the option to create the mapping on startup. Property bags also use two other features: mapping the same type to multiple tables and using an indexer in your entity classes.

8.9.1 Owned types: Adding a normal class into an entity class

EF Core has *owned types*, which allow you to define a class that holds a common grouping of data, such as an address or audit data, that you want to use in multiple places in your database. The owned type class doesn't have its own primary key, so it doesn't have an identity of its own; it relies on the entity class that "owns" it for its identity. In DDD terms, owned types are known as *value objects*.

> **EF6** EF Core's owned types are similar to EF6.x's complex types. The biggest change is that you must specifically configure an owned type, whereas EF6.x considers any class without a primary key to be a complex type (which could cause bugs). EF Core's owned types have an extra feature over EF6.x's implementation: the data in an owned type can be configured to be saved in a separate, hidden table.

Here are two ways of using owned types:

- The owned type data is held in the same table that the entity class is mapped to.
- The owned type data is held in a separate table from the entity class.

OWNED TYPE DATA IS HELD IN THE SAME TABLE AS THE ENTITY CLASS

As an example of an owned type, you'll create an entity class called `OrderInfo` that needs two addresses: `BillingAddress` and `DeliveryAddress`. These addresses are provided by the `Address` class, shown in the following listing. You can mark an `Address` class as an owned type by adding the attribute `[Owned]` to the class. An owned type has no primary key, as shown at the bottom of the listing.

Listing 8.15 The `Address` owned type, followed by the `OrderInfo` entity class

```
public class OrderInfo
{
    public int OrderInfoId { get; set; }
    public string OrderNumber { get; set; }

    public Address BillingAddress { get; set; }
    public Address DeliveryAddress { get; set; }
}

[Owned]
public class Address
{
    public string NumberAndStreet { get; set; }
    public string City { get; set; }
    public string ZipPostCode { get; set; }
    [Required]
    [MaxLength(2)]
    public string CountryCodeIso2 { get; set; }
}
```

The entity class OrderInfo, with a primary key and two addresses

Two distinct Address classes. The data for each Address class will be included in the table that the OrderInfo is mapped to.

The attribute [Owned] tells EF Core that it is an owned type.

An owned type has no primary key.

Because you added the attribute `[Owned]` to the `Address` class, and because you are using the owned type within the same table, you don't need use the Fluent API to configure the owned type. This approach saves you time, especially if your owned type is used in many places, because you don't have to write the Fluent API configuration. But if you don't want to use the `[Owned]` attribute, the next listing shows you the Fluent API to tell EF Core that the `BillingAddress` and the `DeliveryAddress` properties in the `OrderInfo` entity class are owned types, not relationships.

Listing 8.16 The Fluent API to configure the owned types within `OrderInfo`

```
public class SplitOwnDbContext: DbContext
{
    public DbSet<OrderInfo> Orders { get; set; }
    //… other code removed for clarity

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
       modelBuilder.Entity<OrderInfo>()
          .OwnsOne(p => p.BillingAddress);
       modelBuilder.Entity<OrderInfo>()
          .OwnsOne(p => p.DeliveryAddress);
       }
}
```

Selects the owner of the owned type

Uses the OwnsOne method to tell EF Core that property BillingAddress is an owned type and that the data should be added to the columns in the table that the OrderInfo maps to

Repeats the process for the second property, DeliveryAddress

The result is a table containing the two scalar properties in the `OrderInfo` entity class, followed by two sets of `Address` class properties, one prefixed by `BillingAddress_` and another prefixed by `DeliveryAddress_`. Because an owned type property can be `null`, all the properties are held in the database as nullable columns. The `CountryCodeIso2` property in listing 8.15, for example, is marked as `[Required]`, so it should be non-nullable, but to allow for a `null` property value for the `BillingAddress` or `DeliveryAddress`, it is stored in a nullable column. EF Core does this to tell whether an instance of the owned type should be created when the entity containing an owned type is read in.

The fact that the owned type property can be `null` means that owned types within an entity class are a good fit for what DDD calls a *value object*. A value object has no key, and two value objects with the same properties are considered to be equal. The fact that they can be `null` allows for an "empty" value object.

> **NOTE** Nullable owned types were introduced in EF Core 3.0 but had some performance issues. (The SQL uses a `LEFT JOIN`.) EF Core 5 has fixed those performance issues.

The following listing shows part of the SQL Server `CREATE TABLE` command that EF Core produces for the `OrderInfo` entity class with the naming convention.

Listing 8.17 The `SQL CREATE TABLE` command showing the column names

```
CREATE TABLE [Orders] (
    [OrderInfoId] int NOT NULL IDENTITY,
    [OrderNumber] nvarchar(max) NULL,
    [BillingAddress_City] nvarchar(max) NULL,
    [BillingAddress_NumberAndStreet] nvarchar(max) NULL,
    [BillingAddress_ZipPostCode] nvarchar(max) NULL,
    [BillingAddress_CountryCodeIso2] [nvarchar](2) NULL
    [DeliveryAddress_City] nvarchar(max) NULL,
    [DeliveryAddress_CountryCodeIso2] nvarchar(max) NULL,
    [DeliveryAddress_NumberAndStreet] nvarchar(max) NULL,
    [DeliveryAddress_CountryCodeIso2] [nvarchar](2) NULL,
    CONSTRAINT [PK_Orders] PRIMARY KEY ([OrderInfoId])
);
```

Property has a [Required] attribute but is stored as a nullable value to handle the billing/delivery address being null.

By default, every property or field in an owned type is stored in a nullable column, even if they are non-nullable. EF Core does this to allow you to not assign an instance to an owned type, at which point all the columns that the owned type uses are set to NULL. And if an entity with an owned type is read in, and all the columns for an owned type are NULL, the owned type property is set to `null`.

But EF Core 5 added a feature to allow you to say that an owned type is required—that is, must always be present. To do so, you add the Fluent API `IsRequired` method to the `OrderInfo`'s `DeliveryAddress` navigational property mapped to the owned type (see the next listing). In addition, this feature allows the individual nullability of columns to follow normal rules. The DeliveryAddress_CountryCodeIso2 column shown in listing 8.17, for example, is now NOT NULL.

Listing 8.18 The Fluent API to configure the owned types within `OrderInfo`

```
protected override void OnModelCreating
    (ModelBuilder modelBuilder)
{
   modelBulder.Entity<OrderInfo>()
       .OwnsOne(p => p.BillingAddress);
   modelBulder.Entity<OrderInfo>()
       .OwnsOne(p => p.DeliveryAddress);

   modelBulder.Entity<OrderInfo>()
  .Navigation(p => p.DeliveryAddress)
        .IsRequired();
}
```

Selects the DeliveryAddress navigational property

Applying the IsRequired method means that the DeliveryAddress must not be null.

Using owned types can help you organize your database by turning common groups of data into owned types, making it easier to handle common data groups, such as `Address` and so on, in your code. Here are some final points on owned types held in an entity class:

- The owned type navigation properties, such as `BillingAddress`, are automatically created and filled with data when you read the entity. There's no need for an `Include` method or any other form of relationship loading.
- Julie Lerman (@julielerman on Twitter) pointed out that owned types can replace one-to-zero-or-one relationships, especially if an owned type is small. Owned types have better performance and are automatically loaded, which means that they would be better implementations of the zero-or-one `PriceOffer` used in the Book App.
- Owned types can be nested. You could create a `CustomerContact` owned type, which in turn contains an `Address` owned type, for example. If you used the `CustomerContact` owned type in another entity class—let's call it `SuperOrder`—all the `CustomerContact` properties and the `Address` properties would be added to the `SuperOrder`'s table.

OWNED TYPE DATA IS HELD IN A SEPARATE TABLE FROM THE ENTITY CLASS

The other way that EF Core can save the data inside an owned type is in a separate table rather than the entity class. In this example, you'll create a `User` entity class that has a property called `HomeAddress` of type `Address`. In this case, you add a `ToTable` method after the `OwnsOne` method in your configuration code.

Listing 8.19 Configuring the owned table data to be stored in a separate table

```
public class SplitOwnDbContext: DbContext
{
    public DbSet<OrderInfo> Orders { get; set; }
    //… other code removed for clarity

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
```

```
    {
       modelBulder.Entity<User>()
          .OwnsOne(p => p.HomeAddress);
          .ToTable("Addresses");
    }
}
```

Adding ToTable to OwnsOne tells EF Core to store the owned type, Address, in a separate table, with a primary key equal to the primary key of the User entity that was saved to the database.

EF Core sets up a one-to-one relationship in which the primary key is also the foreign key (see section 8.6.1, option 3), and the `OnDelete` state is set to `Cascade` so that the owned type entry of the primary entity, `User`, is deleted. Therefore, the database has two tables: Users and Addresses.

Listing 8.20 The Users and Addresses tables in the database

```
CREATE TABLE [Users] (
    [UserId] int NOT NULL IDENTITY,
    [Name] nvarchar(max) NULL,
    CONSTRAINT [PK_Orders] PRIMARY KEY ([UserId])
);
CREATE TABLE [Addresses] (
    [UserId] int NOT NULL IDENTITY,
    [City] nvarchar(max) NULL,
    [CountryCodeIso2] nvarchar(2) NOT NULL,
    [NumberAndStreet] nvarchar(max) NULL,
    [ZipPostCode] nvarchar(max) NULL,
    CONSTRAINT [PK_Orders] PRIMARY KEY ([UserId]),
    CONSTRAINT "FK_Addresses_Users_UserId" FOREIGN KEY ("UserId")
        REFERENCES "Users" ("UserId") ON DELETE CASCADE
);
```

Notice that non-nullable properties, or nullable properties with the Required setting, are now stored in non-nullable columns.

This use of owned types differs from the first use, in which the data is stored in the entity class table, because you can save a `User` entity instance without an address. But the same rules apply on querying: the `HomeAddress` property will be read in on a query of the `User` entity without the need for an `Include` method.

The Addresses table used to hold the `HomeAddress` data is hidden; you can't access it via EF Core. This situation could be a good thing or a bad thing, depending on your business needs. But if you want to access the `Address` part, you can implement the same feature by using two entity classes with a one-to-many relationship between them.

8.9.2 *Table per hierarchy (TPH): Placing inherited classes into one table*

Table per hierarchy (TPH) stores all the classes that inherit from one another in a single database table. If you want to save a payment in a shop, for example, that payment could be cash (`PaymentCash`) or credit card (`PaymentCard`). Each option contains the amount (say, $10), but the credit card option has extra information, such as an online-transaction receipt. In this case, TPH uses a single table to store all the versions of the inherited classes and return the correct entity type, `PaymentCash` or `PaymentCard`, depending on what was saved.

> **TIP** I have used TPH classes in a couple of projects for my clients, and I find TPH to be a good solution for storing sets of data that are similar when some sets need extra properties. Suppose that you had a lot of product types with common `Name`, `Price`, `ProductCode`, `Weight`, and other properties, but the Sealant products needs `MinTemp` and `MaxTemp` properties, which TPH could implement by using one table rather than lots of tables.

TPH can be configured By Convention, which will combine all the versions of the inherited classes into one table. This approach has the benefit of keeping common data in one table, but accessing that data is a little cumbersome because each inherited type has its own `DbSet<T>` property. But when you add the Fluent API, all the inherited classes can be accessed via one `DbSet<T>` property, which in our example makes the `PaymentCash` / `PaymentCard` example much more useful.

The first example uses multiple `DbSet<T>`s, one for each class, and is configured By Convention. The second example uses one `DbSet<T>` mapped to the base class, which I find to be the more useful version, and shows the TPH Fluent API commands.

CONFIGURING TPH BY CONVENTION

To apply the By Convention approach to the `PaymentCash/PaymentCard` example, you create a class called `PaymentCash` and then a class called `PaymentCard` that inherits from `PaymentCash`, as shown in the following listing. As you can see, `PaymentCard` inherits from `PaymentCash` and adds an extra `ReceiptCode` property.

Listing 8.21 The two classes: `PaymentCash` and `PaymentCard`

```
public class PaymentCash
{
    [Key]
    public int PaymentId { get; set; }
    public decimal Amount { get; set; }
}

//PaymentCredit - inherits from PaymentCash
public class PaymentCard : PaymentCash
{
    public string ReceiptCode { get; set; }
}
```

Listing 8.22, which uses the By Convention approach, shows your application's DbContext with two `DbSet<T>` properties, one for each of the two classes. Because you include both classes, and `PaymentCard` inherits from `PaymentCash`, EF Core will store both classes in one table.

Listing 8.22 The updated application's DbContext with the two `DbSet<T>` properties

```
public class Chapter08DbContext : DbContext
{
    //… other DbSet<T> properties removed
```

```
    //Table-per-hierarchy
    public DbSet<PaymentCash> CashPayments { get; set; }
    public DbSet<PaymentCard> CreditPayments { get; set; }

    public Chapter08DbContext(
        DbContextOptions<Chapter08DbContext> options)
        : base(options)
    { }

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
        //no configuration needed for PaymentCash or PaymentCard
    }
}
```

Finally, this listing shows the code that EF Core produces to create the table that will store both the `PaymentCash` and `PaymentCard` entity classes.

Listing 8.23 The SQL produced by EF Core to build the CashPayment table

```
CREATE TABLE [CashPayments] (
    [PaymentId] int NOT NULL IDENTITY,
    [Amount] decimal(18, 2) NOT NULL,
    [Discriminator] nvarchar(max) NOT NULL,
    [ReceiptCode] nvarchar(max),
    CONSTRAINT [PK_CashPayments]
            PRIMARY KEY ([PaymentId])
);
```

The Discriminator column holds the name of the class, which EF Core uses to define what sort of data is saved. When set by convention, this column holds the name of the class as a string.

The ReceiptCode column is used only if it's a PaymentCredit.

As you can see, EF Core has added a Discriminator column, which it uses when returning data to create the correct type of class: `PaymentCash` or `PaymentCard`, based on what was saved. Also, the `ReceiptCode` column is filled/read only if the class type is `PaymentCard`.

Any scalar properties not in the TPH base class are mapped to nullable columns because those properties are used by only one version of the TPH's classes. If you have lots of classes in your TPH classes, it's worth seeing whether you can combine similar typed properties to the same column. In the `Product` TPH classes, for example, you might have a `Product` type `"Sealant"` that needs a double `MaxTemp` and another `Product` type, `"Ballast"`, that needs a double `WeightKgs`. You could map both properties to the same column by using this code snippet:

```
public class Chapter08DbContext : DbContext
{
    //… other part left out

    Protected override void OnModelCreating
         (ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Sealant>()
```

```
            .Property(b => b.MaxTemp)
            .HasColumnName("DoubleValueCol");

        modelBuilder.Entity<Ballast>()
            .Property(b => b.WeightKgs)
            .HasColumnName("DoubleValueCol");
    }
}
```

USING THE FLUENT API TO IMPROVE OUR TPH EXAMPLE

Although the By Convention approach reduces the number of tables in the database, you have two separate `DbSet<T>` properties, and you need to use the right one to find the payment that was used. Also, you don't have a common `Payment` class that you can use in any other entity classes. But by doing a bit of rearranging and adding some Fluent API configuration, you can make this solution much more useful.

Figure 8.12 shows the new arrangement. You create a common base class by having an abstract class called `Payment` that the `PaymentCash` and `PaymentCard` inherit from. This approach allows you to use the `Payment` class in another entity class called `SoldIt`.

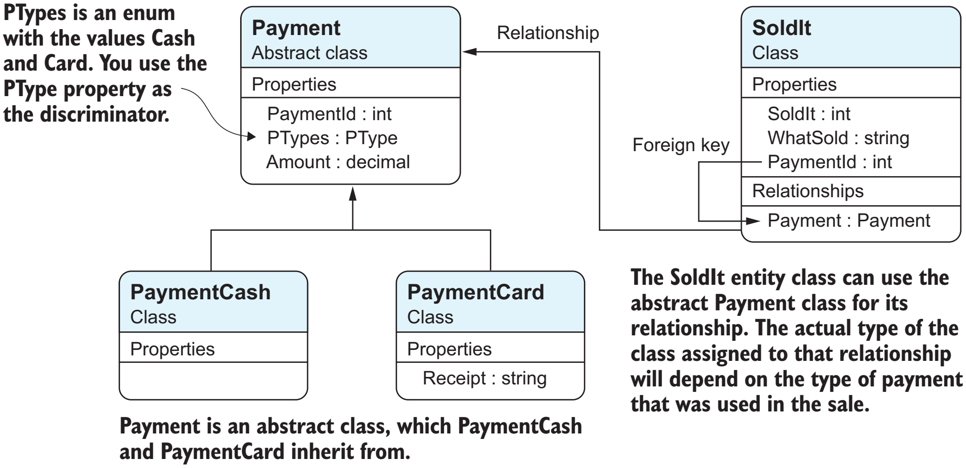


Figure 8.12 By using the Fluent API, you can create a more useful form of the TPH. Here, an abstract class called `Payment` is used as the base, and this class can be used inside another entity class. The actual class type placed in the `SoldIt` payment property will be either `PaymentCash` or `PaymentCard`, depending on what was used when the `SoldIt` class was created.

This approach is much more useful because now you can place a `Payment` abstract class in the `SoldIt` entity class and get the amount and type of payment, whether it's cash or a card. The `PType` property tells you the type (the `PType` property is of type `PTypes`, which is an enum with values `Cash` or `Card`), and if you need the `Receipt` property in the `PaymentCard`, you can cast the `Payment` class to the type `PaymentCard`.

In addition to creating the entity classes shown in figure 8.12, you need to change the application's DbContext and add some Fluent API configuration to tell EF Core about your TPH classes, as they no longer fit the By Convention approach. This listing shows the application's DbContext, with the configuration of the Discrimination column.

Listing 8.24 Changed application's DbContext with Fluent API configuration added

```
public class Chapter08DbContext : DbContext
{
    //… other DbSet<T> properties removed
    public DbSet<Payment> Payments { get; set; }

    public DbSet<SoldIt> SoldThings { get; set; }

    public Chapter08DbContext(
        DbContextOptions<Chapter08DbContext> options)
        : base(options)
    { }

    protected override void OnModelCreating
         (ModelBuilder modelBuilder)
    {
        //… other configurations removed
        modelBuilder.Entity<Payment>()
            .HasDiscriminator(b => b.PType)
            .HasValue<PaymentCash>(PTypes.Cash)
            .HasValue<PaymentCard>(PTypes.Card);
    }
}
```

Defines the property through which you can access all the payments, both PaymentCash and PaymentCard

List of sold items, with a required link to Payment

The HasDiscriminator method identifies the entity as a TPH and then selects the property PType as the discriminator for the different types. In this case, it's an enum, which you set to be bytes in size.

Sets the discriminator value for the PaymentCash type

Sets the discriminator value for the PaymentCard type

> **NOTE** This example uses an abstract class as the base class, which I think is more useful, but it could just as well keep the original `PaymentCash`, with the `PaymentCard` inheriting from it. An abstract base class makes it easier to alter the common TPH properties.

ACCESSING TPH ENTITIES

Now that you've configured a TPH set of classes, let's cover any differences in CRUD operations. Most EF database access commands are the same, but a few changes access the TPH parts of the entities. EF Core does a nice job (as EF6.x did) of handling TPH.

First, the creation of TPH entities is straightforward. You create an instance of the specific type you need. The following code snippet creates a `PaymentCash` type entity to go with a sale:

```
var sold = new SoldIt()
{
    WhatSold = "A hat",
    Payment = new PaymentCash {Amount = 12}
};
context.Add(sold);
context.SaveChanges();
```

Then EF Core saves the correct version of data for that type and sets the discriminator so that it knows the TPH class type of the instance. When you read back the `SoldIt` entity you just saved, with an `Include` to load the `Payment` navigational property, the type of the loaded `Payment` instance will be the correct type (`PaymentCash` or `PaymentCard`), depending on what was used when you wrote it to the database. Also, in this example the `Payment`'s property `PType`, which you set as the discriminator, tells you the type of payment: `Cash` or `Card`.

When you query TPH data, the EF Core `OfType<T>` method allows you to filter the data to find a specific class. The query `context.Payments.OfType<PaymentCard>()` would return only the payments that used a card, for example. You can also filter TPH classes in `Includes`. See this article for more information: http://mng.bz/QmBj.

8.9.3 Table per Type (TPT): Each class has its own table

The EF Core 5 release added the table per type (TPT) option, which allows each entity class inherited from a base class to have its own table. This option is the opposite of the table per hierarchy (TPH) approach covered in section 8.9.2. TPT is a good solution if each class in the inherited hierarchy has lots of different information; TPH is better when each inherited class has a large common part and only a small amount of per-class data.

As an example, you will build a TPT solution for two types of containers: shipping containers used on bulk carrier ships and plastic containers such as bottles, jars, and boxes. Both types of containers have an overall height, length, and depth, but otherwise, they are different. The following listing shows the three entity classes, with the base `Container` abstract class and then the `ShippingContainer` and `PlasticContainer`.

Listing 8.25 The three classes used in the TPT example

```
public abstract class Container
{
    [Key]
    public int ContainerId { get; set; }

    public int HeightMm { get; set; }
    public int WidthMm { get; set; }
    public int DepthMm { get; set; }
}

public class ShippingContainer : Container
{
    public int ThicknessMm { get; set; }
    public string DoorType { get; set; }
    public int StackingMax { get; set; }
    public bool Refrigerated { get; set; }
}

public class PlasticContainer : Container
{
```

The Container class is marked as abstract because it won't be created.

Becomes the primary key for each TPT table

Common part of each container is the overall height, width, and depth

The class inherits the Container class.

These properties are unique to a shipping container.

```
    public int CapacityMl { get; set; }
    public Shapes Shape { get; set; }
    public string ColorARGB { get; set; }
}
```

These properties are unique to a plastic container.

Next, you need to configure your application's DbContext, which has two parts: (a) adding a `DbSet<Container>` property, which you will use to access all the containers, and (b) setting the other container types, `ShippingContainer` and `PlasticContainer`, to map to their own tables. The following listing shows these two parts.

Listing 8.26 The updates to the application's DbContext to set up the TPT containers

```
public class Chapter08DbContext : DbContext
{
    public Chapter08DbContext(
        DbContextOptions<Chapter08DbContext> options)
        : base(options)
    { }

    //… other DbSet<T> removed for clarity
    public DbSet<Container> Containers { get; set; }

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
        //… other configrations removed for clarity

        modelBuilder.Entity<ShippingContainer>()
            .ToTable(nameof(ShippingContainer));
        modelBuilder.Entity<PlasticContainer>()
            .ToTable(nameof(PlasticContainer));
    }
```

This single DbSet is used to access all the different containers.

These Fluent API methods map each container to a different table.

The result of the update to the application's DbContext is three tables:

- *A* `Containers` table, via the DbSet, that contains the common data for each entry
- *A* `ShippingContainer` table containing the `Container` and `ShippingContainer` properties
- *A* `PlasticContainer` table containing the `Container` and `PlasticContainer` properties

You add a `ShippingContainer` and `PlasticContainer` in the normal way: by using the `context.Add` method. But the magic comes when you query the `DbSet<Container> Containers` in the application's DbContext, because it returns all the containers using the correct class type, `ShippingContainer` or `PlasticContainer`, for each entity returned.

You have a few options for loading one type of the TPT classes. Here are three approaches, with the most efficient at the end:

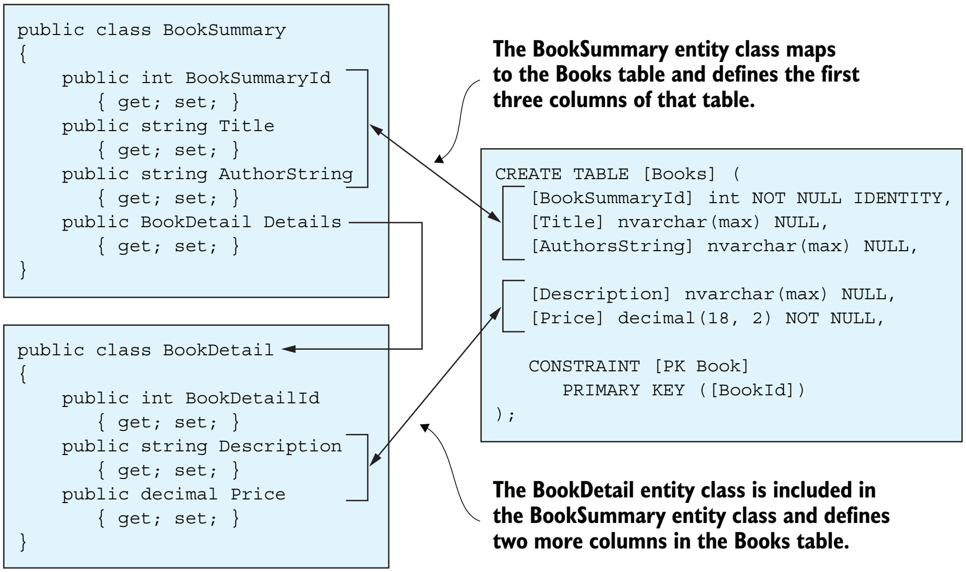
- *Read all query*—`context.Containers.ToList()`
  This option reads in all the TPT types, and each entry in the list will be of the correct type (`ShippingContainer` or `PlasticContainer`) for the type it returns. This option is useful only if you want to list a summary of all the containers.

- `OfType` query—`context.Containers.OfType<ShippingContainer>().ToList()`
  This option reads in only the entries that are of the type `ShippingContainer`.
- `Set` query—`context.Set<ShippingContainer>().ToList()`
  This option returns only the `ShippingContainer` type (just like the `OfType` query), but the SQL is slightly more efficient than the `OfType` query.

8.9.4 Table splitting: Mapping multiple entity classes to the same table

The next feature, called *table splitting*, allows you to map multiple entities to the same table. This feature is useful if you have a large amount of data to store for one entity, but your normal queries to this entity need only a few columns. Table splitting is like building a `Select` query into an entity class; the query will be quicker because you're loading only a subsection of the whole entity's data. It can also make updates quicker by splitting the table across two or more classes.

This example has two entity classes, `BookSummary` and `BookDetail`, both of which map to a database table called Books. Figure 8.13 shows the result of configuring these two entity classes as a table split.



```
public class BookSummary
{
    public int BookSummaryId
       { get; set; }
    public string Title
       { get; set; }
    public string AuthorString
       { get; set; }
    public BookDetail Details
       { get; set; }
}
```

```
public class BookDetail
{
    public int BookDetailId
       { get; set; }
    public string Description
       { get; set; }
    public decimal Price
       { get; set; }
}
```

```
CREATE TABLE [Books] (
    [BookSummaryId] int NOT NULL IDENTITY,
    [Title] nvarchar(max) NULL,
    [AuthorsString] nvarchar(max) NULL,

    [Description] nvarchar(max) NULL,
    [Price] decimal(18, 2) NOT NULL,

   CONSTRAINT [PK Book]
      PRIMARY KEY ([BookId])
);
```

The BookSummary entity class maps to the Books table and defines the first three columns of that table.

The BookDetail entity class is included in the BookSummary entity class and defines two more columns in the Books table.

Figure 8.13 The result of using the table-splitting feature in EF Core to map two entity classes, `BookSummary` and `BookDetail`, to one table, Books. You do this because a book needs a lot of information, but most queries need only the `BookSummary` part. The effect is to build a preselected set of columns for faster querying.

Here's the configuration code.

Listing 8.27 Configuring a table split between `BookSummary` and `BookDetail`

```
public class SplitOwnDbContext : DbContext
{
    //… other code removed

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<BookSummary>()
            .HasOne(e => e.Details)
            .WithOne()
            .HasForeignKey<BookDetail>
                  (e => e.BookDetailId);
        modelBuilder.Entity<BookSummary>()
            .ToTable("Books");

        modelBuilder.Entity<BookDetail>()
            .ToTable("Books");
    }
}
```

Defines the two books as having a relationship in the same way that you'd set up a one-to-one relationship

In this case, the HasForeignKey method must reference the primary key in the BookDetail entity.

You must map both entity classes to the Books table to trigger the table splitting.

After you've configured the two entities as a table split, you can query the `BookSummary` entity on its own and get the summary parts. To get the `BookDetails` part, you can either query the `BookSummary` entity and load the `Details` relationship property at the same time (say, with an `Include` method) or read only the `BookDetails` part straight from the database.

> **NOTE** In part 3 of this book, I build a much more complex Book App, using real book data from Manning Publications. I use table splitting to separate the large descriptions used in the detailed book view from the main part of the Book data. Any updates of, say, the `Book`'s `PublishedOn` property are much quicker because I don't have to read in all the descriptions.

Let me make a few points before leaving this topic:

- You can update an individual entity class in a table split individually; you don't have to load all the entities involved in a table split to do an update.
- You've seen a table split to two entity classes, but you can table-split any number of entity classes.
- If you have concurrency tokens (see section 10.6.2), they must be in all the entity classes mapped to the same table to make sure that the concurrent token values are not out of data when only one of the entity classes mapped to the table is updated.

8.9.5 Property bag: Using a dictionary as an entity class

EF Core 5 added a feature called a *property bag* that uses a `Dictionary<string, object>` type to map to the database. A property bag is used to implement the direct many-to-many relationship feature, where the linking table had to be created at configuration time. You can also use a property bag, but it is useful only in specific areas, such as creating a property-bag entity in a table whose structure is defined by external data.

> **NOTE** A property bag uses two features that aren't described elsewhere in this book. The first feature is *shared entity types*, where the same type can be mapped to multiple tables. The second feature uses a C# indexer property in an entity class to access data, such as `public object this[string key]` … .

As an example, you map a property bag to a table whose name and columns are defined by external data rather than by the structure of a class. For this example, the table is defined in a `TableSpec` class, which is assumed to have been read in on startup, maybe from an appsettings.json file. The following listing shows the application's DbContext with the necessary code to configure and access a table via a property-bag entity.

Listing 8.28 Using a property-bag `Dictionary` to define a table on startup

```
public class PropertyBagsDbContext : DbContext
{
    private readonly TableSpec _tableSpec;

    public PropertyBagsDbContext(
        DbContextOptions<PropertyBagsDbContext> options,
        TableSpec tableSpec)
        : base(options)
    {
        _tableSpec = tableSpec;
    }

    public DbSet<Dictionary<string, object>> MyTable
        => Set<Dictionary<string, object>>(_tableSpec.Name);

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
        modelBuilder.SharedTypeEntity
            <Dictionary<string, object>>(
                _tableSpec.Name, b =>
        {
            foreach (var prop in _tableSpec.Properties)
            {
                var propConfig = b.IndexerProperty(
                    prop.PropType, prop.Name);
                if (prop.AddRequired)
                    propConfig.IsRequired();
            }
        }).Model.AddAnnotation("Table", _tableSpec.Name);
    }
}
```

You pass in a class containing the specification of the table and properties.

The DbSet called MyTable links to the SharedType entity built in OnModelCreating.

Defines a SharedType entity type, which allows the same type to be mapped to multiple tables

You give this shared entity type a name so that you can refer to it.

Adds each property in turn from the tableSpec

Adds an index property to find the primary key based on its name

Sets the property to not being null (needed only on nullable types such as string)

Now you map to the table you want to access.

To be clear, the data in the `TableSpec` class must be the same every time because EF Core caches the configuration. The property-bag entity's configuration is fixed for the whole time the application is running. To access the property-bag entity, you use the `MyTable` property shown in the next listing. This listing shows adding a new entry via a dictionary and then reading it back, including accessing the property bag's properties in a LINQ query.

Listing 8.29 Adding and querying a property bag

```
var propBag = new Dictionary<string, object>
{
    ["Title"] = "My book",
    ["Price"] = 123.0
};
context.MyTable.Add(propBag);
context.SaveChanges();

var readInPropBag = context.MyTable
    .Single(x => (int)x["Id"] == 1);

var title = readInPropBag["Title"];
```

The property bag is of type Dictionary<string, object>.

Sets the various properties using the normal dictionary approaches

For a shared type, such as a property bag, you must provide the DbSet to Add to.

The property-bag entry is saved in the normal way.

To read back, you use the DbSet mapped to the property-bag entity.

To refer to a property/column, you need to use an indexer. You may need to cast the object to the right type.

You access the result by using normal dictionary access methods.

This listing is a specific example in which a property bag is a good solution, but you can configure a property bag manually. Here is some more information on the property bag:

- A property bag's property names follow By Convention naming. The primary key is `"Id"`, for example. But you can override this setting with Fluent API commands as usual.
- You can have multiple property bags. The `SharedTypeEntity` Fluent API method allows you to map the same type to different tables.
- A property bag can have relationships to other classes or property bags. You use the `HasOne`/`HasMany` Fluent API methods, but you can't define navigational properties in a property bag.
- You don't have to set every property in the dictionary when you add a property-bag entity. Any properties/columns not set will be set to the type's default value.

Summary

- If you follow the By Convention naming rules for foreign keys, EF Core can find and configure most normal relationships.
- Two Data Annotations provide a solution to a couple of specific issues related to foreign keys with names that don't fit the By Convention naming rules.
- The Fluent API is the most comprehensive way to configure relationships. Some features, such as setting the action on deletion of the dependent entity, are available only via the Fluent API.

- You can automate some of the configuration of your entity classes by adding code that is run in the DbContext's `OnModelCreating` method.
- EF Core enables you to control updates to navigational properties, including stopping, adding, or removing entries in collection navigational properties.
- EF Core provides many ways to map entity classes to a database table. The main ones are owned types, table per hierarchy, table per type, table splitting, and property bags.

For readers who are familiar with EF6:

- The basic process of configuring relationships in EF Core is the same as in EF6.x, but the Fluent API commands have changed significantly.
- EF6.x adds foreign keys if you forget to add them yourself, but they aren't accessible via normal EF6.x commands. EF Core allows you to access them via shadow properties.
- The EF Core 5 release added a similar feature as EF6.x's many-to-many relationship, with EF Core now automatically creating the linking table (see section 3.4.4), but EF Core's implementation is different from how EF6.x implements this feature.
- EF Core has introduced new features, such as access to shadow properties, alternate keys, and backing fields.
- EF Core's owned types provide features you would have found in EF6.x's complex types. Extra features include storing owned types in their own table.
- EF Core's TPH, TPT, and table-splitting feature are similar to the corresponding features in EF6.x, but owned types and property bags aren't in EF6.x.

9

Handling database migrations

This chapter covers

- Different ways to create commands to update a database's structure
- Three starting points from which you create database structure changes
- How to detect and fix database structure changes that would lose data
- How the characteristics of your application affect the way you apply a database change

This chapter covers ways of changing the structure of a database, referred to as migrating a database. The structure of the database is called the *database schema*; it consists of the tables, columns, constraints, and so on that make up a database. Creating and updating a database schema can seem to be simple because EF Core provides a method called `Migrate` to do it all for you: you create your entity classes and add a bit of configuration, and EF Core builds you a nice, shiny database.

The problem is that EF Core's `Migrate` method hides a whole series of database migration issues that aren't immediately obvious. Renaming a property in an entity class, for example, by default causes that property's database column to be deleted, along with any data it had! So in this chapter, in addition to detailing how to build and apply database migrations, I cover the key issues that you must consider when updating a database. No one wants to be the person who breaks your "live" database.

The EF Core documentation on migrations is excellent (see http://mng.bz/XdR6), so this chapter doesn't try to duplicate that information. Instead. it delves into the options and issues related to migrating a database, along with their pros and cons. You have many ways to create and apply database migrating, and this chapter covers the various options. The chapter also contains examples of handling the more complex issues, such as properly handing migrations that could lose data and applying a migration to a database while the application is still running. This knowledge will help you select the right approach to creating a migration and successfully apply it to a database.

9.1 How this chapter is organized

This chapter starts with section 9.2, which introduces the topic of databases that need migrating and the important issue of ensuring that no data is lost while migrating a database. After that section, two parts cover creating and applying migrations:

- Part 1, creating a database migration, starts at section 9.3. This part covers the three approaches to creating database migrations or creating your EF Core classes and configuration to match an existing database.
- Part 2, applying a migration to a database, starts at section 9.8. This part covers the ways you can apply a migration to a production database, including the complexities of updating a database while the application is still running.

These parts cover lots of approaches for you to consider. Each part has a table that lists the pros, cons, and limitations of each given approach, which should help you make the right choice for your project.

9.2 Understanding the complexities of changing your application's database

This section talks about the issues involved in migrating a database, especially the database that your live application is using. The topics covered in this section are general to all relational databases and any software system. There are many ways to organize your database and application deployment, each with trade-offs of complexity, scalability, availability, and development/operations (DevOps) effort.

Combining the information in this chapter with your knowledge of your application, you can decide which approach to use to create and migrate your databases. Having a thought-through plan or policy for creating and applying migrations will make the migration process safer and quicker.

9.2.1 A view of what databases need updating

Before I describe how to update a database's schema, let's look at the databases that can be involved in an application being developed. Figure 9.1 shows a possible arrangement of a multiperson development team, with development, testing, preproduction, and production.

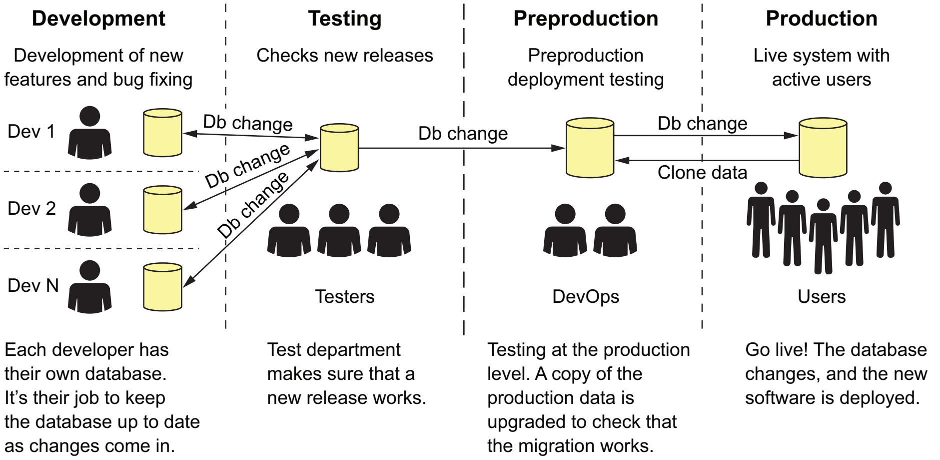


Figure 9.1 Various databases can be used in an application's development, all of which will need database schema changes applied to them. The terms development, testing, preproduction, and production refer to different parts of the development, testing, and deployment of an application, and any associated database schema changes.

Not all development projects have all these stages, and some have more or different stages. Also, this figure assumes that only one database is being used in production, but you may have multiple copies of the same database. You may also have developers sharing a single development database, but that approach has some limitations; see the following note. The permutations are endless. This chapter refers to the development and production databases, but be aware that database schema updates may be needed on other databases.

> **NOTE** Using a single shared database in the development environment can work, but it has limitations. A developer might apply a migration to the database before they merge the code into the main branch, for example, which could cause problems. Section 9.2.2 introduces the topic of migrations that might cause problems.

9.2.2 Handling a migration that can lose data

It's helpful to characterize migrations in two groups: a nonbreaking change or a data-loss breaking change. A *nonbreaking change* is one that doesn't remove tables or columns that have useful data in them, and a *data-loss breaking change* removes those tables or columns. So if you don't want to lose important data, you need to add an extra copy stage to a data-loss breaking change migration so that the data is preserved.

Fortunately, in applications that are being developed, many migrations are the nonbreaking-change type because you are adding new tables and columns to your database. But at times, you want to restructure your database such that you need to move columns in one table to another, possibly new table. Section 9.5 gives two examples of data-loss breaking changes and how to fix them:

- Renaming a property (section 9.5.1)
- Moving columns from one table to another (section 9.5.2)

NOTE Section 9.8 discusses another type of breaking change: an application breaking change, which refers to a migration that would cause errors in the currently running application. This change matters if you are trying to migrate a database while the current application is running.

9.3 Part 1: Introducing the three approaches to creating a migration

Section 9.2 applies to any form of database migration, but from now, on the focus is on EF Core. This focus is important because the job isn't only to change the database; it's also to ensure that the changed database matches the entity classes and the EF Core configuration held by the application's DbContext. If you use EF Core's migration tools, it's a given that the database will match the application's DbContext, but as you will see, that match isn't guaranteed in many other approaches to migrating a database.

You have three main ways to come up with an updated database that matches your application's DbContext. Each approach has a different starting point, which Arthur Vickers (engineering manager of the EF Core team) calls *the source of truth:*

- *Using EF Core's migration features*—This approach considers the entity classes, and the EF Core configuration is the source of truth. This approach to handling migrations is the easiest one, but complex issues such as handling data-loss breaking changes require you to hand-edit migrations.
- *Using SQL scripts to build migrations*—In this approach, the source of truth is the SQL commands used to build/migrate the database. You have complete control of your database schema and can include features that EF Core doesn't configure, such as column-level constraints. But the big challenge is matching your SQL changes to EF Core's internal model.
- *Using EF Core's reverse-engineering tool*—In this approach, the database is the source of truth. You re-create the entity classes and the application's DbContext

with all the required configurations. You'd use this approach mainly to build an EF Core application around an existing database.

Figure 9.2 gives you an overview of the five ways to migrate a database and their key attributes. Each section discussing a migration starts with a table summarizing the approach and including my views about when I think the approach is useful.

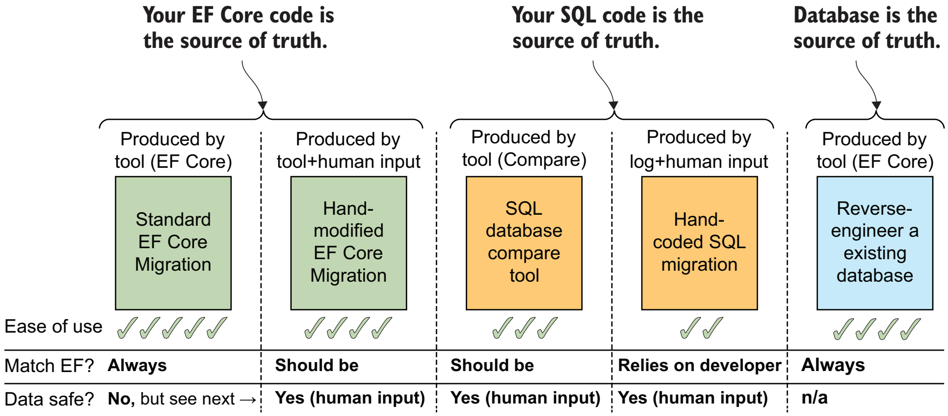


Figure 9.2 A simple summary of the five ways to migrate a database and make sure that the database matches EF Core's internal model of the database

9.4 Creating a migration by using EF Core's add migration command

EF Core's migration tools are the standard way to create and update a database from EF Core. This approach is easiest because it automatically builds the correct SQL commands to update the database, saving you from digging into databases and the SQL language to create and change the application's database.

You start by studying the standard migration produced by EF Core migration tools with no extra editing by you. A standard migration can handle most situations and forms the basis for altering the migration if you need to. Typically, you would need to edit a migration to handle things such as data-loss breaking changes (section 9.5) after you review what the standard can do.

You create a standard migration by using EF Core's migration tools—specifically, the `add migration` command. This command uses the entity classes and the application's DbContext, with its configuration code being the source of truth. But the add migration commands also needs to know the previous state of EF Core's model of the database to decide what needs changing. It does this by looking at a class created by the last run of the EF Core migration tools, which contain a snapshot of EF Core's model of the database. For the first migration, that class won't exist, so the migration

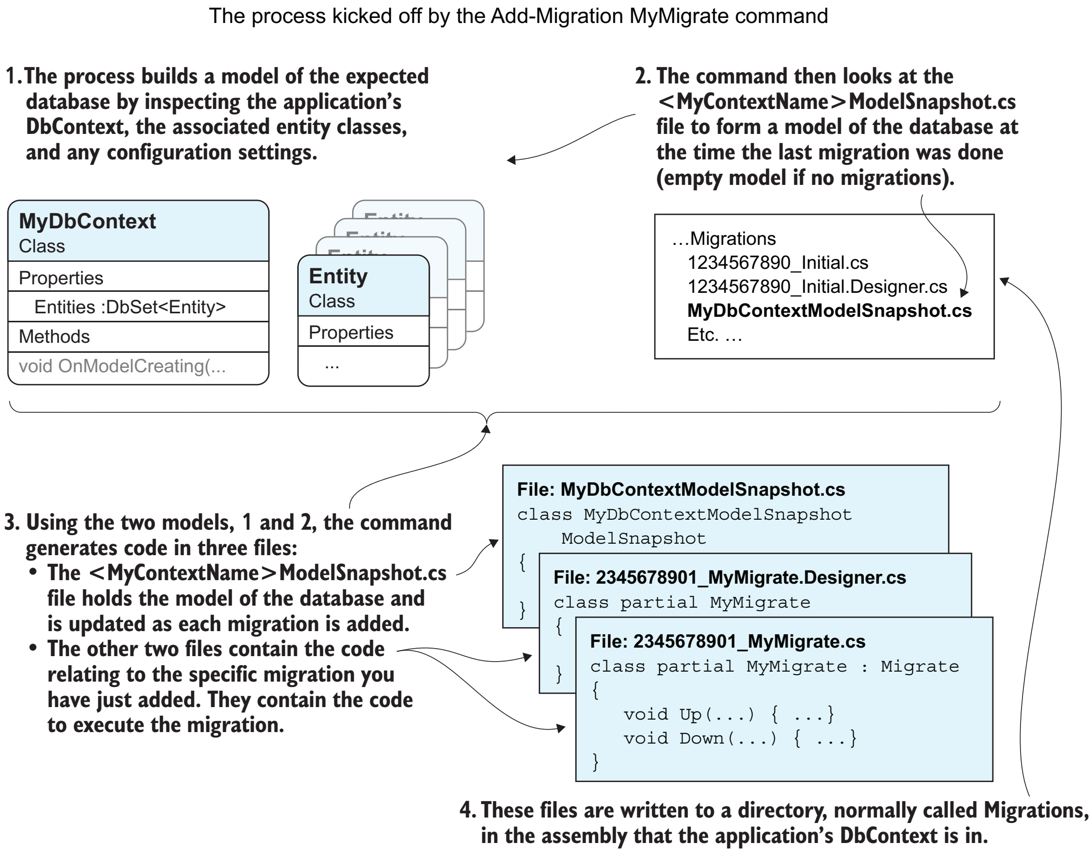


Figure 9.3 Running the `add migration command` to create a new EF Core migration. The command compares two models of the database. One model comes from our current application, with its DbContext, entity classes, and EF Core configuration; the other is from the <MyContextName>ModelSnapshot.cs file (which is empty if this mirgration is your first one). By comparing these two models, EF Core can create code that will update the database schema to match EF Core's current database model.

tools assume that the database has an empty schema—that is, has no tables, indexes, and so on. So when you run the EF Core's `add migration` command, it compares the snapshot class with your current entity classes and the application's DbContext with its configuration code. From that data, it can work out what has changed; then it builds two classes containing the commands to add the changes to the database (figure 9.3).

Don't build your entity classes the same way you build normal classes

EF Core is great at making your database look like normal classes, but you shouldn't build your entity classes quite the same way that you would your normal classes. In normal classes, for example, a good approach to stopping duplication is using

(continued)

properties that access other properties, known as *expression body definitions*. Here's an example:

```
public string FullName => $"{FirstName} {LastName}";
```

That technique works for a normal class, but if you use it for an entity class, a query that filters or sorts on the `FullName` property would fail. In this case, you would need to provide a real property linked to a database column (possibly using the new, persisted computed column; see chapter 10) to make sure that EF Core can sort/filter on that data.

Also, you should think carefully about what properties and relational links you put in an entity class. Refactoring a normal class is easy, but refactoring an entity class requires a migration, possibly including a data-copying stage too.

Remember that your entity classes, with their navigational properties, define the database's structure. Just because EF Core makes it easy to define these things doesn't mean that you shouldn't think about the database structure and its performance.

Before you delve into the `add migration` command, table 9.1 summarizes using a standard migration to update your database's schema. Each section on a migration approach has a table similar to table 9.1 so that you can compare the features and limitations of each approach.

Table 9.1 A summary of the good, the bad, and the limitations of a standard migration created by the `add migration` command

	Notes
Good parts	- Builds a correct migration automatically - Handles seeding of the databas - Doesn't require knowledge of SQL - Includes a remove migration feature (see section 9.4.4)
Bad parts	- Only works if your code is the source of truth
Limitations	- Standard EF Core migrations cannot handle breaking changes (but see section 9.5). - Standard EF Core migrations are database-specific (but see section 9.5.4).
Tips	Watch out for error messages when you run the `add migration` command. If EF Core detects a change that could lose data, it outputs an error message but still creates the migration files. You *must* alter the migration script; otherwise, you will lose data (see section 9.5.2).
My verdict	This approach is an easy way to handle migrations, and it works well in many cases. Consider this approach first if your application code is driving the database design.

TIP I recommend an EF Core Community Standup video that covers some EF Core 5's features and the philosophy behind EF Core's migration features; see http://mng.bz/yYmq.

9.4.1 Requirements before running any EF Core migration command

To run any of the EF Core migration tools' commands, you need to install the required code and set up your application in a certain way. There are two versions of the EF Core migration tools: the `dotnet-ef` command-line interface (CLI) tools and Visual Studio's Package Manager Console (PMC) version.

To install the CLI tools, you need to install them on your development computer via the appropriate command prompt. The following command will install the `dotnet-ef` tools globally so that you can use them in any directory:

```
dotnet tool install --global dotnet-ef
```

To use Visual Studio's PMC feature, you must include the NuGet package `Microsoft.EntityFrameworkCore.Tools` in your main application, and the correct EF Core database provider NuGet package, such as `Microsoft.EntityFrameworkCore.SqlServer`, in the project that holds the application's DbContext you want to migrate.

These tools must be able to create an instance of the DbContext you want to migrate. If your startup project is an ASP.NET Core web host or .NET Core generic host, the tools can use it to get an instance of a DbContext set up in the startup class.

If you aren't using ASP.NET Core, you can add a class that implements the `IDesignTimeDbContextFactory<TContext>` interface. This class must be in the same project as the DbContext you want to migrate. The following listing shows an example taken from the `Part2` branch of the associated GitHub repo.

Listing 9.1 A class that provides an instance of the DbContext to the migration tools

```
public class DesignTimeContextFactory
    : IDesignTimeDbContextFactory<EfCoreContext>
{
    private const string connectionString =
        "Server=(localdb)\\mssqllocaldb;Database=..."

    public EfCoreContext CreateDbContext(string[] args)
    {
        var optionsBuilder =
            new DbContextOptionsBuilder<EfCoreContext>();
        optionsBuilder.UseSqlServer(connectionString);

        return new EfCoreContext(optionsBuilder.Options);
    }
}
```

EF Core tools use this class to obtain an instance of the DbContext.

This interface defines a way that the EF Core tools find and create this class.

The interface requires this method, which returns a valid instance of the DbContext.

You need to provide a connection string to your local database.

You use the normal commands to set up the database provider you are using.

Returns the DbContext for the EF Core tools to use

9.4.2 Running the add migration command

To create an EF Core migration, you need to run the `add migration` command from a command line (CLI tools) or in Visual Studio's PMC window. The two ways to migrate a database, CLI tools and PMC, have different names and parameters. The following list shows an `add migration` command that I used to create a migration in the Book App. Note that the CLI version was run in the directory of the BookApp ASP.NET Core project:

- *CLI*—`dotnet ef migrations add Ch09Migrate -p ../DataLayer`
- *PMC*—`Add-Migration Ch09Migrate -Project DataLayer`

> **NOTE** There are lots of commands, with multiple parameters, and it would take many pages to reproduce the EF Core documentation. Therefore, I direct you to EF Core's command-line reference at http://mng.bz/MXEn.

9.4.3 Seeding your database via an EF Core migration

EF Core's migrations can contain data that will be added to the database, a process known as seeding the database. A good use of this feature is adding constants to your database, such as your product types and customer types for an e-commerce site. I should say that seeded data can be changed, so the data isn't a constant, but you can change it only via a migration, so it's best to use it for data that doesn't change (much).

> **NOTE** As well as adding the seed data when a migration is applied, the `context.Database.EnsureCreated()` method (usually used in unit testing) seeds the created database. See chapter 17 for more on unit testing.

You add seed data via Fluent API configuration, using the `HasData` method. Listing 9.2 gives an example of ways you can link seed data via its primary and foreign keys. This example has seed data that is more complex than I usually have, but I'm providing it to show you the various ways that you can set up seed data. The classes used in this example are

- A `Project` entity class with a `ProjectManager` of type `User`
- The `User` entity class, which holds the user's `Name` and address
- The `Address` class, an owned type (see section 8.9.1) that holds the address part

Listing 9.2 An example of setting up seed data via the `HasData` Fluent API method

Seeding is configured via the Fluent API.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Project>().HasData(
        new { ProjectId = 1, ProjectName = "Project1"},
        new { ProjectId = 2, ProjectName = "Project2"});
    modelBuilder.Entity<User>().HasData(
        new { UserId = 1, Name = "Jill", ProjectId = 1 },
        new { UserId = 2, Name = "Jack", ProjectId = 2 });
    modelBuilder.Entity<User>()
        .OwnsOne(x => x.Address).HasData(
```

Adds two default projects. Note that you must provide the primary key.

Each Project and a ProjectManager. Note that you set the foreign key of the project they are on.

The User class has an owned type that holds the User's address.

```
            new {UserId = 1, Street = "Street1", City = "city1"},
            new {UserId = 2, Street = "Street2", City = "city2"});
}
```

Provide the user's addresses. Note that you use the UserId to define which user you are adding data to.

As you can see from listing 9.2, you must define the primary key, even if is usually generated by the database, so that you can define relationships by setting foreign keys to the appropriate primary key. And if you change the primary key, the previous seeded entry is removed. Also, if you keep the original primary key but change the data in that entry, the migration will update that entry.

NOTE The directory Chapter09Listings\SeedExample in the Test project of the associated GitHub repo contains an example of what happens when you change your seed data between migrations. The second migration contains code to delete, update, and insert seed data due to changes in the `HasData` parts.

9.4.4 Handling EF Core migrations with multiple developers

When multiple developers are working on a project that uses EF Core's migration feature to update the database schema, you might bump into software merges in which one developer's migration is in conflict with your migration. This section gives you some advice on what to do. I do assume that you are using source control and that you have your own development database to try out a migration locally.

First, if your migration has no conflicts with a migration that you just merged into your software, you shouldn't have a source control conflict, because EF Core's migrations are designed to be team-friendly (unlike EF6 migration code). You might get migrations applied in a slightly different order; perhaps you created your migration yesterday, and someone's else's migration was produced today and applied to the main database. That situation shouldn't cause a problem if there are no merge conflicts, because EF Core can handle out-of-order migrations.

You will know if you have a migration merge conflict because your source control system will show a conflict in the migration snapshot file, which has the name <DbContextClassName>ModelSnapShot. If this conflict happens, here's the recommended way to fix it:

1. Abort the source control merge that contained a migration change that conflicted with your migration.
2. Remove the migration you created by using either of the following commands (*Note:* Keep the entity classes and configuration changes; you will need them later):
   a. *CLI*—`dotnet ef migrations remove`
   b. *PMC*—`Remove-Migration`
3. Merge the incoming migration you abandoned in step 1. A merge conflict should no longer appear in the migration snapshot file.
4. Use the `add migration` command to re-create your migration.

That migration conflict resolution process works in most cases, but it can get complex. My recommendation for projects in which migration conflicts can happen are

- Merge the main/production branch into your local version before you create a migration.
- Have only one migration in a source control merge into your main/production branch, because undoing two migrations is hard work.
- Tell your development team members if you think that your migration might affect their work.

9.4.5 Using a custom migration table to allow multiple DbContexts to one database

EF Core creates a table if you apply an EF Core migration to a database. EF Core uses this table to find out what migrations have been applied to the database so that it knows what migration should be applied to the database you are migrating. By default, that table is called __EFMigrationsHistory, but you can change the name via an option method called `MigrationsHistoryTable`.

There aren't many reasons for changing the migration history table, but sharing a database across multiple EF Core DbContexts is one of them. Here are two examples:

- *Saving money by combining databases*—You are building an ASP.NET Core application with individual user accounts that needs an accounts database. Your application's DbContext also needs a database. By using a custom migration table on your application's DbContext would allow both contexts to use the same database.
- *Using a separate DbContext for each business group*—In part 3 of this book, I want to make the project easier to extend as it gets bigger. Therefore, I have separate DbContexts: one for the book-display code and another for the order-processing code.

Both examples work, but using EF Core's migration system with either takes a bit more effort. The first migration example—saving money by combining databases—is easier because the two databases you are combining don't share any tables, views, and so on. But because both databases use EF Core's migration system, they need a different migration history table. ASP.NET Core's individual user account database uses the default name for the migration history table, so you need to change the name of your application's DbContext. The next listing shows how to do that when you are registering your application's DbContext in ASP.NET Core's startup class.

Listing 9.3 Changing the name of the migration history table for your DbContext

Registers your application's DbContext as a service in ASP.NET Core

The second parameter allows you to configure at the database provider level.

The MigrationsHistoryTable method allows you to change the migration table name and optionally the table's schema.

```
services.AddDbContext<EfCoreContext>(
    options => options.UseSqlServer(connection,
        dbOptions =>
             dbOptions.MigrationsHistoryTable("NewHistoryName")));
```

Next, of course, you must migrate each of the DbContexts—in this case, the ASP.NET Core's individual user account context and your application's DbContext. Otherwise, your job is done.

For the second example—having a separate DbContext for each business group—you need a different migration history table name for each DbContext so that each migration is separate. You should also specify separate directories for the migration classes for each DbContext, which you can do via an option in the `add migration` command. That command will stop any clashes of class names if you use the same migration name in both DbContexts.

> **NOTE** You can also place the migration classes in a separate project if you want to. You need to tell the `add migration` command which project to place the migration in. Then you use the `MigrationsAssembly` method when you set up the database options. See http://mng.bz/aonB.

This example, however, has another problem for you to deal with: each DbContext needs to access the table called Books, which would duplicate the migration of that table. The Books table is shared because both DbContexts must be able to read it (to show the books and create an order for books, respectively).

You have several options to fix this problem, but the best is to use the `ExcludeFromMigrations` Fluent API command, which stops that entity class from being included in a migration. In the `BookDbContext`/`OrderDbContext` example, you could remove the migration of the `Book` entity class in the `OrderDbContext`, as shown in this code snippet:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Book>()
        .ToTable("Books",
             t => t.ExcludeFromMigrations());
}
```

If the `Book` entity class is mapped to a view, not to a table (see section 7.9.3), the migration tools will not include that view in a migration. For this example, that approach is a good one, as we want the `BookDbContext` to have read/write access, but the `OrderDbContext` should only have read access.

My approach to creating migrations

My approach to building migrations relies on having unit tests that can check things against a database. I realize that some developers don't like that approach, but I've found that not being able to unit-test my code against a real database requires me to build/apply a migration and then run the application to test my changes. Using unit tests against a real database makes me develop faster, and each unit test I write improves the coverage of the application I am working on.

Normally, I build a comprehensive set of unit tests across the whole application, other than the final UI/WebAPI side. Many of my unit tests use the database because

(continued)

it’s the quickest way to set up the test data; EF Core makes setting up a test database easy. Sure, for complex business logic I use a repository pattern (see section 4.2), which I can stubb out, but for straightforward queries and updates, I can use test databases. As a result, I can implement a new feature in stages and check as I go by running my unit tests.

This approach does require the databases in unit tests to be up to date with the current EF Code Model; the schema must match your current entity classes and DbContext configuration. Many years of experience (and some suggestions from the EF Core team) have honed my approach, which I share with you in chapter 17. This approach allows me to build a complex feature in smaller steps, with the unit-test databases always in step with EF Core’s current Model. Only after all the code is written and the unit tests pass do I finally create a migration.

9.5 Editing an EF Core migration to handle complex situations

EF Core migration tools are powerful and well thought out, but they can’t handle every possible database migration, such as a data-loss breaking change. The EF Core team knows this, so it provided multiple ways to alter a migration class by hand. Let’s look at the types of migrations that a standard migration can’t handle without help:

- *Data-loss breaking changes,* such as moving columns from one table to a new table
- *Adding SQL features that EF Core doesn’t create,* such as adding user-defined functions, SQL stored procedures, views, and so on
- *Altering a migration to work for multiple database types,* such as handling both SQL Server and PostgreSQL

You can fix these problems by editing the standard migration class created via the `add migration` command. To do this, you need to edit the migration class whose filename ends with the migration name and has a type of .cs, such as …_InitialMigration.cs. In the following sections, you learn the different types of edits that can improve or fix your migrations, but table 9.2 provides a summary of the pros and cons of hand-editing a migration to achieve the required migration.

Table 9.2 A summary of the good, the bad, and the limitations of a migration created by the `add migration` command edited by you to handle situations that the standard migration can’t handle on its own

	Notes
Good parts	▪ You start with most of the migration build via the `add migration` command. ▪ You can customize the migration. ▪ You can add SQL extra features, such as stored procedures.
Bad parts	▪ You need to know more about the database structure. ▪ Some edits require SQL skills.

Table 9.2 A summary of the good, the bad, and the limitations of a migration created by the `add migration` command edited by you to handle situations that the standard migration can't handle on its own *(continued)*

	Notes
Limitations	Your edits aren't checked by EF Core, so you could get a mismatch between the updated database and your entity classes and application's DbContext.
Tips	Same as for standard migrations (see table 9.1)
My verdict	This approach is great for small alterations, but making big changes can be hard work, as you are often mixing C# commands with SQL. If you expect to be editing lots of your migrations to add SQL features, you should consider an SQL script approach (see section 9.6.2) as an alternative.

9.5.1 Adding and removing MigrationBuilder methods inside the migration class

Let's start with a simple example of fixing a migration that contains a data-loss breaking change. This example looks at what happens if you change the name of a property in an entity class, which causes a data-loss breaking change. This problem can be fixed by removing two commands and replacing them with `MigrationBuilder`'s `RenameColumn` method inside the migration class.

This example comes from chapter 7, where you changed the `CustomerId` property in the `Order` entity class to `UserId` to automate adding a Query Filter (see section 7.15.4). The standard migration sees this operation as being the removal of the `CustomerId` property and the addition of a new property called `UserId`, which would cause any existing values in the CustomerId column to be lost. To fix this problem, make the following changes in the migration class generated by the standard migration generated in chapter 7:

- Remove the `AddColumn` command that adds the new `UserId` column.
- Remove the `DropColumn` command that removes the existing `CustomerId` column.
- Add a `RenameColumn` command to rename the `CustomerId` column to `UserId`.

The following listing shows the start of the altered migration class, the name of which is taken from the migration name, `Chapter07`. The methods that need to be removed are commented out, and the new `RenameColumn` method is added.

Listing 9.4 The updated migration class with old commands replaced

```
public partial class Chapter07 : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {

        //migrationBuilder.AddColumn<Guid>(
        //    name: "UserId",
        //    table: "Orders",
```

Migration class created by the add migration command that has been edited

There are two methods in a migration class. Up applied the migration, and another method called Down removed this migration.

The command to add the new UserId column should not run, so you comment it out.

```
        //    type: "uniqueidentifier",
        //    nullable: false,
        //    defaultValue:
        //        new Guid("00000000-0000-0000-0000-000000000000"));


        //migrationBuilder.DropColumn(
        //    name: "CustomerId",
        //    table: "Orders");

        migrationBuilder.RenameColumn(
            name: "CustomerId",
            table: "Orders",
            newName: "UserId");

        //… rest of the migration code left out
    }
}
```

The command to remove the existing CustomerId column should not run, so you comment it out.

The correct approach is to rename the CustomerId column to UserId.

That code will change the `Up` migration from one that loses data to one that preserves the data held in the old CustomerId column. A migration class created by the `add migration` command also contains a `Down` method. This method undoes the migration if the `Up` migration has been applied to a database (see the `remove` command in section 9.4.4). Therefore, it is best practice to edit the `Down` method with the correct commands to undo the migration. The `Down` part that goes with listing 9.4 would also be edited do the reverse from the `Up` part. You would remove the `AddColumn/DropColumn` commands in the `Down` part and replace them with `RenameColumn`, but now the rename is from UserId back to CustomerId.

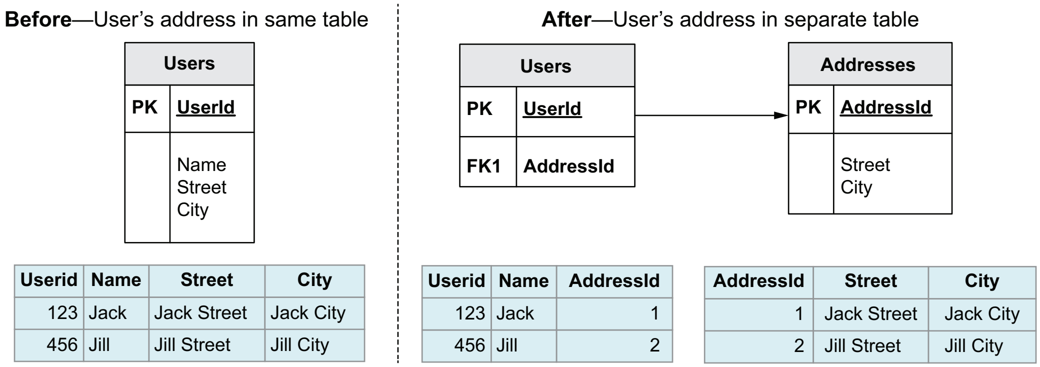
NOTE I haven't shown you the altered `Down` method, but you can find this migration class in the Migrations folder of the DataLayer project in the GitHub repo, branch `Part2`.

9.5.2 Adding SQL commands to a migration

There can be two main reasons for adding SQL commands to a migration: to handle a data-loss breaking change and to add or alter parts of the SQL database that EF Core doesn't control, such as adding views or SQL stored procedures.

As an example of adding SQL commands to a migration, you are going to handle a data-loss breaking change. In this case, you are going to start with a database with a `User` entity class that contains each user's `Name` and their address in the properties `Street` and `City`. As the project progresses, you decide that you want to copy the address part to another table and have the `User` entity class reference it via a navigational property. Figure 9.4 shows the before and after states of the database's schema and the content of the tables.

The best way to handle this situation with EF Core's migration is to add some SQL commands to copy over the data, but the process isn't trivial. Changing the migration requires adding SQL code.



Userid	Name	Street	City
123	Jack	Jack Street	Jack City
456	Jill	Jill Street	Jill City

Userid	Name	AddressId
123	Jack	1
456	Jill	2

AddressId	Street	City
1	Jack Street	Jack City
2	Jill Street	Jill City

Figure 9.4 The original (before) database schema and data, with one table called Users. The new (after) database schema has a new table, Addresses, and the address data in the original Users table has been moved to the Addresses table. Also, the Users table address columns, Street and City, have been removed, and a new foreign key, `AddressId`, has been added to link to the User's addresses.

NOTE You can see the whole migration in the associated GitHub repo at http://mng.bz/goME.

First, you change your `User` entity class to remove the address and link to the new `Address` entity class to the DbContext. Then you create a new migration by using the `add migration` command, which will warn you that it may result in the loss of data. At this point, you are ready to edit the migration.

The second step is adding a series of SQL commands, using the `MigrationBuilder` method `Sql`, such as `migrationBuilder.Sql("ALTER TABLE…")`. The following listing shows you the SQL commands without the `migrationBuilder.Sql` so that they are easier to see.

Listing 9.5 The SQL Server commands to copy over the addresses to a new table

```
ALTER TABLE [Addresses]
    ADD [UserId] [int] NOT NULL

INSERT INTO [Addresses] ([UserId],[Street],[City])
    SELECT [UserId],[Street],[City] FROM [Users]

UPDATE [Users] SET [AddressId] = (
    SELECT [AddressId]
        FROM [Addresses]
        WHERE [Addresses].[UserId] = [Users].[Userid])

ALTER TABLE [Addresses]
    DROP COLUMN [UserId]
```

Adds a temporary column to allow the correct foreign key to be set in the Users table

Copies over all the address data, with the User's primary key, to the addresses table

Uses the temporary UserId column to make sure that the right foreign keys are set up

Removes the temporary UserId column in the Addresses table, as it's not needed anymore

Sets the foreign key in the Users table back to the Addresses table

You add these SQL commands to the migration by using the `migrationBuilder.Sql` method for each SQL command, placing them after the Addresses table is created but before the foreign key is set up. Also, the `MigrationBuilder` methods that drop (remove) the address properties from the Users table must be moved to after the SQL code has run; otherwise, the data will have gone before your SQL can copy that data over.

NOTE Section 9.8.1 covers a way to run C# code before and after a specific migration has been applied to a database. That approach is another way to copy data, but the SQL approach often performs better.

9.5.3 *Adding your own custom migration commands*

If you often add certain types of SQL commands to a migration, you can build some templating code to make your edits easier to write. Building templates, such as adding an SQL View to the database, is a good idea if you use an SQL feature often, because the cost of creating the template is less effort than handcoding the SQL feature multiple times. You have two ways to create a template:

- Create extension methods that take the `MigrationBuilder` class in and build commands with `MigrationBuilder's Sql` method. These extension methods tend to be database-specific.
- A more complex but more versatile approach is to extend the `MigrationBuilder` class to add your own commands. This approach allows you to access methods to build commands that work for many database providers.

In this section, I discuss only the first approach. The second is an advanced version that is well described in the EF Core documentation at http://mng.bz/xGBe.

As an example, you are going to create an extension method that will allow you to create SQL Views more easily. The extension method takes in the class that will be mapped to the View so that it can find the properties to map to the columns (assuming that you are using only properties and By Convention column naming). The following listing shows the extension method that will create a view within a migration.

Listing 9.6 Extension method to add/alter an SQL view in an EF Core migration

An extension method must be in a static class.

The method needs the class that is mapped to the view so that it can get its properties.

The MigrationBuilder provides access to the migration methods—in this case, the Sql method.

```
public static class AddViewExtensions
{
    public static void AddViewViaSql<TView>(
        this MigrationBuilder migrationBuilder,
        string viewName,
        string tableName,
        string whereSql)
        where TView : class
    {
```

The method needs the name to use for the view and the name of the table it is selecting from.

Views have a Where clause that filters the results returned.

Ensures that the TView type is a class

```
        if (!migrationBuilder.IsSqlServer())
            throw new NotImplementedException(“warning…”)

        var selectNamesString = string.Join(", ",
            typeof(TView).GetProperties()
            .Select(x => x.Name));

        var viewSql =
            $"CREATE OR ALTER VIEW {viewName} AS " +
            $"SELECT {selectNamesString} FROM {tableName} " +
            $"WHERE {whereSql}";

        migrationBuilder.Sql(viewSql);
    }
}
```

This method throws an exception if the database isn’t Server because it uses an SQL Server view format.

Gets the names of the properties in the class mapped to the view and uses them as column names

Creates the SQL command to create/update a view

Uses MigrationBuilder’s method to apply the created SQL to the database

You would use this technique in a migration by adding it to the `Up` method (and a `DROP VIEW` command in the `Down` method to remove it). Here is a code snippet that creates a view for the `MyView` class, which has the properties `MyString` and `MyDateTime`:

```
migrationBuilder.AddViewViaSql<MyView>(
    "EntityFilterView", "Entities",
    "MyDateTime >= '2020-1-1'");
```

The resulting SQL looks like this snippet:

```
CREATE OR ALTER VIEW EntityFilterView AS
SELECT MyString, MyDateTime
FROM Entities
WHERE MyDateTime >= '2020-1-1'
```

9.5.4 Altering a migration to work for multiple database types

EF Core migrations are database-provider-specific—that is, if you build a migration for SQL Server, it almost certainly won’t work for a PostgreSQL database. You don’t often need migrations for multiple database types, however. In fact, I don’t recommend using multiple database types with the same EF Core code, as subtle differences between database types can catch you out (see chapter 16). But if you need to support migrations for two or more types of databases, the recommended way is to build separate migrations for each database provider. If you want to use an SQLite database for a Linux version of your application and an SQL Server database for a Windows version of your application, for example, you would need to execute the following steps.

The first step is creating a specific DbContext for each database type. The easiest way is to create a main application’s DbContext and inherit it in your other database types. The following listing shows two applications’ DbContexts, with the second one inheriting the first one.

Listing 9.7 Two DbContexts that have the same entity classes and configuration

```
public class MySqlServerDbContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    // … other DbSets left out

    protected override void OnModelCreating
        (ModelBuilder modelBuilder)
    {
        //… your Fluent API code goes here
    }
}

public class MySqliteDbContext : MySqlServerDbContext
{

}
```

Inherits the normal DbContext class

Adds all the DbSet properties and Fluent APIs, which are used in both database types

The MySqliteDbContext inherits the Sql Server DbContext class instead of the normal DbContext.

The MySqliteDbContext inherits the DbSet properties and Fluent APIs from the Sql Server DbContext.

The next step is creating a way for the migration tools to access each DbContext with the database provider defined. The cleanest way is to create an `IDesignTimeDbContextFactory<TContext>` class, as described in section 9.4.1. Alternatively, you can override the `OnConfiguring` method in each DbContext to define the database provider.

At this point, you can create a migration for each database type by using the `Add-Migration` command (see section 9.4.2). The important point is that each migration must be in a separate project so that when you create a migration, it can access the correct migration classes for the type of database to which the DbContext is linked. You tell EF Core where the migration classes can be found by using the `MigrationsAssembly` method when you create the database option. The following code snippet shows the `AddDbContext` method used to register an application's DbContext with its database provider and the migrations for that database in a project called `Database.SqlServer`:

```
services.AddDbContext<MySqlServerDbContext>(
    options => options.UseSqlServer(connection,
       x => x.MigrationsAssembly("Database.SqlServer")));
```

Alternatively, you have one migration and add `if/then` code inside the migration to change what the migration does based on the database provider. This approach isn't recommended because it's harder to maintain. If you want more information on this approach, I suggest looking at the EF Core documentation, which covers both approaches (http://mng.bz/pV08).

NOTE Cosmos DB and NoSQL databases in general don't use EF Core migrations because they don't have a fixed schema, like SQL databases, and they're normally migrated by means of some form of upgrade script. Migrating a

Cosmos DB database accessed via EF Core does have some issues, which I discuss in chapter 16.

9.6 Using SQL scripts to build migrations

The next way to manage your database schema change is to produce SQL *change scripts* and then apply them to any of your databases. Change scripts contain SQL commands that update the schema of your database. This approach to handling database schema updates is more traditional and gives you much better control of the database features and the schema update. You need good knowledge of SQL commands to write and understand these migration scripts, but tools can generate these migration scripts for you by comparing databases.

As with the migrations that EF Core can create, your aim is to create a migration that will alter the schema of your database to match the EF Core's internal model of the database. In this section, you will consider two approaches:

- Using SQL database comparison tools to produce migration from the current database schema to the desired database schema
- Handcoding a change script to migrate the database

Although option 1 should produce an exact match to EF Core's internal model of the database, option 2 relies on the developer to write the correct SQL to match what EF Core needs. If the developer makes a mistake (which I can testify is easy to do), your application may fail with an exception; worse, it may silently lose data. So at the end of this section, I describe a tool I have created that compares a database's schema with EF Core's current model of the database and tells you whether there are any differences.

9.6.1 Using SQL database comparison tools to produce migration

One approach to creating an SQL change script is to compare two databases: your original database and a new database created by EF Core after you've updated the EF Core configuration. Tools can compare two databases and show the differences in their schemas. Many of these comparison tools can also create a script that will change your original database to the same schema as the database you want to move to. So if you can create a database with the schema you want, a comparison tool can create the SQL change script needed to update a database to the required database. SQL comparison tools make creating SQL change scripts quite easy, but like everything, they have their own quirks. Before you look at the details, table 9.3 gives you an overview of this approach.

Several open source and commercial comparison tools are available for many database server types; they can compare database schemas and output SQL change scripts. This example uses the SQL Server Object Explorer built into Visual Studio (any version), which you can find in the Data Storage and Processing workload of the Visual Studio installer. You can get the tool directly by choosing Tools > SQL Server > New Schema Comparison.

Table 9.3 A summary of the good, the bad, and the limitations of using an SQL database comparison tool to build SQL change scripts to migrate a database

	Notes
Good parts	▪ Tools build the correct SQL migration script for you.
Bad parts	▪ You need some understanding of databases and SQL. ▪ SQL comparison tools often output every setting under the sun to make sure that they get everything right, which makes the SQL code output hard to understand. ▪ Not all SQL comparison tools produce a migration remove script.
Limitations	Tools do not handle breaking changes, so they need human input.
Tips	I use this approach only for complex/large migrations, and I strip out any extra settings to make the code easier to work with.
My verdict	This approach is useful and especially good for people who aren't comfortable with the SQL language. It's also useful for people who have written their own SQL migration code and want to check that their code is correct.

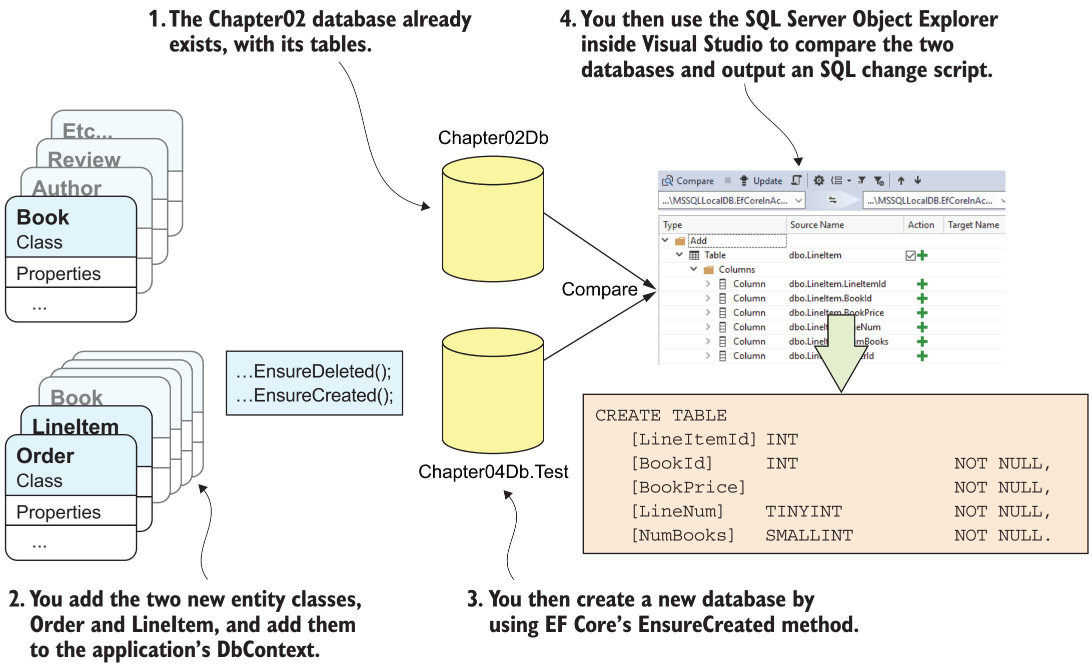
NOTE You can find a step-by-step guide to using SQL Server Object Explorer at http://mng.bz/OEDR.

Figure 9.5 shows how to compare the database in chapter 2 with the changes in chapter 4, where you add `Order` and `LineItem` entity classes. An SQL comparison tool relies on having two databases:

- The first database is the current state of the database, known as the Target database. You want to update to the new schema, which is shown as Chapter02Db in figure 9.5. This database is most likely your production database or some other database that matches the target schema.
- The second database, known as the Source database, must have a schema to which you want to update your database, shown as Chapter04Db.Test in figure 9.5. This database is most likely in your development environment. One nice feature I use to get a database like this one is EF Core's `EnsureCreated` method. This method, usually used in unit testing, creates a database based on the current entity classes and EF Core configuration.

By setting two databases as the Source and Target databases in the compare SQL schema tool, you can compare the two database schemas and then produce an SQL change script that transitions the schema of the Target database to the schema of the Source database.

This process initially tells you the differences; then you have an option to create an SQL change script that will migrate a database from the initial database schema to the required database schema. This option produces an SQL change script that will migrate a database from the initial database schema, Chapter02Db in figure 9.5, to the schema of the target database, Chapter04Db.Test in figure 9.5. I cover how to apply a change script in section 9.8.4.



1. The Chapter02 database already exists, with its tables.

4. You then use the SQL Server Object Explorer inside Visual Studio to compare the two databases and output an SQL change script.

2. You add the two new entity classes, Order and LineItem, and add them to the application's DbContext.

3. You then create a new database by using EF Core's EnsureCreated method.

```
CREATE TABLE
    [LineItemId] INT
    [BookId]     INT                  NOT NULL,
    [BookPrice]                       NOT NULL,
    [LineNum]    TINYINT              NOT NULL,
    [NumBooks]   SMALLINT             NOT NULL.
```

Figure 9.5 The process of building an SQL change script by comparing two databases. The important point is that the second database, Chapter04Db.Test, is created by EF Core, so you know that it matches the current EF Core model. In this example, you use the SQL Server Object Explorer feature of Visual Studio to compare the two databases and build an SQL change script that will migrate the Chapter02 database to the correct level for the software changes added in chapter 4.

9.6.2 *Handcoding SQL change scripts to migrate the database*

Another approach is to create the SQL commands needed for a migration yourself. This option is attractive to developers who want to define the database in ways that EF Core can't. You can use this approach to set more-rigorous `CHECK` constraints on columns, add stored procedures or user-defined functions, and so on via SQL scripts.

The only disadvantage for a software developer is that you need to know enough SQL to write and edit the SQL change scripts. This requirement might put off some developers, but it's not as bad as you think, because you can look at the SQL EF Core outputs to create a database and then tweak that SQL with your changes. Table 9.4 gives you an overview of this approach.

The job of creating an SQL change script is made easier by the migration `script-dbcontext` command, which outputs the SQL commands that EF Core would use to create a new database (equivalent to calling the `context.Database.EnsureCreated` method). The following listing shows a small part of the SQL produced by the `EnsureCreated` method, with the focus on the Review table and its indexes.

Table 9.4 A summary of the good, the bad, and the limitations of handcoding the SQL change scripts to migrate a database

	Notes
Good parts	- You have total control of the database structure, including parts that EF Core won't add, such as user-defined functions and column constraints.
Bad parts	- You must understand SQL commands such as `CREATE TABLE`. - You must work out what the changes are yourself (but see the Tip row). - There's no automatic migration remove script. - This approach is not guaranteed to produce a correct migration (but see CompareEfSql in section 9.6.3).
Limitations	None
Tips	You can use the `Script-DbContext` migration command to get the actual SQL that EF Core would output and then look for the differences in the SQL from the previous database schema, which makes writing the SQL migrations much easier.
My verdict	This approach is for someone who knows SQL and wants complete control of the database. It certainly makes you think about the best settings for your database, which can improve performance.

Listing 9.8 Part of SQL generated by `EnsureCreated` when creating a database

```
-- other tables left out
CREATE TABLE [Review] (
    [ReviewId] int NOT NULL IDENTITY,
    [VoterName] nvarchar(100) NULL,
    [NumStars] int NOT NULL,
    [Comment] nvarchar(max) NULL,
    [BookId] int NOT NULL,
    CONSTRAINT [PK_Review] PRIMARY KEY ([ReviewId]),
    CONSTRAINT [FK_Review_Books_BookId]
        FOREIGN KEY ([BookId])
        REFERENCES [Books] ([BookId]) ON DELETE CASCADE

-- other SQL indexes left out
CREATE INDEX [IX_Review_BookId] ON [Review] ([BookId]);
```

Creates the Review table, with all its columns and constraints

Says that the database will provide a unique value when a new row is created

Says that the ReviewId column is the primary key

Says that the BookId column is a foreign key link to the Books table and that if the Books row is deleted, the linked Review row will be deleted too

Says that there should be an index of the BookId foreign key to improve performance

Because you know what entity classes of EF Core configurations you have changed, you can find the appropriate part of the SQL that should reflect your changes. This information should help you write your SQL command, and you're more likely to write SQL change scripts that match what EF Core expects.

As with EF Core's migrations, you create a series of SQL change scripts that need to be applied to your database in order. To aid this process, you should name your scripts with something that defines the order, such as a number or a sortable date. Here are example SQL script names that I used for a client project:

```
Script001 - Create DatabaseRegions.sql
Script002 - Create Tenant table.sql
Script003 - TenantAddress table.sql
Script004 - AccountingCalenders table.sql
```

As well as being applied to the database in order, script names should be applied only once; I cover how in section 9.8.

Should I be writing a remove migrations for my SQL change scripts?

EF Core's migrations creates both the `Up` migration method and a `Down` migration method. The `Down` method, which is known as a *reverting migration*, contains code to undo the `Up` migration. Some developers who move to SQL change scripts worry about not having a remove migration feature.

The fact is that although it's great to have a way to remove a migration, you aren't likely to use it much. EF Core can automatically produce a migration that reverts its migration, but when it comes to SQL change scripts, building a `Down` script isn't automatic, so if you want a `Down` script, you have to write that SQL.

Therefore, I create a remove migration only if I need it, so any remove migration is another new SQL change script that reverts the last migration. But be warned: I do this only as a result of intensive testing of my migrations well before production, because having to write a `Down` migration script because your production system is down due to a bad migration is a bit stressful!

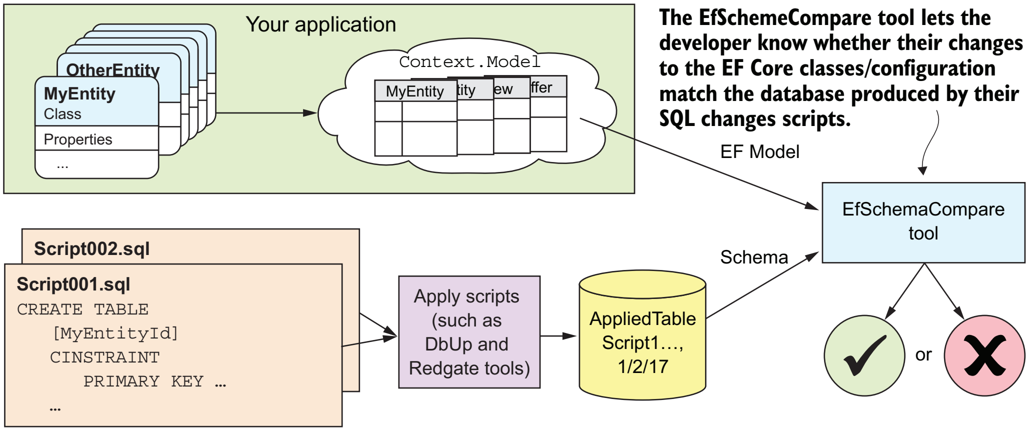
9.6.3 Checking that your SQL change scripts matches EF Core's database model

I have used handcoded SQL change scripts in several projects, both in EF6 and EF Core, and the main concern is making sure that my modifications to the database match EF Core's model of the database. So I created a tool, which I refer to as EfSchemaCompare, that compares EF Core's model of the database with the schema of an actual database. Although EfSchemaCompare has some limitations, it provides good feedback on differences between a migrated database and EF Core's model of the database.

NOTE I cover EF Core's model of the database, accessed by the `Model` property in your application's DbContext, in chapter 11.

Figure 9.6 shows how the EfSchemaCompare tool compares a database that has been updated by your SQL change scripts against EF Core's database model.

The EfSchemaCompare tool is available in my EfCore.SchemaCompare library (see http://mng.bz/Yq2B). With this tool, I create unit tests that check my development



The EfSchemeCompare tool lets the developer know whether their changes to the EF Core classes/configuration match the database produced by their SQL changes scripts.

Figure 9.6 The EfSchemaCompare tool compares EF Core's model of the database, which it forms by looking at the entity classes and the application's DbContext configuration, with the database schema of a database that has been updated via your SQL change scripts. The tool outputs human-readable error messages if it finds a difference.

database—and, more important, my production database—to see whether the EF Core's database model has drifted away from the actual database schema.

9.7 Using EF Core's reverse-engineering tool

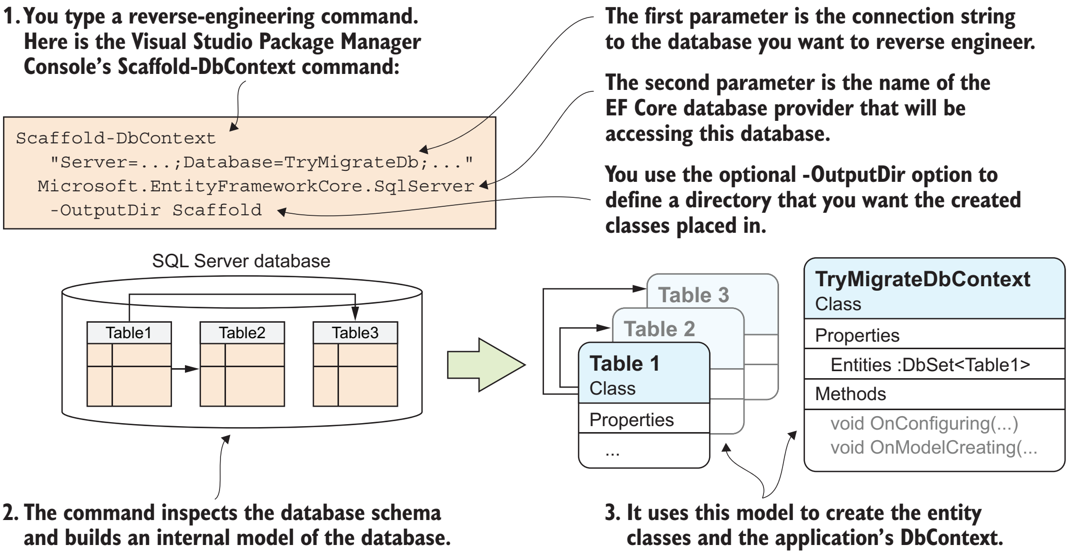
In some cases, you already have a database that you want to access via EF Core code. For this purpose, you need to apply the opposite of migrations and allow EF Core to produce your entity classes and application's DbContext by using your existing database as the template. This process is known as reverse engineering a database. This approach says that the database is the source of truth. You use EF Core's reverse-engineering tool, also known as scaffolding, to re-create the entity classes and the application's DbContext with all the required configurations. Table 9.5 gives you an overview of this approach, and figure 9.7 shows the process.

Table 9.5 A summary of the good, the bad, and the limitations of reverse-engineering a database as a way to access an existing database or continually update your entity classes and application DbContext to match a changed database

	Notes
Good parts	▪ The tool builds the EF Core code/classes from an existing database. ▪ The tool allows you to make the database the source of truth, and your EF Core code and classes are created and updated as the database schema changes.
Bad parts	▪ Your entity classes can't be edited easily, such as to change the way that the collections navigational properties are implemented. But see section 9.7.2 for a solution to this issue. ▪ The tool always adds navigational links at both ends of the relationship (see section 8.2).

Table 9.5 A summary of the good, the bad, and the limitations of reverse-engineering a database as a way to access an existing database or continually update your entity classes and application DbContext to match a changed database *(continued)*

	Notes
Limitations	None
Tips	When you are going to repeatedly reverse engineer a database, I recommend using the Visual Studio EF Core Power Tools extension, as it remembers the setting from the last time you used the reverse-engineering feature.
My verdict	If you have an existing database that you need to access via EF Core, reverse engineering is going to save you a lot of time.



1. You type a reverse-engineering command. Here is the Visual Studio Package Manager Console's Scaffold-DbContext command:

```
Scaffold-DbContext
   "Server=...;Database=TryMigrateDb;..."
  Microsoft.EntityFrameworkCore.SqlServer
   -OutputDir Scaffold
```

The first parameter is the connection string to the database you want to reverse engineer.

The second parameter is the name of the EF Core database provider that will be accessing this database.

You use the optional -OutputDir option to define a directory that you want the created classes placed in.

2. The command inspects the database schema and builds an internal model of the database.

3. It uses this model to create the entity classes and the application's DbContext.

Figure 9.7 Typical use of EF Core's reverse-engineering command, which inspects the database found via the database connection string and then generates the entity classes and the application's DbContext to match the database. The command uses the foreign-key database relationships to build a fully defined relationship between the entity classes

You use this approach mainly when you want to build an EF Core application around an existing database, but I also describe a way to manage migrations. To start, let's look at how to run the reverse-engineering tool. You have two options:

- Run EF Core's reverse-engineering tool via a command line.
- Use the EF Core Power Tools Visual Studio extension.

9.7.1 Running EF Core's reverse-engineering command

You can reverse engineer a database from a command line (CLI tools) or Visual Studio's PMC window. CLI and PMC have different names and parameters. The following list shows the `scaffold` command to reverse engineer the BookApp database. Note that commands are run in the directory of the BookApp ASP.NET Core project and that the database connection string is in the appsettings.json file in that project:

- *CLI*—`dotnet ef dbcontext scaffold name=DefaultConnection Microsoft .EntityFrameworkCore.SqlServer`
- *PMC*—`Scaffold-DbContext -Connection name=DefaultConnection -Provider Microsoft.EntityFrameworkCore.SqlServer`

NOTE There are lots of commands, with multiple parameters, and it would take many pages to reproduce the EF Core's documentation. Therefore, I direct you to EF Core's command-line reference at http://mng.bz/MXEn.

9.7.2 Installing and running EF Core Power Tools reverse-engineering command

The EF Core Power Tools Visual Studio extension was created and maintained by Erik Ejlskov Jensen, known as @ErikEJ in GitHub and on Twitter. This tool uses EF Core's reverse-engineering service but provides a visual frontend to make it easier to use. This extension is helpful because the reverse-engineering code often needs lots of parameters, including long connection strings. Erik's tool also adds some features, such as the ability to customize the templates that produce the code.

First, you need to install the EF Core Power Tools Visual Studio extension. You can find the EF Core Power Tools at http://mng.bz/Gx0v. If you aren't familiar with installing Visual Studio extensions, see http://mng.bz/zxBB.

After you have installed the extension, right-click a project in Visual Studio's Solution Explorer. You should see a command called EF Core Power Tools, with a Reverse Engineering subcommand. Please read the EF Core Power Tools wiki in its GitHub repo (https://github.com/ErikEJ/EFCorePowerTools/wiki).

9.7.3 Updating your entity classes and DbContext when the database changes

One way to handle database changes is to migrate your database and then run the reverse-engineering tool to re-create your entity classes and application's DbContext. That way, you know that the database schema and EF Core's model are in step.

Using EF Core's reverse-engineering tool directly works, but you must remember all the settings for each run. The EF Core project has a feature on backlog (issue #831) that would try to preserve the current class and alter only the properties and relationships that changed. That sort of feature would be great, but it would be complex to implement, so it hasn't been considered for a while. Fortunately, EF Core Power Tools extension is a decent substitute.

EF Core Power Tools has been designed to make updating the entity classes and application's DbContext easy, with nice features such as remembering your last run by adding a file to your project. I talked to Erik Ejlskov; he said he uses an SQL Server database project (.sqlproj) to keep the SQL Server schema under source control, and the resulting SQL Server .dacpac files to update the database and EF Core Power Tools to update the code.

For me, the downside of reverse engineering to handle migrations is that I can't easily change the design entity classes, such as to follow a DDD style (see chapter 13). But you could use reverse engineering once to get the entity classes and DbContext and then swap over to use the EF Core code as the source of truth. Then you can edit the entity classes to the style you want, but after that, you would need to swap migrating your database via EF Core's migrations or SQL change scripts.

> **NOTE** Some of my experiments with reverse engineering show that EF Core's `OnDelete` configuration settings aren't quite what I expect; see EF Core issue #21252. When I asked Erik about this situation, his answer was along these lines: the database `ON DELETE` setting is right, and that's the important part.

9.8 Part 2: Applying your migrations to a database

Up to this point, you have been considering different ways to migrate a database. In this section, you consider how to apply your migration to a database. The way that you create a migration influences how you can apply it. If you created your migrations by using SQL change scripts, for example, you can't apply them by using EF Core's `Migrate` method. Here is a list of the techniques you will be evaluating in the rest of this chapter:

- Calling EF Core's `Database.Migrate` method from your main application
- Executing EF Core's `Database.Migrate` method from a standalone application designed only to migrate the database
- Applying an EF Core migration via an SQL change script and applying it to a database
- Applying SQL change scripts by using a migration tool

The other issue that affects how you migrate your database is the environment you are working in—specifically, the characteristics of the application that accesses the database being migrated, with special focus on your production system. The first characteristic is whether you are running multiple instances of the application, such as multiple instances of an ASP.NET Core, which is known as scaled-out in Microsoft Azure. This characteristic is important because all the ways of applying a migration to a database rely on only one application's trying to change the database's schema. Having multiple instances running, therefore, rules out some of the simpler migration update techniques, such as running a migration when the application starts because all the multiple instances will try to run at the same time (but see @zejji's solution to this limitation in a note in section 9.8.1).

The second characteristic is whether the migration is applied while the current application is running. This situation happens if you have applications that need to be up all the time, such as email systems and sites that people want to access at any time, such as GitHub and Amazon. I refer to these types of applications as continuous-service applications.

Every migration applied to a database of a continuous-service application must not be an application-breaking change; the migrated database must still work with the currently running application code. If you add a non-nullable column with no default SQL value, for example, when the old application creates a new row, the database will reject it, as the old application didn't provide a value to fill in the new column. This application-breaking change must be split into a series of nonbreaking changes, as covered in section 9.9.2.

The following sections look at four ways to apply a migration to your database, with references to the application's characteristics. Some of the most complex issues related to database schema changes are covered in section 9.9.

9.8.1 Calling EF Core's Database.Migrate method from your main application

You saw this approach for ASP.NET Core in section 5.9.2, but to recap, you add some code that calls `context.Database.Migrate` before the main application starts. This approach is by far the easiest way to apply a migration, but it has a big limitation: you should not run multiple instances of the `Migrate` method at the same time. If your application has multiple instances running at the same time—the many app characteristic—you cannot use this approach. Table 9.6 provides an overview of this approach.

Table 9.6 A summary of the good, the bad, and the limitations of calling EF Core's `Database.Migrate` method from your main application

	Notes
Good parts	▪ This approach is relatively easy to implement. ▪ It ensures that the database is up to date before your application runs.
Bad parts	▪ You must *not* run two or more `Migrate` methods in parallel. ▪ There is a small period when your application isn't responding; see the note after this table. ▪ If the migration has an error, your application won't be available. ▪ It can be hard to diagnose startup errors.
Limitations	This approach does not work if multiple instances of the application are running (but see @zejji's solution to this limitation after this note).
Tips	For ASP.NET Core applications, I still recommend applying the migration in your CI/CD pipeline, even if you expect to run only one instance of the web app (see section 9.8.2), because your app won't be deployed if the migration fails, and you will be ready to scale out if you need to.
My verdict	If you can guarantee that only one instance of your application is starting up at any one time, this approach is a simple solution to migrating your database. Unfortunately, that situation isn't typical for websites and local applications.

NOTE This approach assumes that you are deploying your application without using any continuously running features, such as Azure Web App slots and swapping. In that case, the old application will be stopped before the new application starts. During that (brief) time, any accesses to the application will fail, possibly losing the data that they were editing.

The BookApp in the associated GitHub repo uses this approach, which means that you can run the application on your development machine, and it will create the database for you automatically (if you have localdb installed), which shows how useful it is. But for applications that you need to scale out, this approach isn't going to work.

NOTE The GitHub user @zejji posted an approach that ensures that the `Migrate` method is called only once in an application that has multiple instances running at the same time. This approach overcomes one of the problems of calling the `Migrate` method on startup; see http://mng.bz/VGw0.

Finding what migrations the Database.Migrate method will apply to the database

When you use the `context.Database.Migrate` method to migrate a database, you may want to run some C# code if a certain migration is applied. I used this technique to fill in a new property/column added in a certain migration. You can find out what migrations are going to be applied to the database by calling the `GetPendingMigrations` method before you call the `Migrate` method and the method called `GetAppliedMigrations` to get the migrations that have been applied to the database.

Both methods return a set of strings of the filenames that hold the migration. BookApp, for example, has a migration class called `InitialMigration`, which is in a file called something like 20200507081623_InitialMigration. The following listing shows how you might detect that the `InitialMigration` had been applied so that you can run your C# code on the migrated database.

Listing 9.9 Detecting what migrations have been applied to the database

```
context.Database.Migrate();
if (context.CheckIfMigrationWasApplied(nameof(InitialMigration)))
{
    //... run your C# code for this specific migration
}

//Extension method to detect a specific migration was applied
public static bool CheckIfMigrationWasApplied(
    this DbContext context, string className)
{
    return context.Database.GetAppliedMigrations()
        .Any(x => x.EndsWith(className));
}
```

You call the migration method to apply any missing migrations to the database.

You use the extension method to find whether the InitialMigration was added to the database.

Code that must run after the InitialMigration has run

Simple extension method to detect a specific migration from the class name

The GetAppliedMigrations method returns a filename for each migration applied to the database.

All the filenames end with the class name, so we return true if any filename ends with className.

I have used this approach to good effect, but be warned that if your C# code takes too long in a ASP.NET Core application, your web server may time out the application, in which case your extra C# migration update code would be stopped in the middle of its work.

9.8.2 Executing EF Core's Database.Migrate method from a standalone application

Instead of running the migration as part of your startup code, you can create a standalone application to apply a migration to your databases. You could add a console application project to your solution, for example, using your application's DbContext to call the `context.Database.Migrate` method when it's run, possibly taking the database connection string as a parameter. Another option is calling the CLI command `dotnet ef database update`, which in EF Core 5 can take a connection string. This approach can be applied when the application is running or when it is stopped. This section assumes that the application is stopped. In section 9.9, I cover the approach to use while the application is running. Table 9.7 gives you an overview of this approach.

Table 9.7 A summary of the good, the bad, and the limitations of executing EF Core's `Database.Migrate` method from a standalone application

	Notes
Good parts	■ If the migration fails, you get good feedback from the migration. ■ This approach overcomes the problem that the `Migrate` method isn't thread safe.
Bad parts	■ Your application is down while the migration is applied. (But see section 9.9 for an example of migrating a database while the application is running.)
Limitations	None
My verdict	This option is a good one if you have multiple instances of your application. In your CI/CD pipeline, for example, you could stop the current applications, run one of EF Core's `Migrate` commands (such as `dotnet ef database update`), and then upload and start your new application.

If no applications are accessing the database, perhaps because they have all stopped, there are no issues involved in applying your migration to the database. This approach is what I call a *down for maintenance migration*; see figure 9.8 for details.

9.8.3 Applying an EF Core's migration via an SQL change script

In some cases, you want to use EF Core's migrations, but you want to check the migrations or apply them via SQL change scripts. You can get EF Core to create SQL change scripts, but watch out for a few things if you take this approach. The default SQL change script produced by EF Core, for example, contains only the script to update the database, with no check of whether a migration has already been applied. The reason is that developers normally apply SQL change scripts via some sort of deployment system that handles the job of working out what migrations need to be applied to the database being migrated. Table 9.8 gives you an overview of this approach.

> **NOTE** There is also a way to output a script that checks whether the migration has been applied, which is covered at the end of this section.

Table 9.8 A summary of the good, the bad, and the limitations of applying an EF Core's migration via an SQL change scripts

	Notes
Good parts	▪ EF Core will build your migrations for you and then give you the migration as SQL. ▪ The SQL scripts generated by EF Core update the migration history table.
Bad parts	▪ You need an application to apply the migrations to your databases.
Limitations	None
Tips	▪ Be aware that the individual migrations don't check whether the migration has been applied to the database. This approach assumes that some other application is keeping track of the migrations. ▪ If you need a migration that checks whether it has already been applied to the database, you need to add the `idempotent` parameter to the command.
My verdict	If you want to check/sign off a migration or use a more comprehensive app/database deployment system, such as Octopus Deploy or a RedGate product, this approach is the way to go.

The basic command to turn the latest migration into an SQL script is

- *CLI*—`dotnet ef migrations script`
- *PMC*—`Script-Migration`

These two commands output the SQL for the last migration with no check of whether that migration has been applied to the database. But when you add the `idempotent` parameter to these commands, the SQL code that they produce contains checks of the migration history table and applies only migrations that haven't been applied to the database.

> **NOTE** There are lots of commands, with multiple parameters, and it would take many pages to reproduce the EF Core's documentation. Therefore, I direct you to EF Core's command-line reference at http://mng.bz/MXEn.

Since the EF Core 5 release, the SQL script created by the `Script-Migration` command has applied a migration within an SQL transaction. The whole of the migration will be applied to the database unless there is an error, in which case none of the migration will be applied.

> **WARNING** SQLite has some limitations on applying a migration in one transaction because some of the migration commands use transactions themselves, which means that a migration that fails may have applied part of the changes.

9.8.4 Applying SQL change scripts by using a migration tool

If you have gone for the SQL-change-scripts approach, it's likely that you already know how you will apply these change scripts to the database. You will need to use a migration tool such as DbUp (open source) or free or commercial tools such as RedGate's flyaway. Typically, these migration tools have their own version of EF Core migration history table. (DbUp calls this table SchemaVersions.)

How you implement the migration depends on the migration tool you use. DbUp, for example, is a NuGet package, so you can use it the same way as EF Core's `Migrate` method: call it on startup or as a separate application in your CI/CD pipeline, and so on. Other migration tools may not be callable from NET Core but use some form of command line or deployment pipeline integration. Table 9.9 gives you an overview of this approach.

Table 9.9 A summary of the good, the bad, and the limitations of applying SQL change scripts by using a migration tool

	Notes
Good parts	■ The tool works in all situations. ■ It works well with deployment systems.
Bad parts	■ You must manage the scripts yourself and make sure that their names define the order in which they will be applied.
Limitations	None
Tips	When I used this approach, I did a unit test to see whether a migrated test database matched EF Core's internal model by using my EfSchemaCompare tool (see section 9.6.3).
My verdict	I used SQL change scripts and DbUp in a couple of client projects, and they worked well. With some of the improvements in EF Core, I might be tempted back to using EF Core migrations.

9.9 Migrating a database while the application is running

Section 9.8 started a definition of two characteristics of the application that is accessing the database, and one of them was whether an application always needs to be available (a continuous-service application). Migrating a database while the application is running requires some extra work, which is covered in this section.

To start, let's compare the two types of applications: one that can be stopped for a migration or software update and one that must continue to provide a service while it's being updated (figure 9.8).

The rest of this section discusses how to migrate a database on a continuous-service application. There are two situations:

- The migration doesn't contain any changes that would cause the currently running application (referred to as the original app) to fail.
- The migration contains changes that would cause the original app to fail (application-breaking changes).

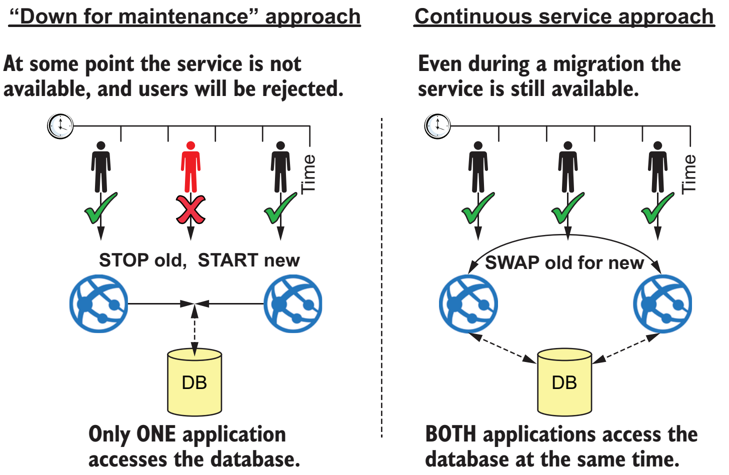


Figure 9.8 The example on the left is what happens if you replace the old application with a new application—in this case, also including a database migration. In this scenario, there is a period of time, known as *down time*, when neither the old or the new application is running, so there is a possibility of a user’s request being lost or rejected. The example on the right has an existing application providing a service, and a new version of the application is run up, ready to take over. When the new application starts, it applies a migration to the database. When the new application is ready, a “swap” occurs, and it seamlessly takes over the service.

Things to consider when stopping an application for a database update

You need to consider what will happen if you stop an application abruptly. That event could cause users to lose irretrievable data, or a user of an e-commerce site could lose their order. For this reason, you should consider a warning or soft stop.

I had this problem on an e-commerce system I built some years ago and developed a “down for maintenance” approach. This approach provided an onscreen warning to users, indicating that the site would close in a certain number of minutes. During the closing, I showed a “This site is down for maintenance” page and stopped users from accessing any pages. You can read about this project at http://mng.bz/mXkN, but be warned: I built it in 2016 with ASP.NET MVC.

Another way to softly stop your application is to provide read-only access to the database. You disable every method that could update the database. The application is still reading the database, so you can’t change the existing database structures, but you can add new tables and safely copy data into them. After you’ve loaded the new application, you can apply another database schema update to remove the database parts that are no longer needed.

9.9.1 Handling a migration that doesn't contain an application-breaking change

When I'm working on a new application with a new database, I tend to grow the database schema as the project progresses, perhaps by adding new tables that the previous versions of the software don't know about. These types of additions normally don't create migration, which breaks the application that is running in production. With a little bit of extra work, you can often make migrations that can be easily applied to a continuous-service application. Here are some issues to consider:

- If you're adding a new scalar property to an existing table, the old application won't set it. That's OK, because SQL will give it a default value. But what default do you want the property to have? You can control that setting by setting an SQL default value for the column (see chapter 10) or make it nullable. That way, the existing application running in production won't fail if you create a new row.
- If you're adding a new foreign-key column to an existing table, you need to make that foreign key nullable and have the correct cascade-delete settings. That approach allows the old application to add a new row to that table without the foreign-key constraint's reporting an error.

TIP Testing a (supposedly) nonbreaking database change that alters columns in existing tables is highly recommended, especially if you're going to a production database.

Some of these issues, such as making a column nullable when it would normally be non-nullable, might require a second migration to change the nullability of the database columns when your new application is in place. This situation leads to the multiple-step migration approach for dealing with application breaking changes.

9.9.2 Handling application-breaking changes when you can't stop the app

Applying an application breaking migration to a continuous-service application is one of the most complicated migrations there is. In fact, a couple of developers I talked to who work on continuous-service applications try hard to avoid an application-breaking migration. As I said in section 9.9.1, nonbreaking changes are the norm, so you might consider a "down for maintenance" approach for the (rare?) application-breaking changes. But if you really do need to apply an application-breaking change to your continuous-service application, read on.

As an example, you are going to consider to handle a database migration that moves columns from an Users table to a new Addresses table. In the original migration in section 9.5.2, this "move columns" issue was done by one migration, but it worked only because the original application was stopped, and after the migration finished, the new application ran.

For a continuous-service application, the move-columns task must be broken into a series of stages so that each migration doesn't break the two applications that are running at the same time. As a result, we end up with three migrations:

- `ADD`—The first migration is applied while App1 is currently running and adds new database features that the new interim application (App2) needs to run.
- `COPY`—The second migration is applied after App1 has stopped and before App3, the target application, has started. This migration copies the data in its final format.
- `SUBTRACT`—The last migration is a clean-up, which runs only when App2 has stopped and App3 has taken over. At this point, it can remove the old tables and columns that are now redundant.

The `ADD` and then `SUBTRACT` migrations, with maybe a `COPY` in the middle, represent the common approach to applying breaking changes to continuous-service applications. At no time should the database be incorrect for two applications that are running. In this example, you have five stages, as shown in figure 9.9.

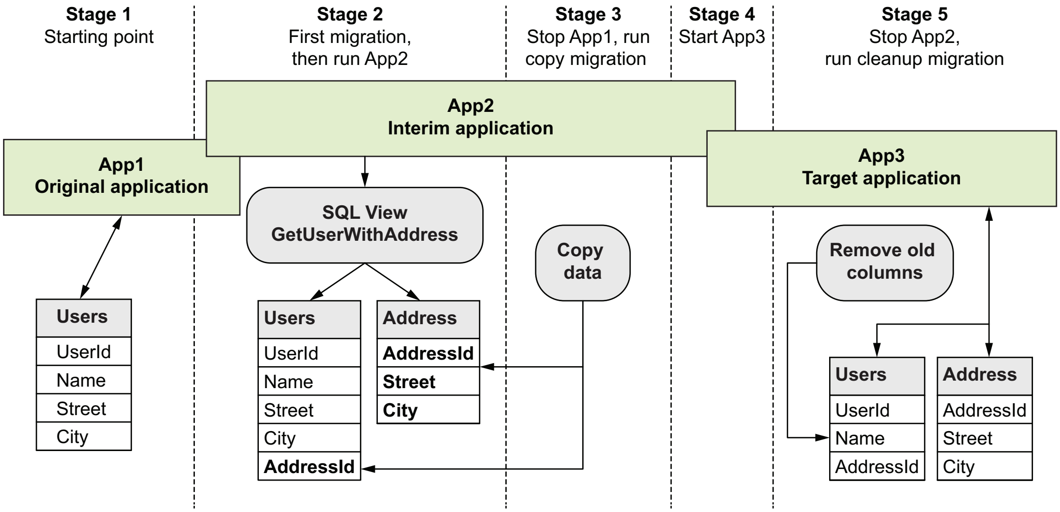


Figure 9.9 The five stages of turning an application-breaking migration so that the database isn't out of step with the two applications that are running at the time. The first migration changes the database so that App2 can work with App1; the next changes the database so that App3 can work with App2; and the final migration cleans up the database.

Here is a detailed breakdown of these stages:

- *Stage 1*—This stage is the starting point, with the original application, App1, running.

- *Stage 2*—This stage is the most complex one. It does the following:
  a. Runs a migration that creates a new Addresses table and links it to the current user.
  b. Adds an SQL View that returns a User with their address from either the old Users' Street/City columns or from the new Address table.
  c. The interim application, App2, uses the SQL View to read the User, but if it needs to add or update a User's address, it will use the new Address table.
- *Stage 3*—App1 is stopped, so there is no possibility that new addresses will be added to the Users table. At this point, the second migration runs and copies any address data in the Users table to the new Addresses table.
- *Stage 4*—At this point, the target application, App3, can be run; it gets a User's address only from the new Addresses table.
- *Stage 5*—App2 is stopped, so nothing is accessing the address part of the old User's table. This stage is when the last migration runs, cleaning up the database by removing the Street and City columns from the Users table, deleting the SQL View needed by App2, and fixing the User/Address relationship as required.

I could list all the code and migrations for this example, but to save space, I emulated this multistage migration in an unit test called `Ch09_FiveStepsMigration`, which you can find at http://mng.bz/0m2N. That way, you can see and run the whole process.

Summary

- The easiest way to create a migration is via EF Core's migration feature, but if you have a migration that removes or moves columns, you need to hand-edit before the migration will work.
- You can build SQL change scripts by using a database comparison tool or by hand. This approach gives you complete control of the database. But you need to check that your SQL change scripts create a database that matches EF Core's internal model of the database.
- If you have an existing database, you can use EF Core's `scaffold` command or the more visual EF Core Power Tools Visual Studio extension to create the entity classes and the application's DbContext with all its configurations.
- Updating a production database is a serious undertaking, especially if data could be lost in the process. How you apply migration to a production system depends on the type of migration and certain characteristics of your application.
- There are several ways to apply a migration to a database. The simplest approach has significant limitations, but the complex approaches can handle all migration requirements.
- Applying migration to a database while the application is running requires extra work, especially if the migration changes the database schema to the point that the current application will fail.

For readers who are familiar with EF6:

- EF Core's migration feature is significantly changed and improved, but anyone who has done EF6 migrations shouldn't have a problem swapping to EF Core's migration system.
- There's no automatic migration in EF Core; you control when a migration happens.
- It's easier to combine EF Core's migrations in a multiperson team.

10

Configuring advanced features and handling concurrency conflicts

This chapter covers

- Using an SQL user-defined function in EF Core queries
- Configuring columns to have default values or computed values
- Configuring SQL column properties on databases not created by EF Core
- Handling concurrency conflicts

This chapter discusses several advanced configuration features that interact directly with your SQL database, such as using SQL *user-defined functions* (UDFs) and computed columns. These features allow you to move some of your calculations or settings into the SQL database. Although you won't use these features every day, they can be useful in specific circumstances.

The second half of this chapter is about handling multiple, near-simultaneous updates of the same piece of data in the database; these updates can cause problems known as *concurrency conflicts*. You'll learn how to configure one property/column or a whole entity/table to catch concurrency conflicts, as well as how to capture and then write code to correct the concurrency conflict.

10.1 DbFunction: Using user-defined functions (UDFs) with EF Core

SQL has a feature called UDFs that allows you to write SQL code that will be run in the database server. UDFs are useful because you can move a calculation from your software into the database, which can be more efficient because it can access the database directly. UDFs can return a single result, which is referred to as *scalar-valued function*, and one that can return multiple data in a result, known as a *table-valued function*. EF Core supports both types of UDFs.

> **DEFINITION** An SQL *user-defined function* (UDF) is a routine that accepts parameters, performs an SQL action (such as a complex calculation), and returns the result of that action as a value. The return value can be a scalar (single) value or a table. UDFs differ from *SQL stored procedures* (StoredProc) in that UDFs can only query a database, whereas a StoredProc can change the database.

UDFs are useful, especially when you want to improve the performance of an EF Core query. I found some SQL (see https://stackoverflow.com/a/194887/1434764) that is quicker than EF Core at creating a comma-delimited string of authors' names. So instead of having to convert the whole of the Book App's book list query to SQL, I could replace only the part that returns the authors' names as a comma-delimited string. The steps for using a UDF in EF Core are as follows:

Configuration:

1. Define a method that has the correct name, input parameters, and output type that matches the definition of your UDF. This method acts as a reference to your UDF.
2. Declare the method in the application's DbContext or (optionally) in a separate class if it's a scalar UDF.
3. Add the EF Core configuration commands to map your static UDF reference method to a call to your UDF code in the database.

Database setup:

4. Manually add your UDF code to the database by using some form of SQL command.

Use:

5. Now you can use the static UDF reference in a query. EF Core will convert that method to a call to your UDF code in the database.

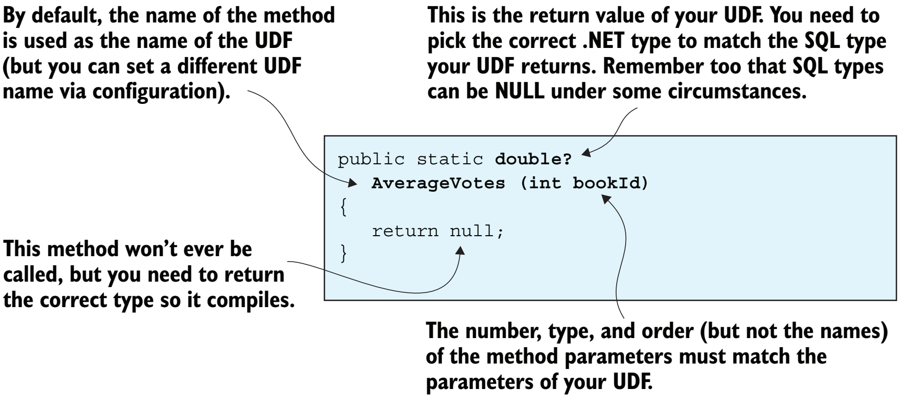
With that process in mind, let's detail the three stages: configuration, database setup, and use.

> **NOTE** The configuration and database-setup stages can be applied in any order, but both must be applied before you can use your UDF in a query.

10.1.1 Configuring a scalar-valued UDF

The configuration for a scalar-valued UDF consists of defining a method to represent your UDF and then registering that method with EF Core at configuration time. For this example, you're going to produce a UDF called `AverageVotes` that works out the average review votes for a book. `AverageVotes` takes in the primary key of the book you want to calculate for and returns a nullable double value—`null` if no reviews exist or the average value of the review votes if there are reviews.

You can define the UDF representation as a static or nonstatic method. Nonstatic definitions need to be defined in your application's DBContext; the static version can be placed in a separate class. I tend to use static definitions because I don't want to clutter the application's DBContext class with extra code. Figure 10.1 shows the static method that will represent the `AverageVotes` UDF in your software, with rules for forming this method.



```
public static double?
    AverageVotes (int bookId)
{
    return null;
}
```

By default, the name of the method is used as the name of the UDF (but you can set a different UDF name via configuration).

This is the return value of your UDF. You need to pick the correct .NET type to match the SQL type your UDF returns. Remember too that SQL types can be NULL under some circumstances.

This method won't ever be called, but you need to return the correct type so it compiles.

The number, type, and order (but not the names) of the method parameters must match the parameters of your UDF.

Figure 10.1 An example static method that will represent your UDF inside your EF Core code. The callouts highlight the parts that EF Core will use to map any calls to your UDF code and the rules that you need to follow when building your own method to map to your UDF.

> **NOTE** The UDF representation method is used to define the signature of the UDF in the database: it will never be called as a NET method.

You can register your static UDF representation method with EF Core by using either of the following:

- `DbFunction` attribute
- Fluent API

You can use the `DbFunction` attribute if you place the method representing the UDF inside your application's DbContext. In the example shown in the following listing, the `DbFunction` attribute and the static method are in bold.

Listing 10.1 Using a `DbFunction` attribute with a static method inside DbContext

```
public class Chapter08EfCoreContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    //… other code removed for clarity

    public Chapter08EfCoreContext(
        DbContextOptions<Chapter08EfCoreContext> options)
        : base(options) {}

    [DbFunction]
    public static double? AverageVotes(int id)
    {
        return null;
    }

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
       //… no Fluent API needed
    }
}
```

The DbFunction attribute defines the method as being a representation of your UDF.

The return value, the method name, and the number, type, and order of the method parameters must match your UDF code.

The method is never called, but you need the right type for the code to compile.

If you use the DbFunction attribute, you don't need any Fluent API to register the static method.

The other approach is to use the Fluent API to register the method as a UDF representation. The advantage of this approach is that you can place the method in any class, which makes sense if you have a lot of UDFs. This listing shows the Fluent API approach for the same method, `AverageVotes`, but it's defined in a class called `MyUdfMethods`, as shown in figure 10.1.

Listing 10.2 Registering your static method representing your UDF using Fluent API

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    //… other configuration removed for clarity

    modelBuilder.HasDbFunction(
        () => MyUdfMethods.AverageVotes(default(int)))
        .HasSchema("dbo");
}
```

Fluent API is placed inside the OnModelCreating method inside your application's DbContext.

HasDbFunction will register your method as the way to access your UDF.

Adds a call to your static method representation of your UDF code

You can add options. Here, you add HasSchema (not needed in this case); other options include HasName.

After you've used either of these configuration approaches, EF Core knows how to access your UDF in a query.

10.1.2 Configuring a table-valued UDF

EF Core 5 has added support for table-valued UDFs, which allow you to return multiple values in the same way that querying a table returns multiple values. The difference from querying a normal table is that the table-valued UDF can execute SQL code inside the database, using the parameters you provide to the UDF.

The table UDF example returns three values: the `Book`'s `Title`, the number of `Reviews`, and the average `Review Votes` for the `Book`. This example needs a class to be defined that will accept the three values coming back from the table-valued UDF, as shown in the following code snippet:

```
public class TableFunctionOutput
{
    public string Title { get; set; }
    public int ReviewsCount { get; set; }
    public double? AverageVotes { get; set; }
}
```

Unlike a scalar UDF, a table UDF can be defined in only one way—within your application's DbContext—because it needs access to a method inside the `DbContext` class called `FromExpression` (called `CreateQuery` before EF Core 5). What you are doing is defining the name and signature of the table-valued UDF: the name, the return type, and the parameters' type all must match your UTF. The following listing shows how you define the signature of your table UDF.

Listing 10.3 Defining a table-valued UDF within your application's DbContext

```
public class Chapter10EfCoreContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    //… other code removed for clarity

    public Chapter10EfCoreContext(
        DbContextOptions<Chapter10EfCoreContext> options)
        : base(options) {}

    public IQueryable<TableFunctionOutput>
        GetBookTitleAndReviewsFiltered(int minReviews)
    {
        return FromExpression(() =>
            GetBookTitleAndReviewsFiltered(minReviews));
    }

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {

    modelBuilder.Entity<TableFunctionOutput>()
            .HasNoKey();
```

The FromExpression will provide the IQueryable result.

The return value, the method name, and the parameters type must match your UDF code.

You place the signature of the method within the FromExpression parameter.

You must configure the TableFunctionOutput class as not having a primary key.

```
        modelBuilder.HasDbFunction(() =>
            GetBookTitleAndReviewsFiltered(default(int)));
        //… other configurations left out
    }
}
```

You register your UDF method by using the Fluent API.

It might seem strange that you call the method within itself, but remember that you are only defining the signature of your UDF. EF Core will replace the inner method call with a call to your UDF when you use it in a query.

10.1.3 Adding your UDF code to the database

Before you can use the UDF you've configured, you need to get your UDF code into the database. A UDF normally is a set of SQL commands that run on the database, so you need to add your UDF code to the database manually before you call the UDF.

The first way is by adding a UDF by using EF Core's migration feature. To do this, you use the `migrationBuilder.Sql` method described in section 9.5.2. In chapter 15, I use two UDFs to improve the performance of the Book App; I added those UDFs to the database by editing a migration and then added code to create the two UDFs.

Another approach is to add a UDF by using EF Core's `ExecuteSqlRaw` or `ExecuteSqlInterpolated` method, covered in section 11.5. This approach is more applicable to unit testing than to production use where you aren't using migrations to create your database, in which case you must add the UDFs manually. The following listing uses EF Core's `ExecuteSqlRaw` command to add the SQL code that defines the `AverageVotes` UDF.

Listing 10.4 Adding your UDF to the database via the `ExecuteSqlRaw` method

```
public const string UdfAverageVotes =
    nameof(MyUdfMethods.AverageVotes);

context.Database.ExecuteSqlRaw(
    $"CREATE FUNCTION {UdfAverageVotes} (@bookId int)" +
    @"  RETURNS float
      AS
      BEGIN
      DECLARE @result AS float
      SELECT @result = AVG(CAST([NumStars] AS float))
           FROM dbo.Review AS r
           WHERE @bookId = r.BookId
      RETURN @result
      END");
```

Captures the name of the static method that represents your UDF and uses it as the name of the UDF you add to the database

Uses EF Core's ExecuteSqlRaw method to add the UDF to the database

The SQL code that follows adds a UDF to an SQL server database.

This code should be executed before your EF Core queries call the UDF. As I said, chapter 9 gives more details on how to do this properly in a production environment.

NOTE I have not listed the table UDF SQL code in this chapter. You can find the method called `AddUdfToDatabase` in the repo at http://mng.bz/pJQz.

10.1.4 Using a registered UDF in your database queries

Having registered the UDF as mapped to your method and added your UDFs to the database, you're ready to use UDFs in a database query. You can use this method as a return variable or as part of the query filter or sorting. The following listing has a query that includes a call to a scalar-values UDF that returns information about a book, including the average review votes.

Listing 10.5 Using a scalar-valued UDF in a EF Core query

```
var bookAndVotes = context.Books.Select(x => new Dto
{
    BookId = x.BookId,
    Title = x.Title,
    AveVotes = MyUdfMethods.AverageVotes(x.BookId)
}).ToList();
```

A normal EF Core query on the Books table

Calls your scalar valued UDF by using its representing method

This listing produces the following SQL code to run on the database, with the UDF call in bold:

```
SELECT [b].[BookId], [b].[Title],
[dbo].AverageVotes([b].[BookId]) AS [AveVotes]
FROM [Books] AS [b]
```

NOTE EF Core can calculate the average without using a UDF via the LINQ command `x.Reviews.Average(q => (double?)q.NumStars)`. The calculation of the average votes is a running theme in this book, so you use it in the `AverageVotes` UDF example too.

A table-valued UDF requires a class to return the multiple values. The following code snippet shows a call to our `GetBookTitleAndReviewsFiltered` table-valued UDF:

```
var result = context.GetBookTitleAndReviewsFiltered(4)
    .ToList()
```

Scalar and table UDFs can also be used in any part of an EF Core query, as return values or for sorting or filtering. Here's another example, in which your scalar-valued UDF returns only books whose average review is 2.5 or better:

```
var books = context.Books
    .Where(x =>
        MyUdfMethods.AverageVotes(x.BookId) >= 2.5)
    .ToList();
```

10.2 Computed column: A dynamically calculated column value

Another useful SQL-side feature is a computed column (also known as a *generated column*). The main reason for using computed columns is to move some of the calculation—such as some string concatenations—into the database to improve performance. Another good use of computed columns is to return a useful value based on other columns in the row. An SQL computed column containing `[TotalPrice] AS (NumBook * BookPrice)`, for example, would return the total price for that order, making your C# code easier to write.

> **EF6** You can use computed columns in EF6.x, but EF6.x can't create them for you, so you have to add them via a direct SQL command. EF Core now provides a configuration method to define computed columns so that when EF Core creates or migrates a database, it'll add the computed column.

A *computed column* is a column in a table whose value is calculated by using other columns in the same row and/or an SQL built-in function. You can also call systems or UDFs (see section 10.1) with columns as parameters, which gives you a wide range of features.

There are two versions of SQL *computed columns:*

- One that does the calculation every time the column is read. I refer to this type as a *dynamic computed column* in this section.
- One that does the calculation only when the entity is updated. This type is a called *persisted computed column* or *stored generated column*. Not all databases support persisted computed columns.

As an example of both types of SQL computed columns, you'll use a dynamic computed column to get only the year of the person's birth from a backing field that holds the date of birth. This example mimics the code in section 7.14.3 that hides the exact birth date, but now the date-to-year code is done in the SQL database.

The second example of SQL computed columns is a persisted computed column that fixes the problem of not using lambda properties in entity classes (see section 9.3). In that example, you had a `FullName` property, which was formed by combining the `FirstName` and `LastName` properties, but you couldn't use a lambda property, as EF Core can't filter/order on a lambda property. When you use a persisted computed column, however, the computed column is updated whenever the row is updated, and you can use the FullName column in any filter, order, search, and similar operation. You declare the properties in the normal way in the class, as shown in the following listing, but because the computed columns are read-only, you make the setter private.

Listing 10.6 `Person` entity class with two computed column properties

```
public class Person
{
    public int PersonId { get; set; }
    public int YearOfBirth { get; private set; }

    [MaxLength(50)]
    public string FirstName { get; set; }
    [MaxLength(50)]
    public string LastName { get; set; }
    [MaxLength(101)]
    public string FullName { get; private set; }

    //other properties/methods left out…
}
```

This property is a computed column. You give it a private setter, as it's a read-only property.

Because you want to add an index to the FullName, you need make it and its parts fewer than 450 characters.

Then you need to configure the two computed columns and the index. The only way to configure columns is to use the Fluent API. This listing shows the various configurations for the `Person` entity class.

Listing 10.7 Configuring two computed columns, one persistent, and an index

```
public class PersonConfig : IEntityTypeConfiguration<Person>
{
    public void Configure
        (EntityTypeBuilder<Person> entity)
    {
        entity.Property<DateTime>("_dateOfBirth")
            .HasColumnName("DateOfBirth");

        entity.Property(p => p.YearOfBirth)
            .HasComputedColumnSql(
                "DatePart(yyyy, [DateOfBirth])");

        entity.Property(p => p.FullName)
            .HasComputedColumnSql(
                "[FirstName] + ' ' + [LastName]",
                stored:true);

        entity.HasIndex(x => x.FullName);
    }
}
```

Configures the backing field, with the column name DateOfBirth

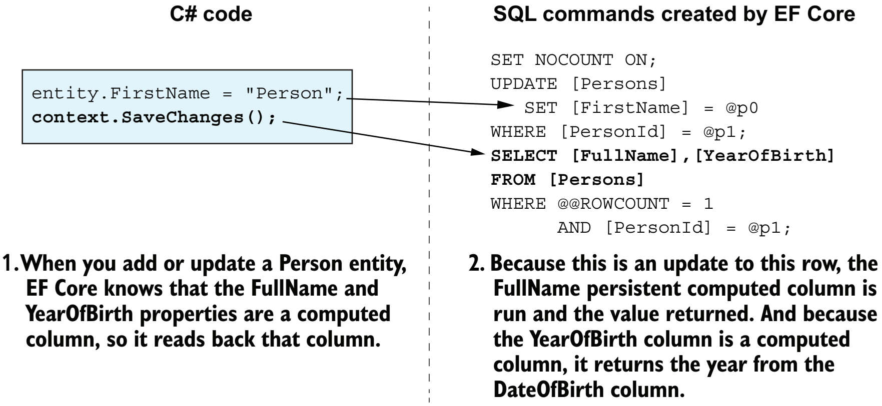
Configures the property as a computed column and provides the SQL code that the database server will run

Makes this computed column a persisted computed column

Adds an index to the FullName column because you want to filter/sort on that column

Figure 10.2 shows what happens when you update the Person table. EF Core knows that the table contains a computed column, so it reads the value back after an add or update.

NOTE To focus this figure on a single computed column, I show only the YearOfBirth column value, but in reality, the YearOfBirth and FullName column values are fed back because the `Person` entity class has two computed columns.



```
entity.FirstName = "Person";
context.SaveChanges();
```

```
SET NOCOUNT ON;
UPDATE [Persons]
   SET [FirstName] = @p0
WHERE [PersonId] = @p1;
SELECT [FullName],[YearOfBirth]
FROM [Persons]
WHERE @@ROWCOUNT = 1
      AND [PersonId] = @p1;
```

1. When you add or update a Person entity, EF Core knows that the FullName and YearOfBirth properties are a computed column, so it reads back that column.

2. Because this is an update to this row, the FullName persistent computed column is run and the value returned. And because the YearOfBirth column is a computed column, it returns the year from the DateOfBirth column.

Figure 10.2 Because EF Core knows that FullName and YearOfBirth are computed columns, it'll read back the values in these two columns into the entity that took part in an addition or update to the row. The FullName will have changed because this operation is an update to the row, and the YearOfBirth column is always recalculated, so it is also returned.

The dynamic computed column is recalculated on each read: for simple calculations, the compute time will be minimal, but if you call a UDF that accesses the database, the time taken to read the data from the database can increase. Using a persisted computed column overcomes this problem. Both types of computed columns can have an index in some database types, but each database type has limitations and restrictions. SQL Server doesn't allow an index on computed columns whose value came from a date function, for example.

10.3 Setting a default value for a database column

When you first create a .NET type, it has a default value: `0` for an `int`, `null` for a `string`, and so on. Sometimes, it's useful to set a different default value for a property. If you asked someone their favorite color, but they didn't reply, you could provide the default string `not given` instead of the normal `null` value. You could set the default value in .NET by using the C# 6.0 autoproperty initializer feature with code such as this:

```
public string Answer { get; set; } = "not given";
```

But with EF Core, you have two other ways to set a default value. First, you can configure EF Core to set up a default value within the database by using the `HasDefaultValue` Fluent API method. This method changes the SQL code used to create the table in the database and adds an SQL `DEFAULT` command containing your default value for that column if no value is provided. Generally, this approach is useful if rows are added to your database via raw SQL commands, as raw SQL often relies on the SQL `DEFAULT` command for columns that the SQL INSERT doesn't provide values for.

The second approach is to create your own code that will create a default value for a column if no value is provided. This approach requires you to write a class that inherits the `ValueGenerator` class, which will calculate a default value. Then you have to configure the property or properties to use your `ValueGenerator` class via the `Configure` Fluent API method. This approach is useful when you have a common format for certain type of values, such as creating a unique string for a user's order of books.

Before exploring each approach, let's define a few things that EF Core's default value-setting methods have in common:

- Defaults can be applied to properties, backing fields, and shadow properties. We'll use the generic term *column* to cover all three types, because they all end up being applied to a column in the database.
- Default values (`int`, `string`, `DateTime`, `GUID`, and so on) apply only to scalar (nonrelational) columns.
- EF Core will provide a default value only if the property contains the CLR default value appropriate to its type. If a property of type `int` has the value `0`, for example, it's a candidate for some form of provided default value, but if the property's value isn't `0`, that nonzero value will be used.
- EF Core's default value methods work at the entity-instance level, not the class level. The defaults won't be applied until you've called `SaveChanges` or (in the case of the value generator) when you use the `Add` command to add the entity.

To be clear: default values happen only on new rows added to the database, not to updates. You can configure EF Core to add a default value in three ways:

- Using the `HasDefaultValue` method to add a constant value for a column
- Using the `HasDefaultValueSql` method to add an SQL command for a column
- Using the `HasValueGenerator` method to assign a value generator to a property

EF6 These three methods for setting a default value are new to EF Core. EF6.x has no equivalent commands.

10.3.1 Using the HasDefaultValue method to add a constant value for a column

The first approach tells EF Core to add the SQL `DEFAULT` command to a column when it creates a database migration, providing a simple constant to be set on a column if a new row is created and the property mapped to that column has a default value. You can add the SQL `DEFAULT` command to a column only via a Fluent API method called `HasDefaultValue`. The following code sets a default date of 1 January 2000 to the column DateOfBirth in the SQL table called People.

Listing 10.8 Configuring a property to have a default value set inside the SQL database

```
protected override void OnModelCreating
   (ModelBuilder modelBuilder)
{
```

You must configure the setting of a default value via Fluent API commands.

```
    modelBuilder.Entity<DefaultTest>()
        .Property("DateOfBirth")
        .HasDefaultValue(new DateTime(2000,1,1));
    //… other configurations left out
}
```

You add an SQL DEFAULT to a column via the HasDefaultValue method.

If the SQL code that EF Core produces is asked to create/migrate an SQL Server database, it looks like the following SQL snippet, with the default constraint in bold:

```
CREATE TABLE [Defaults] (
    [Id] int NOT NULL IDENTITY,
    -- other columns left out
    [DateOfBirth] datetime2 NOT NULL
            DEFAULT '2000-01-01T00:00:00.000',
    CONSTRAINT [PK_Defaults] PRIMARY KEY ([Id])
);
```

If the column in a new entity has the CLR default value, EF Core doesn't provide a value for that column in the SQL `INSERT`, which means that the database server will apply the default constraint of the column definition to provide a value to insert into the new row.

> **NOTE** If you are working with a database not created by EF Core, you still need to register the configuration because EF Core must not set that column if the value in the related property contains the CLR default value for that type.

10.3.2 Using the HasDefaultValueSql method to add an SQL command for a column

Providing a constant default value at the database level doesn't add a lot over setting a default value in your code unless your application, or another application, uses direct SQL commands to create a new row. What is more useful is gaining access to some of SQL's system functions that return the current date/time, which the `HasDefaultValueSql` method allows you to do.

In some situations, it's useful to get the time when a row is added to the database. In such a case, instead of providing a constant in the SQL `DEFAULT` command, you can provide an SQL function that will provide a dynamic value when the row is added to the database. SQL Server, for example, has two functions—`getdate` and `getutcdate`—that provide the current local datatime and the UTC datatime, respectively. You can use these functions to automatically capture the exact time when the row was inserted. The configuration of the column is the same as the constant example in listing 10.8 except that the string used calls the SQL `getutcdate` function, as shown in this code snippet:

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
```

```
    modelBuilder.Entity<DefaultTest>()
        .Property(x => x.CreatedOn)
        .HasDefaultValueSql("getutcdate()");
    …
}
```

If you want to use this column to track when the row was added, you need to make sure that the .NET property isn't set by code (remains at the default value). You do this by using a property with a private setter. The following code snippet shows a property with a private setter and creates a simple tracking value that automatically tells you when the row was first inserted into the database:

```
public DateTime CreatedOn {get; private set;}
```

This feature is a useful one. In addition to accessing system functions such as `getutcdate`, you can place your own SQL UDFs in a default constraint. There's a limit to the SQL commands that you can place—you can't reference another column in the default constraint, for example—but the `HasDefaultValue` Fluent API method can provide useful features compared with setting a default in your code.

10.3.3 Using the HasValueGenerator method to assign a value generator to a property

The third approach to adding a default value is executed not in the database, but inside your EF Core code. EF Core allows the class that inherits from the class `ValueGenerator` or `ValueGenerator<T>` to be configured as a value generator for a property or backing field. This class will be asked for a default value if both of the following statements are true:

- The entity's `State` is set to `Added`; the entity is deemed to be a new entity to be added to the database.
- The property hasn't already been set; its value is at the .NET type's default value.

EF Core has a value generator that will provide unique GUID values for primary keys, for example. But for our example, the following listing shows a simple value generator that creates a unique string by using the `Name` property in the entity, the current date as a string, and a unique string from a GUID to create a value for the property `OrderId`.

Listing 10.9 A value generator that produces a unique string for the `OrderId`

```
public class OrderIdValueGenerator
    : ValueGenerator<string> //
{
    public override bool
        GeneratesTemporaryValues => false;

    public override string Next
        (EntityEntry entry)
```

The value generator needs to inherit from EF Core's ValueGenerator<T>.

Set this to false if you want your value to be written to the database.

This method is called when you Add the entity to the DbContext.

```
    {
        var name = entry.
            Property(nameof(DefaultTest.Name))
                .CurrentValue;
        var ticks = DateTime.UtcNow.ToString("s");
        var guidString = Guid.NewGuid().ToString();
        var orderId = $"{name}-{ticks}-{guidString}";
        return orderId;
    }
}
```

The parameter gives you access to the entity that the value generator is creating a value for. You can access its properties.

Selects the property called "Name" and gets its current value

Provides the date in sortable format

Provides a unique string

The orderId combines these three parts to create a unique orderId containing useful info.

The method must return a value of the Type you have defined at T in the inherited ValueGenerator<T>.

The following code configures the use of a value generator:

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<DefaultTest>()
        .Property(p => p.OrderId)
        .HasValueGenerator((p, e) =>
            new OrderIdValueGenerator());
    …
}
```

Note that the value generator's `Next` method is called when you `Add` the entity via `context.Add(newEntity)` but before the data is written to the database. Any database-provided values, such as the primary key using SQL `IDENTITY`, won't be set when the `Next` method is called.

> **NOTE** You can use a `NextAsync` version if you need to implement an async version, such as using an async method to access the database while generating the default. In that case, you need to use the `AddAsync` method when adding the entity to the database.

The value generator is a specialized feature with limited applications, but one that's worth knowing about. Chapter 11 shows you how to intercept writes to the database to add tracking or other information, which is more work but provides more capabilities than the value generator.

10.4 Sequences: Providing numbers in a strict order

Sequences in a database enable you to produce numbers in strict order with no gaps, such as 1,2,3,4. Key values created by the SQL `IDENTITY` command aren't guaranteed to be in sequence; they might be like this: 1,2,10,11. Sequences are useful when you want a guaranteed known sequence, such as for an order number for purchases.

The way that sequences are implemented differs among database servers, but in general, a sequence is assigned not to a specific table or column, but to a schema. Every time a column wants a value from the sequence, it asks for that value. EF Core can set up a sequence and then, by using the `HasDefaultValueSql` method, set the value of a column to the next value in the sequence.

The following listing shows an `Order` entity class with an `OrderNo` that uses a sequence. The `HasDefaultValueSql` SQL fragment is for an SQL Server database and will be different for other database servers. This example adds an SQL sequence to a migration or to a database created via the `context.Database.EnsureCreated()` method and obtains the next value in the sequence by setting a default value on the OrderNo column.

Listing 10.10 The DbContext with the Fluent API configuration and the `Order` class

```
class MyContext : DbContext
{
    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating
       (ModelBuilder modelBuilder)
    {
        modelBuilder.HasSequence<int>(
                "OrderNumbers", "shared")
            .StartsAt(1000)
            .IncrementsBy(5);

        modelBuilder.Entity<Order>()
            .Property(o => o.OrderNo)
            .HasDefaultValueSql(
                "NEXT VALUE FOR shared.OrderNumbers");
    }
}

public class Order
{
    public int OrderId { get; set; }
    public int OrderNo { get; set; }
}
```

Creates an SQL sequence OrderNumber in the schema "shared." If no schema is provided, it uses the default schema.

(Optional) Allows you to control the sequence's start and increments. The default is to start at 1 and increment by 1.

A column can access the sequence number via a default constraint. Each time the NEXT VALUE command is called, the sequence is incremented.

EF6 This feature is new in EF Core, with no corresponding feature in EF6.

10.5 Marking database-generated properties

When working with an existing database, you may need to tell EF Core about specific columns that are handled differently from what EF Core expects. If your existing database has a computed column that you didn't set up by using EF Core's Fluent API (see section 10.2), EF Core needs to be told that the column is computed so that it handles the column properly.

I should say straightaway that marking columns in this way isn't the norm, because EF can work out the column attributes itself based on the configuration commands you provided. You *don't* need any of the features in this section if you use EF Core to do the following:

- Create or migrate the database via EF Core.
- Reverse-engineer your database, as described in chapter 9. (EF Core reads your database schema and generates your entity classes and application DbContext.)

You might use these features if you want to use EF Core with an existing database without reverse engineering. In that case, you need to tell EF Core about columns that don't conform to its normal conventions. The following sections teach you how to mark three different types of columns:

- Columns that change on inserting a new row or updating a row
- Columns that change on inserting a new row
- "Normal" columns—that is, columns that are changed only by EF Core

EF6 EF6 has the same Data Annotation for setting the database-generated properties, but EF Core provides Fluent API versions too.

10.5.1 Marking a column that's generated on an addition or update

EF Core needs to know whether a column's value is generated by the database, such as a computed column, if for no other reason than it's read-only. EF Core can't "guess" that the database sets a column's value, so you need to mark it as such. You can use Data Annotations or the Fluent API.

The Data Annotation for an add-or-update column is shown in the following code snippet. Here, EF Core is using the existing `DatabaseGeneratedOption.Computed` setting. The setting is called `Computed` because that's the most likely reason for a column to be changed on add or update:

```
public class PersonWithAddUpdateAttibutes
{
    …

    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public int YearOfBirth { get; set; }
}
```

This code snippet uses the Fluent API to set the add-or-update setting for the column:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .Property(p => p.YearOfBirth)
        .ValueGeneratedOnAddOrUpdate();
    …
}
```

10.5.2 Marking a column's value as set on insert of a new row

You can tell EF Core that a column in the database will receive a value via the database whenever a new row is inserted to the database. Two common situations are

- Via an SQL `DEFAULT` command, which provides a default value if no value is given in the `INSERT` command.
- By means of some form of key generation, of which SQL's `IDENTITY` command is the primary method. In these cases, the database creates a unique value to place in the column when a new row is inserted.

If a column has the SQL `DEFAULT` command on it, it will set the value if EF Core creates a new row and no value was provided with a value. In that case, EF Core must read back the value that the SQL `DEFAULT` command set for the column; otherwise, the data inside your entity class will not match the database.

The other situation in which EF Core needs to read back the value of a column is for a primary-key column when the database provides the key value, because EF Core won't know that the key was generated by SQL's `IDENTITY` command. This situation is most likely the reason why the annotation's `DatabaseGeneratedOption` is called `Identity`, as shown in the following code snippet:

```
public class MyClass
{
    public int MyClassId { get; set;}
    …
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int SecondaryKey { get; set;}
}
```

The second example does the same thing but uses the Fluent API. For this example, you have a column with a default constraint. The following snippet of Fluent API code sets this constraint:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .Property("DateOfBirth")
        .ValueGeneratedOnAdd();


    …
}
```

10.5.3 Marking a column/property as "normal"

All scalar properties that aren't keys, don't have an SQL default value, and aren't computed columns are *normal*—that is, only you set the value of the property. In rare cases, you may want to set a property to be normal, and EF Core provides ways to do that. The one case in which this approach might be useful is for a primary key that uses a GUID; in that case, your software supplies the value.

DEFINITION A *GUID* is a *globally unique identifier,* a 128-bit integer that can be used safely anywhere. It makes a good key value in a few cases. In one case, the software wants to define the key, normally because some other part of the software needs the key before the row is inserted. In another case, you have replicated databases with inserts into both or all databases, which makes creating a unique key more difficult.

My tests show that if you use a GUID as a primary key, EF Core will automatically create a GUID value if you don't supply one (EF Core provides a value generator for GUID primary keys). Also, if the database provider is for SQL Server, EF Core uses a value generator called `SequentialGuidValueGenerator`, which is optimized for use in Microsoft SQL server clustered keys and indexes. You can turn this value generator with a Data Annotation:

```
public class MyClass
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public Guid MyClassId { get; set;}
    …
}
```

You can also use the following Fluent API configuration:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<MyClass>()
        .Property("MyClassId")
        .ValueGeneratedNever();
    …
}
```

10.6 Handling simultaneous updates: Concurrency conflicts

Concurrency conflicts represent a big topic, so let me start by explaining what simultaneous updates look like before explaining why they can be problems and how you can handle them. Figure 10.3 shows an example of simultaneous updates to the PublishedOn column in a database. This update happens because of two separate pieces of code running in parallel, which read the column and then update it.

By default, EF Core uses an Optimistic Concurrency pattern. In figure 10.3, the first update is lost because it's overwritten by the second. Although this situation is often acceptable, in some cases, overwriting someone else's update is a problem. The following sections explain unacceptable overwrites, known as *concurrency conflicts,* and show how EF Core enables you to detect and fix such conflicts.

10.6.1 Why do concurrency conflicts matter?

If you think about it, a setting can be overwritten anyway. You could set the publication date of a book to 1/1/2020, and tomorrow you could change it to 1/1/2040, so why are concurrency conflicts such a big deal?

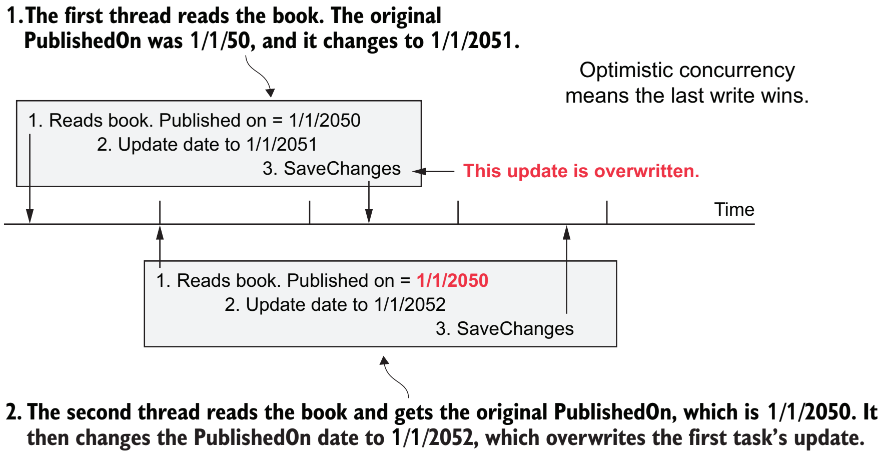


Figure 10.3 Two pieces of code (say, in a web application) running in parallel that make near-simultaneous updates of the same column (in this case, the publication date of the same book). By default, EF Core allows the second write to win, and the first write is lost. This situation is called optimistic concurrency, but the "last write win" rule may not be useful in all cases.

In some cases, concurrent conflicts do matter. In financial transactions, for example, you can imagine that the purity and auditing of data are going to be important, so you might want to guard against concurrency changes. Another concurrent conflict exists in the example in section 8.7, where you calculated the average book review votes. In that case, if two people added reviews at the same time, that recalculation would be incorrect, so you need to detect and fix that conflict if that example is going to be robust.

Other human-level concurrent conflicts can occur. Instead of two tasks clashing on updates, two users looking at screens can clash, with the same default result: the second person to click the Submit button overwrites the update that the first person thought they had done. (Section 10.6.4 covers the details.)

Sometimes, you get around concurrency conflicts by design, by creating applications in such a way that dangerous concurrent updates can't happen. For an e-commerce website that I designed, for example, I had an order-processing system that used background tasks, which could've caused concurrent conflicts. I got around this potential problem by designing the order processing to remove the possibility of concurrent updates:

- I split the customer order information into an immutable order part that never changed. This part contained data such as what was ordered and where it should be sent. After that order was created, it was never changed or deleted.
- For the changing part of the order, which was the order status as it moved through the system, I created a separate table to which I added each new order status as it occurred, with the date and time. (This approach is known as *event*

sourcing.) Then I could get the latest order status by sorting by date/time order and picking the status with the newest date and time. This result would be out of date if another status was added after I read the status, of course, but concurrency handling would detect this addition.

This design approach meant that I never updated or deleted any order data, so concurrent conflicts couldn't happen. It did make handling a customer change to an order a bit more complicated, but orders were safe from concurrent-conflict issues.

But when concurrent conflicts *are* issues, and you can't design around them, EF Core provides two ways of detecting a concurrent update and, when the update is detected, a way of getting at all the relevant data so you can implement code to fix the issue.

10.6.2 EF Core's concurrency conflict–handling features

EF Core's concurrency conflict-handling features can detect a concurrency update in two ways, activated by adding one of the following to an entity class:

- A *concurrency token* to mark a specific property/column in your entity class as one to check for a concurrency conflict
- A *timestamp* (also known as a rowversion), which marks a whole entity class/row as one to check for a concurrency conflict

EF6 Concurrency-handling features are the same as in EF6.x but reimplemented in EF Core.

In both cases, when `SaveChanges` is called, EF Core produces database server code to check for updates of any entities that contain concurrency tokens or timestamps. If that code detects that the concurrency tokens or timestamps have changed since it read the entity, it throws a `DbUpdateConcurrencyException` exception. At that point, you can use EF Core's features to inspect the differing versions of the data and apply your custom code to decide which of the concurrent updates wins. Next, you'll learn how to set up the two approaches—a concurrency token and a timestamp—and how EF Core detects the change.

DETECTING A CONCURRENT CHANGE VIA CONCURRENCY TOKEN

The concurrency-token approach allows you to configure one or more properties as concurrency tokens. This approach tells EF Core to check whether the current database value is the same as the value found when the tracked entity was loaded as part of the SQL `UPDATE` command sent to the database. That way, the update will fail if the loaded value and the current database value are different. Figure 10.4 shows an example of marking the `PublishedOn` property as a concurrency token, after which a concurrency conflict occurs.

To set up this example, you add the `ConcurrencyCheck` Data Annotation to the `PublishedOn` property in our `ConcurrencyBook` entity class, shown in the following listing. EF Core finds this Data Annotation during configuration and marks the property as a concurrency token.

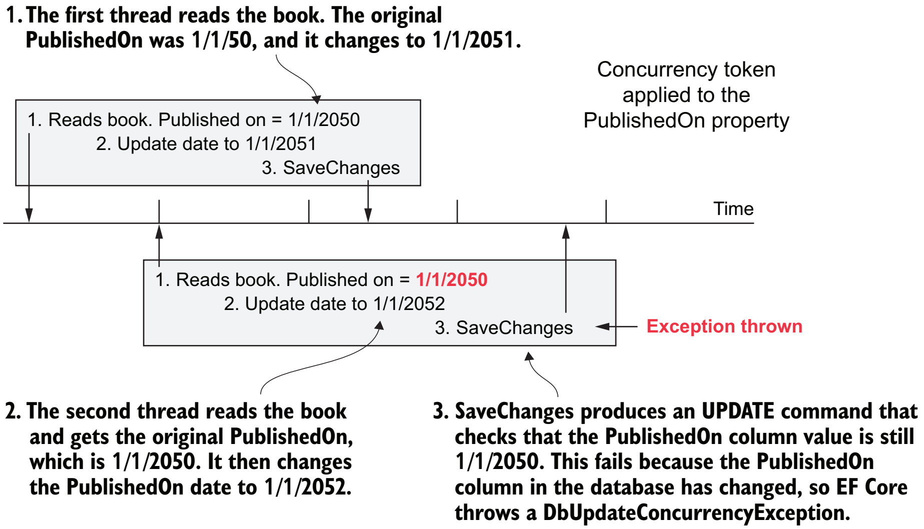


Figure 10.4 Two pieces of code—say, in a web application—running in parallel that make a near-simultaneous update of the `PublishedOn` column. Because you've marked the `PublishedOn` property as a concurrency token, EF Core uses a modified SQL `UPDATE` command that performs the update only if the database's PublishedOn column is the same as it was when it read in the `Book` entity. If it isn't the same, the `UPDATE` fails, and `SaveChanges` throws a `DbUpdateConcurrencyException`.

Listing 10.11 The `ConcurrencyBook` entity class, with a `PublishedOn` property

```
public class ConcurrencyBook
{
    public int ConcurrencyBookId { get; set; }
    public string Title { get; set; }

    [ConcurrencyCheck]
    public DateTime PublishedOn { get; set; }

    public ConcurrencyAuthor Author { get; set; }
}
```

Tells EF Core that the PublishedOn property is a concurrency token, which means that EF Core will check whether it has changed when you update it

In this case, you've used the `ConcurrencyCheck` Data Annotation to define the property as a concurrency token, which has the benefit of making it clear to anyone looking at the code that the `PublishedOn` property has special handling. Alternatively, you can define a concurrency token via the Fluent API, as shown in the next listing.

Listing 10.12 Setting a property as a concurrency token by using the Fluent API

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
```

The OnModelCreating method is where you place the configuration of the concurrency detection.

```
    modelBuilder.Entity<ConcurrencyBook>()
        .Property(p => p.PublishedOn)
        .IsConcurrencyToken();

    //… other configuration removed
}
```

Defines the PublishedOn property as a concurrency token, which means that EF Core checks whether it has changed when writing out an update

After either listing 10.11 or 10.12 is added, figure 10.4 shows that when `SaveChanges` is called, instead of overwriting the first update, it detects that another task has updated the PublishedOn column and throws an exception.

Listing 10.13 simulates a concurrent update by running an SQL command that changes the PublishedOn column between the EF Core code that reads and then updates the book. The SQL command represents another thread of the web application, or another application that has access to the same database, updating the PublishedOn column. In this case, a `DbUpdateConcurrencyException` exception is thrown when `SaveChanges` is called in the last line.

Listing 10.13 Simulating a concurrent update of the PublishedOn column

```
var firstBook = context.Books.First();

context.Database.ExecuteSqlRaw(
    "UPDATE dbo.Books SET PublishedOn = GETDATE()"+
    " WHERE ConcurrencyBookId = @p0",
    firstBook.ConcurrencyBookId);
firstBook.Title = Guid.NewGuid().ToString();
context.SaveChanges();
```

Loads the first book in the database as a tracked entity

Simulates another thread/application, changing the PublishedOn column of the same book

Changes the title in the book to cause EF Core to update the book

This SaveChanges throws a DbUpdateConcurrencyException.

The important thing to note is that only the property marked as a concurrency token is checked. If your SQL-simulated update changed, say, the `Title` property, which isn't marked as a concurrency token, no exception would be thrown.

You can see this effect in the SQL that EF Core produces to update the `Title` in the next listing. The SQL `WHERE` clause contains not only the primary key of the book to update, but also the PublishedOn column.

Listing 10.14 SQL code to update `Book` where PublishedOn is a concurrency token

```
SET NOCOUNT ON;
UPDATE [Books] SET [Title] = @p0
WHERE [ConcurrencyBookId] = @p1
      AND [PublishedOn] = @p2;
SELECT @@ROWCOUNT;
```

The test fails if the PublishedOn column has changed, which stops the update.

Returns the number of rows updated by this SQL command

When EF Core runs this SQL command, the `WHERE` clause finds a valid row to update only if the PublishedOn column hasn't changed from the value EF Core read in from the database. Then EF Core checks the number of rows that have been updated by the

SQL command. If the number of rows updated is zero, EF Core raises `DbUpdateConcurrencyException` to say that a concurrency conflict exists; it can catch a concurrency conflict caused by another task by changing the PublishedOn column or deleting the row when this task does an update.

The good thing about using a concurrency token is that it works on any database because it uses basic commands. The next way of detecting concurrency changes relies on a database server-side feature.

DETECTING A CONCURRENT CHANGE VIA TIMESTAMP

The second way to check for concurrency conflicts is to use what EF Core calls a timestamp. A timestamp works differently from a concurrency token, as it uses a unique value provided by the database server that changes whenever a row is inserted or updated. The whole entity, rather than specific properties or columns, is protected against concurrency changes.

Figure 10.5 shows that when a row with a property/column marked as a timestamp is inserted or updated, the database server produces a new, unique value for that column, which has the effect of detecting an update to an entity/row whenever `SaveChanges` is called.

The timestamp database type is database-type-specific: SQL Server's concurrency type is `ROWVERSION`, which maps to `byte[]` in .NET; PostgreSQL has a column called

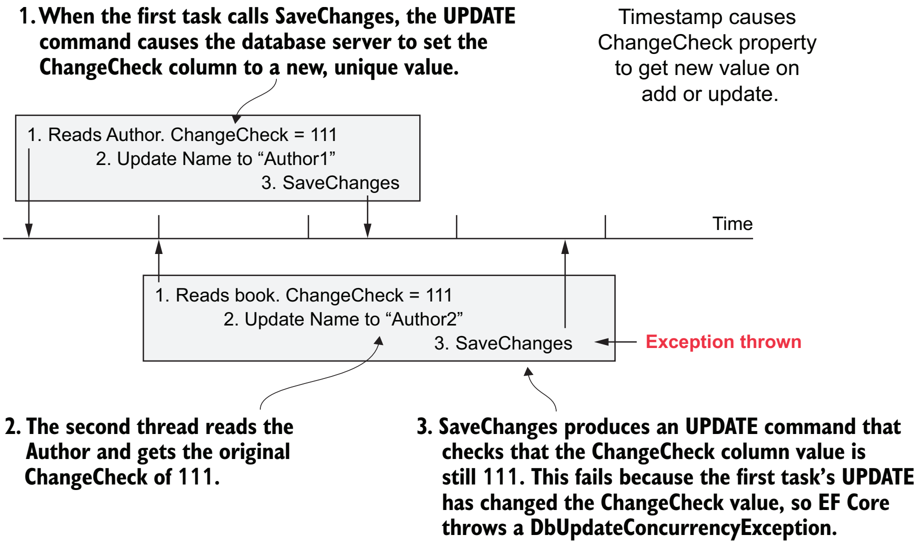


Figure 10.5 Configuring a property as a timestamp means that the corresponding column in the table must be set to a database server type that will be set to a new, unique value every time an SQL `INSERT` or `UPDATE` command is applied to the row. (If you use EF Core to create your database, the database provider will ensure the use of the correct column type.) Then, when EF Core does an update, it checks that the timestamp column has the same value as when the entity was read in. If the value is different, EF Core will throw an exception.

xmin that is an unsigned 32-bit number; and Cosmos DB has a JSON property called `_etag`, which is a string containing a unique value. EF Core can use any of these types via the appropriate database provider. For the examples of using a timestamp, I'm going to use SQL Server's timestamp; other databases will work in a similar way, but with a different .NET type.

The following listing adds a `ChangeCheck` property, which watches for any updates to the whole entity, to an entity class called `ConcurrencyAuthor`. In this case, the `ChangeCheck` property has a `Timestamp` Data Annotation, which tells EF Core to mark it as a special column that the database will update with a unique value. In the case of SQL Server, the database provider will set the column as an SQL Server `rowversion`; other databases have different approaches to implementing the `TimeStamp` column.

Listing 10.15 The `ConcurrencyAuthor` class, with the `ChangeCheck` property

```
public class ConcurrencyAuthor
{
    public int ConcurrencyAuthorId { get; set; }
    public string Name { get; set; }
    [Timestamp]
    public byte[] ChangeCheck { get; set; }
}
```

Marks the ChangeCheck property as a timestamp, causing the database server to mark it as an SQL ROWVERSION. EF Core checks this property when updating to see whether it has changed.

Again, you use a Data Annotation, `Timestamp`, to mark the `ChangeCheck` property as a timestamp. This approach is my recommended way of configuring concurrency handling, because it makes obvious to anyone looking at the code that there's special concurrency handling of this entity. Alternatively, you can use the Fluent API to configure a timestamp, as shown in the following listing.

Listing 10.16 Configuring a timestamp by using the Fluent API

```
protected override void
    OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<ConcurrencyAuthor>()
        .Property(p => p.ChangeCheck)
        .IsRowVersion();
}
```

OnModelCreating is where you place the configuration of the concurrency detection.

Defines an extra property called ChangeCheck that will be changed every time the row is created/updated. EF Core checks whether this property has changed when it does an update.

Both configurations create a column in a table that the database server will change automatically whenever there's an `INSERT` or `UPDATE` to that table. For SQL Server database, the column type is set to `ROWVERSION`, as shown in the following listing. Other database servers can use different approaches, but they all provide a new, unique value on an `INSERT` or `UPDATE`.

Listing 10.17 The SQL to create the Authors table, with a `timestamp` column

```
CREATE TABLE [dbo].[Authors] (
    [ConcurrencyAuthorId] INT  IDENTITY (1, 1),
    [ChangeCheck]         TIMESTAMP NULL,
    [Name]                NVARCHAR (MAX) NULL
);
```

If the table is created by EF Core, sets the column type to TIMESTAMP if your property is of type byte[]. This column's value will be updated on each INSERT or UPDATE.

You simulate a concurrent change by using the code in listing 10.18, which consists of three steps:

1. You use EF Core to read in the Authors row that you want to update.
2. You use an SQL command to update the Authors table, simulating another task updating the same Author that you read in. EF Core doesn't know anything about this change because raw SQL bypasses EF Core's tracking snapshot feature. (See section 11.5 in chapter 11 for details about EF Core's raw SQL commands.)
3. In the last two lines, you update the Author's name and call `SaveChanges`, which causes a `DbUpdateConcurrencyException` to be thrown because EF Core found that the ChangeCheck column has changed from step 1.

Listing 10.18 Simulating a concurrent update of the `ConcurrentAuthor` entity

```
var firstAuthor = context.Authors.First();
context.Database.ExecuteSqlRaw(
    "UPDATE dbo.Authors SET Name = @p0"+
    " WHERE ConcurrencyAuthorId = @p1",
    firstAuthor.Name,
    firstAuthor.ConcurrencyAuthorId);
firstAuthor.Name = "Concurrency Name";
context.SaveChanges();
```

Loads the first author in the database as a tracked entity

Simulates another thread/application updating the entity. Nothing is changed except the timestamp.

Changes something in the author to cause EF Core to do an update to the book

Throws DbUpdateConcurrencyException

This code is like the case in which you used a concurrency token. The difference is that the timestamp detects an update of the row via the unique value in the property/column called ChangeCheck. You can see this difference in the following listing, which shows the SQL that EF Core produces to update the row with the check on the timestamp property, `ChangeCheck`.

Listing 10.19 The SQL code to update the author's name, with ChangeCheck check

```
SET NOCOUNT ON;
UPDATE [Authors] SET [Name] = @p0
WHERE [ConcurrencyAuthorId] = @p1
      AND [ChangeCheck] = @p2;
SELECT [ChangeCheck]
FROM [Authors]
```

Checks that the ChangeCheck column hasn't been changed since you read in the book entity

Because the update will change the ChangeCheck column, EF Core needs to read it back so that its in-memory copy is correct.

```
WHERE @@ROWCOUNT = 1
      AND [ConcurrencyAuthorId] = @p1;
```

Checks whether one row was updated in the last command. If not, the ChangeCheck value won't be returned, and EF Core will know that a concurrent change has taken place.

The `UPDATE` part checks whether the ChangeCheck column is the same value as the copy it found when it first read the entity, and if so, it executes the update. The second part returns the new ChangeCheck column that the database server created after the current update, but only if the `UPDATE` was executed. If no value is returned for the `ChangeCheck` property, EF Core knows that a concurrency conflict has happened and throws a `DbUpdateConcurrencyException`.

Your choice between the two approaches—concurrency token and timestamp—depends on your business rules. The concurrency-token approach provides specific protection of the property/properties you place it on and is triggered only if a property marked as a concurrency token is changed. The timestamp approach catches any update to that entity.

10.6.3 Handling a DbUpdateConcurrencyException

Now that you've seen the two ways that EF Core detects a concurrent change, you're ready to look at an example of catching `DbUpdateConcurrencyException`. The way you write your code to fix a concurrency conflict depends on your business reasons for capturing it. The example in listing 10.20 shows how to capture the `DbUpdateConcurrencyException` and what data you have available for making your decisions to fix this concurrency exception.

Listing 10.20 shows a method that you call after you've updated the `Book` entity with your change. This method, `BookSaveChangesWithChecks`, calls `SaveChanges` and captures any `DbUpdateConcurrencyException` exception if one happens; it also uses a method called `HandleBookConcurrency`, where you've put the logic to handle a concurrency exception on a `Book` entity.

Listing 10.20 The method you call to save changes that trap concurrency conflicts

```
public static string BookSaveChangesWithChecks
    (ConcurrencyDbContext context)
{
    string error = null;
    try
    {
        context.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var entry = ex.Entries.Single();
        error = HandleBookConcurrency(
            context, entry);
        if (error == null)
            context.SaveChanges();
    }
```

Called after the Book entity has been updated in some way

Calls SaveChanges within a try...catch so that you can catch DbUpdateConcurrencyException if it occurs

Catches DbUpdateConcurrencyException and puts in your code to handle it

In this case, you know that only one Book will be updated. In other cases, you might need to handle multiple entities.

Calls the HandleBookConcurrency method, which returns null if the error was handled or an error message if it wasn't

If the conflict was handled, you need to call SaveChanges to update the Book.

```
    return error;
}
```

Returns the error message or null if there's no error

The `BookSaveChangesWithChecks` method returns a string, which is `null` if successful or an error message if it can't handle this concurrency conflict. (In this example, you handle an update conflict, but you return an error message on a delete conflict; see the `HandleBookConcurrency` method in listing 10.21.) Note that you must call the `SaveChanges` method again, but only if you've fixed the concurrency problem. Otherwise, the method keeps looping around with the same exception.

The `HandleBookConcurrency` method handles a `Book` entity update concurrency conflict. You have at your disposal three versions of the database data, shown in the Exception Handler box in figure 10.6. In this example, you're looking at the `PublishedOn` property, which is protected by a concurrency token. Figure 10.6 shows the sequence of events and the value of the PublishedOn column at each stage.

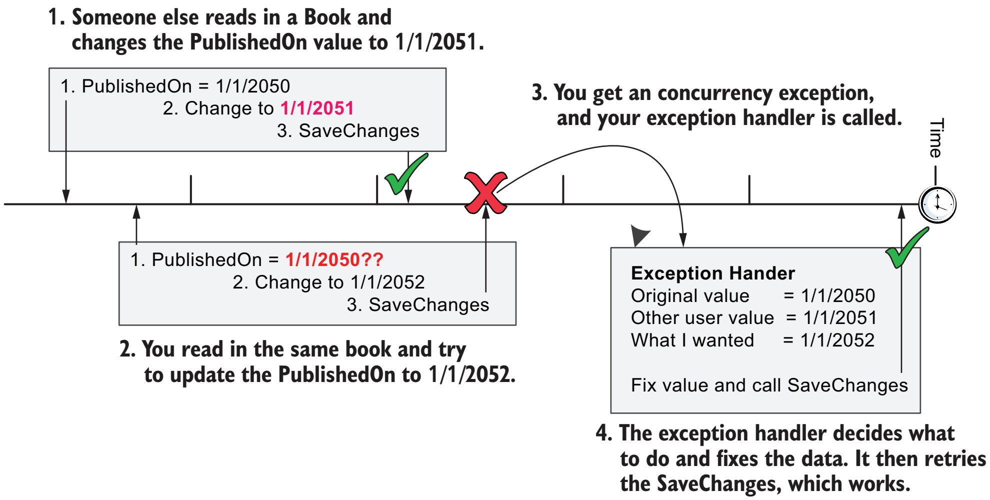


Figure 10.6 An overlap of two people accessing the same book can be caught by using a concurrency token in this example (but the stages would be the same if a timestamp were used). At stage 3, a concurrency exception happens, and your exception handler is called (see listing 10.21). Your exception handler gets a copy of the original `PublishedOn` date that your change read in, the value that the other user set the `PublishedOn` date to, and the actual value you wanted to set the `PublishedOn` date to.

Listing 10.21 shows the content of your exception hander called `HandleBookConcurrency`. The code names some of the variables, starting with `originalValue`, `otherUserValue`, and `whatIWantedItToBe`. These variables correspond to the three versions of the data shown in figure 10.6.

Listing 10.21 Handling a concurrent update on the book

```
private static string HandleBookConcurrency(
    DbContext context,
    EntityEntry entry)
{
    var book = entry.Entity
        as ConcurrencyBook;
    if (book == null)
        throw new NotSupportedException(
"Don't know how to handle concurrency conflicts for " +
            entry.Metadata.Name);

    var whatTheDatabaseHasNow =
        context.Set<ConcurrencyBook>().AsNoTracking()
            .SingleOrDefault(p => p.ConcurrencyBookId
                == book.ConcurrencyBookId);
    if (whatTheDatabaseHasNow == null)
        return "Unable to save changes.The book was deleted by another
     user.";

    var otherUserData =
        context.Entry(whatTheDatabaseHasNow);

    foreach (var property in entry.Metadata.GetProperties())
    {
        var theOriginalValue = entry
            .Property(property.Name).OriginalValue;
        var otherUserValue = otherUserData
            .Property(property.Name).CurrentValue;
        var whatIWantedItToBe = entry
            .Property(property.Name).CurrentValue;

        // TODO: Logic to decide which value should be written to database
        if (property.Name ==
            nameof(ConcurrencyBook.PublishedOn))
        {
            entry.Property(property.Name).CurrentValue =
            //… your code to pick which PublishedOn to use
        }

        entry.Property(property.Name).OriginalValue =
            otherUserData.Property(property.Name)
                .CurrentValue;
    }
    return null;
}
```

Takes in the application's DbContext and the ChangeTracking entry from the exception's Entities property

You want to get the data that someone else wrote into the database after your read.

Handles only ConcurrencyBook, so throws an exception if the entry isn't of type Book

Entity must be read as NoTracking; otherwise, it'll interfere with the same entity you're trying to write.

Concurrency conflict method doesn't handle the case where the book was deleted, so it returns a user-friendly error message.

You get the EntityEntry<T> version of the entity, which has all the tracking information.

You go through all the properties in the book entity to reset the Original values so that the exception doesn't happen again.

Holds the version of the property at the time you did the tracked read of the book

Holds the version of the property as written to the database by someone else

Holds the version of the property that you wanted to set it to in your update

Business logic to handle PublishedOn: sets to your value or the other person's value, or throws an exception

Here, you set the OriginalValue to the value that someone else set it to. This code works for concurrency tokens or a timestamp.

You return null to say that you handled this concurrency issue.

The main part you need to change is the section starting with the comment `// TODO`. You should put your code to handle the concurrent update there. What you put there depends on the business rules in your application. In section 10.6.4, I show you a worked-through example with business logic, but in listing 10.21, the focus is on the three parts of the data: the original values, the other users' values, and what you want the `PublishedOn` value to be.

Note that your `HandleBookConcurrency` method also detects that a concurrency conflict caused by the original `Book` entity has been deleted. In that case, when your concurrency-handling method tries to reread the actual row in the database by using the `Book`'s primary key, it won't find that row and will return `null`. Your current implementation doesn't handle that case and returns an error message to show the user.

Reference to more complex concurrency examples

Because concurrency handling is pretty hard to understand, I have made two simplifications on the descriptions in this chapter. They are as follows:

- `HandleBookConcurrency` shown in listing 10.21 handles only one entity.
- The `BookSaveChangesWithChecks` method shown in listing 10.20 assumes that a second concurrency issue isn't thrown when the `HandleBookConcurrency` code has corrected the first concurrency issue.

In real applications, you might need to handle multiple entities in your concurrency handler, and you can't assume that you won't get another concurrency exception when you write the corrected entity that threw the first concurrency exception. Fortunately, chapter 15 provides examples of handling both of these issues.

In section 15.5, I describe a way to store values that contain preevaluated values, such as the average votes for a book, to improve the performance of the Book App when working with large amounts of data. These extra values have to be updated whenever the appropriate entities are changed, but of course, multiple updates would cause concurrency issues, so I had to solve both of these issues.

For simplification 1 (only one entity), have a look at listing 15.9, which handles multiple entities with concurrency issues and also different types of concurrency issues within one entity class.

For simplification 2 (concurrency within a concurrency), see listing 15.8, which adds a `do` / `while` loop around the call to `SaveChanges`. This loop means the code will catch a concurrency within a concurrency; the concurrency handles are designed for that possibility.

10.6.4 The disconnected concurrent update issue

In applications such as websites, another concurrency-update scenario can occur that encompasses the user-interaction part of the system. The examples so far have covered simultaneous code updates, but if you bring in the human factor, the problem is more likely to occur and may be more business-relevant.

Figure 10.7 shows employee John Doe getting a pay raise set by both John's boss and human resources. The time between each entity's seeing the figure and deciding what to do is measured in minutes instead of milliseconds, but if you don't do anything

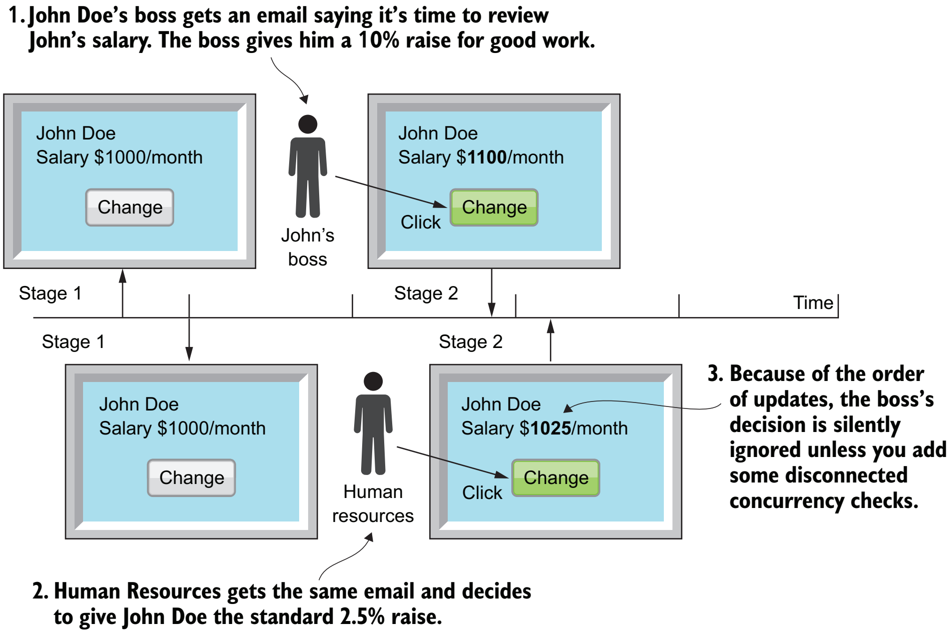


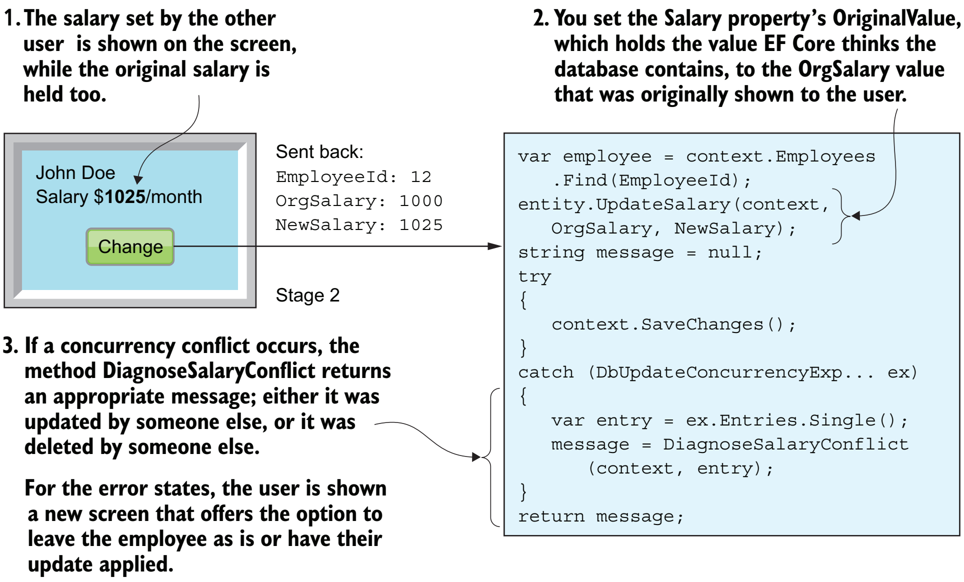
Figure 10.7 A concurrency problem running in human time. John Doe's salary review is due, and two people—John's boss and a human resources employee—try to update his salary at the same time. Unless you add concurrency checks, the boss's update, which came first, is silently ignored, which most likely isn't the correct business outcome.

about it, you can have another concurrency conflict, potentially with the wrong salary being set.

Although this example looks much like the concurrency-conflicts example in section 10.6.2, the change is in the way that a disconnected concurrency conflict is found. To handle a disconnected update, the original value of the property you're protecting (in this case, the `Salary`) must be passed from the first stage of the disconnect to the second stage. Then your second stage must use that original `Salary` in the concurrency-conflict check during the update part of the process.

Also, the way that a concurrency conflict is dealt with is different. Typically, in a human-user case, the decision about what should happen is given back to the user. If a conflict occurs, the user is shown a new screen indicating what happened and is given a choice about what should be done. This disconnected concurrent situation changes the code that handles `DbUpdateConcurrencyException` into more of a diagnostic role than code that fixes the problem.

If a concurrency conflict exists, the user is shown a new screen with an error message indicating what happened. Then the user is invited to accept the current state or apply the update, knowing that this update will override the last user's update.



1. The salary set by the other user is shown on the screen, while the original salary is held too.

2. You set the Salary property's OriginalValue, which holds the value EF Core thinks the database contains, to the OrgSalary value that was originally shown to the user.

Sent back:

```
EmployeeId: 12
OrgSalary: 1000
NewSalary: 1025
```

```
var employee = context.Employees
    .Find(EmployeeId);
entity.UpdateSalary(context,
    OrgSalary, NewSalary);
string message = null;
try
{
    context.SaveChanges();
}
catch (DbUpdateConcurrencyExp... ex)
{
    var entry = ex.Entries.Single();
    message = DiagnoseSalaryConflict
        (context, entry);
}
return message;
```

3. If a concurrency conflict occurs, the method DiagnoseSalaryConflict returns an appropriate message; either it was updated by someone else, or it was deleted by someone else.

For the error states, the user is shown a new screen that offers the option to leave the employee as is or have their update applied.

Figure 10.8 After the user has changed the salary and clicked the Change button, the new salary and the original salary values are sent back to the web application. Then the application calls the `UpdateSalary` method, shown in listing 10.24, which both updates the salary and sets the original value expected in the database when it does the update. If a concurrency conflict is found, a new screen with an appropriate error message is shown to the user, who can then accept the existing database state or apply their own update to the employee.

Figure 10.8 shows what happens when the user clicks the Change button after setting the new salary. As you can see, the original salary, which was displayed to the user on the first screen, is sent back with the other data and used in the concurrency check when the `Salary` is updated. (See the `UpdateSalary` method in listing 10.24.)

Listing 10.22 shows the entity class used for this example, with the `Salary` property set as a concurrency token. You also create a method called `UpdateSalary` that contains the code you need to execute to update the `Salary` property in such a way that `DbUpdateConcurrencyException` will be thrown if the `Salary` value has changed from the value originally shown on the user's screen.

Listing 10.22 Entity class used to hold an employee's salary with concurrency check

```
public class Employee
{
    public int EmployeeId { get; set; }

    public string Name { get; set; }

    [ConcurrencyCheck]
    public int Salary { get; set; }
```

Salary property set as a concurrency token by the ConcurrencyCheck attribute

```
  public void UpdateSalary
    (DbContext context,
     int orgSalary, int newSalary)
  {
    Salary = newSalary;
    context.Entry(this).Property(p => p.Salary)
      .OriginalValue = orgSalary;
  }
}
```

Updates the Salary in a disconnected state

Sets the Salary to the new value

Sets the OriginalValue, which holds the data read from the database, to the original value that was shown to the user in the first part of the update

After applying the `UpdateSalary` method to the `Employee` entity of the person whose salary you want to change, you call `SaveChanges` within a `try…catch` block to update the `Employee`. If `SaveChanges` raises `DbUpdateConcurrencyException`, the job of the `DiagnoseSalaryConflict` method shown in the following listing isn't to fix the conflict, but to create an appropriate error message so that the user can decide what to do.

Listing 10.23 Returns different errors for update or delete concurrency conflicts

```
private string DiagnoseSalaryConflict(
  ConcurrencyDbContext context,
  EntityEntry entry)
{
  var employee = entry.Entity
    as Employee;
  if (employee == null)
    throw new NotSupportedException(
"Don't know how to handle concurrency conflicts for " +
      entry.Metadata.Name);

  var databaseEntity =
    context.Employees.AsNoTracking()
      .SingleOrDefault(p =>
        p.EmployeeId == employee.EmployeeId);

  if (databaseEntity == null)
    return
$"The Employee {employee.Name} was deleted by another user. " +
$"Click Add button to add back with salary of {employee.Salary}" +
" or Cancel to leave deleted.";

  return
$"The Employee {employee.Name}'s salary was set to " +
$"{databaseEntity.Salary} by another user. " +
$"Click Update to use your new salary of {employee.Salary}" +
$" or Cancel to leave the salary at {databaseEntity.Salary}.";
}
```

Called if a DbUpdateConcurrencyException occurs. Its job isn't to fix the problem but to form an error message and provide options for fixing the problem.

If the entity that failed wasn't an Employee, you throw an exception, as this code can't handle that.

You want to get the data that someone else wrote into the database after your read.

Must be read as NoTracking; otherwise, it'll interfere with the same entity you're trying to write.

Checks for a delete conflict: the employee was deleted because the user attempted to update it.

Error message to display to the user, with two choices about how to carry on

Otherwise, the error must be an update conflict, so you return a different error message with the two choices for this case.

Listing 10.24 shows two methods: one for the update conflict case and one for the delete conflict. These methods are called depending on which sort of concurrency conflict was found (update or delete), and only if the user wants to apply an update to `Employee`.

The update conflict can be handled by using the same `UpdateSalary` method used for the normal update, but now the `orgSalary` parameter is the salary value as read back when the `DbUpdateConcurrencyException` was raised. The `FixDeleteSalary` method is used when the concurrent user deletes the `Employee` and the current user wants to add the `Employee` back with their new salary value.

Listing 10.24 Two methods to handle update and delete conflicts

```
public class Employee
{
    public int EmployeeId { get; set; }

    public string Name { get; set; }

    [ConcurrencyCheck]
    public int Salary { get; set; }

    public void UpdateSalary
        (DbContext context,
         int orgSalary, int newSalary)
    {
        Salary = newSalary;
        context.Entry(this).Property(p => p.Salary)
            .OriginalValue = orgSalary;
    }

    public static void FixDeletedSalary
    (DbContext context,
        Employee employee)
    {
        employee.EmployeeId = 0;
        context.Add(employee);
    }
}
```

Set as a concurrency token by the ConcurrencyCheck attribute

The same method used to update the Salary can be used for the Update conflict, but this time, it's given the original value that was found when the DbUpdateConcurrencyException occurred.

Sets the Salary to the new value

Sets the OriginalValue, which is now the value that the database contained when the DbUpdateConcurrencyException occurred

Handles the Delete concurrency conflict

The key must be at the CLR default value for an Add to work.

Adds the Employee because it was deleted from the database and therefore must be added back

NOTE These disconnected concurrency-conflict examples use a concurrency token, but they work equally well with a timestamp. To use a timestamp instead of passing the `Salary` concurrency token used in these examples, you'd pass the timestamp and set the timestamp's original value before any update.

Summary

- Using SQL user-defined functions (UDFs) with EF Core to move calculations into the database can improve query performance.
- Configuring a column as an SQL computed column allows you to return a computed value based on the other properties in the row.

- EF Core provides two ways to set a default value for a property/column in an entity; these techniques go beyond what setting a default value via .NET could achieve.
- EF Core's `HasSequence` method allows a known, predictable sequence provided by the database server to be applied to a column in a table.
- When the database is created/migrated outside EF Core, you need to configure columns that behave differently from the norm, such as telling EF Core that a key is generated in the database.
- EF Core provides concurrency tokens and timestamps to detect concurrency conflicts.
- When a concurrency conflict is detected, EF Core throws `DbUpdateConcurrencyException` and then allows you to implement code to handle the conflict.

For readers who are familiar with EF6:

- The three default value methods, the `HasSequence` method, and the setting of a computed column aren't available in EF6.x.
- EF Core's handling of a concurrency conflict is identical to the way that EF6.x handles a concurrency conflict, but Microsoft suggests a few minor changes in the way that the `DbUpdateConcurrencyException` should be handled; see http://mng.bz/O1VE.

11

Going deeper into the DbContext

This chapter covers

- Seeing how your application's DbContext detects changes in tracked entities
- Using the change tracking method in your DbContext to build an audit trail
- Using raw SQL commands via the DbContext's `Database` property
- Finding the entities to database mapping using DbContext's `Model` property
- Using EF Core's database connection resiliency

This chapter looks at the properties and methods available in the application's DbContext. You've seen a few of these properties and methods before, such as the `Add`, `Update,` and `Remove` methods covered in chapter 3, but in this chapter, you'll dig deeper into how they work. You'll also look at some other properties and methods that haven't been covered in earlier chapters. You will look at each method used to write to the database, ways to make saving data quicker, and ways to execute SQL commands directly on your database. You'll also look at accessing and using your EF Core configuration information.

This chapter discusses the `DbContext` properties for setting the `State` of an entity class, including what to do if your call to `SaveChanges` is taking too long to run. But we'll start with an overview of the four properties in the `DbContext` class, with pointers to coverage of their related features.

11.1 Overview of the DbContext class's properties

Your application's DbContext, which inherits EF Core's `DbContext` class, is the key to accessing your database. Everywhere your application wants to use EF Core, it has to use an instance of your application's DbContext.

This chapter focuses on the methods and data of the public properties that were inherited from EFCore's `DbContext` class. These properties provide information or methods that allow you to better manage your entity classes and their mapping to your database:

- `ChangeTracker`—Provides access to EF Core's change tracking code. You used the `ChangeTracker` property in chapter 4 to run data validation before `SaveChanges`. You'll spend quite a bit of time looking at an entity class's `State` in this chapter, including the `ChangeTracker` property (section 11.4).
- `ContextId`—A unique identifier for the instance of the `DbContext`. Its main role is to be a correlation ID for logging and debugging so that you can see what reads and writes were done from the same instance of the application's DbContext.
- `Database`—Provides access to three main groups of features:
  - Transaction control, covered in section 4.7.2
  - Database creation/migration, covered in chapter 9
  - Raw SQL commands, covered in section 11.5
- `Model`—Provides access to the database model that EF Core uses when connecting to or creating a database. Section 11.6.2 covers this topic.

11.2 Understanding how EF Core tracks changes

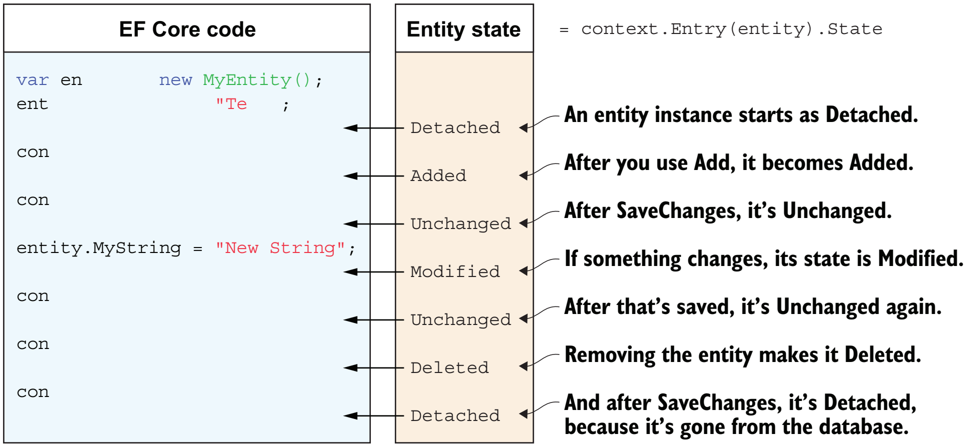
EF Core uses a property called `State` that's attached to all tracked entities. The `State` property holds the information about what you want to happen to that entity when you call the application's DbContext method, `SaveChanges`.

> **DEFINITION** As you may remember from chapter 2, *tracked entities* are entity instances that have been read in from the database using a query that didn't include the `AsNoTracking` method. Alternatively, after an entity instance has been used as a parameter to EF Core methods—such as the `Add`, `Update`, `Remove`, or `Attach` method—it becomes tracked.

This `State` property, an `enum` of type `EntityState`, is normally set by the change tracking feature inside EF Core, and in this section, you're going to explore all the ways the `State` can be set. Chapter 3 gave you a brief introduction to `State` but skipped many of its features, especially those related to relationships, as well as extra commands, which this section covers. The following list, repeated from chapter 3, lists possible values of the `State` property, which is accessed via the EF command `context.Entry(myEntity).State`:

- `Added`—The entity doesn't yet exist in the database. `SaveChanges` will insert it.
- `Unchanged`—The entity exists in the database and hasn't been modified on the client. `SaveChanges` will ignore it.
- `Modified`—The entity exists in the database and has been modified on the client. `SaveChanges` will update it.
- `Deleted`—The entity exists in the database but should be deleted. `SaveChanges` will delete it.
- `Detached`—The entity you provided isn't tracked. `SaveChanges` doesn't see it.

Figure 11.1 shows the change of `State` of the entity instance, without any relationships, as it's added, modified, and deleted from the database. The figure is a good overview of the values that the `State` of an entity can have.



```
var en        new MyEntity();
ent                "Te    ;

con

con

entity.MyString = "New String";

con

con

con
```

An entity instance starts as Detached.

After you use Add, it becomes Added.

After SaveChanges, it's Unchanged.

If something changes, its state is Modified.

After that's saved, it's Unchanged again.

Removing the entity makes it Deleted.

And after SaveChanges, it's Detached, because it's gone from the database.

Figure 11.1 The code on the left uses all the standard ways of creating, updating, and deleting data in a database. The right column shows the EF Core state of the entity as it moves through each of these stages.

When you have an entity in the `Modified` state, another per-property `boolean` flag, `IsModified`, comes into play. This flag identifies which of the properties, both scalar and navigational, have changed in the entity. This `IsModified` property for a scalar property is accessed via

```
context.Entry(entity).Property("PropertyName").IsModified,
```

and the `IsModified` property for navigational properties is accessed via

```
context.Entry(entity).Navigation("PropertyName").IsModified
```

These two ways of setting the `IsModified` property provide a per property/backing field/shadow property flag to define what has changed if the entity's `State` is set to `Modified`.

11.3 Looking at commands that change an entity's State

Figure 11.1 covers a simple entity, but when relationships are involved, the `State` settings get more complex. The following subsections present the commands that can change the `State` of an entity and its relationships.

EF Core's approach to setting the `State` of an entity class has been finely tuned, based on feedback from the previous versions of EF (EF6.x and EF Core 1.x), to set the `State` of related entities to the most "natural" `State` setting based on certain criteria, especially when you are adding/updating an entity with relationships. To give you an example, if you use the `Add` method to add a new entity with relationships to the database, EF Core will decide whether any relationship entities' `State` should be set to `Added` or `Modified`, depending on whether EF Core is tracking the entity. Generally, this decision is the right one for most `Add` calls, but knowing how EF Core decides how to set the `State` helps you when your needs fall outside normal use.

> **EF6** The setting for the `State` of an entity in EF Core differs from how EF6.x would set the `State` when you use methods such as `Add` and `Remove`. This chapter describes how EF Core sets the `State` of an entity. If you are interested in the changes from EF6.x, I recommend that you read this thread on the EF Core Git issues site: http://mng.bz/YA8A.

To start this section on an entity's `State`, table 11.1 lists the commands/actions that change an entity's `State`.

Table 11.1 All the EF Core commands/actions that can change a tracked entity's `State`, showing an example of each command/action and the final tracking `State` of the entity

Command/action	Example	Final `State`
`Add/AddRange`	`context.Add(entity);`	`Added`
`Remove/RemoveRange`	`context.Remove(entity);`	`Deleted`
`Changing` a `property`	`entity.MyString = "hello";`	`Modified`
`Update/UpdateRange`	`context.Update(entity);`	`Modified`
`Attach/AttachRange`	`context.Attach(entity);`	`Unchanged`
Setting `State` directly	`context.Entry(entity).State = …`	Given `State`
Setting `State` via `TrackGraph`	`context.ChangeTracker.TrackGraph(…`	Given `State`

NOTE The `SaveChange`/`SaveChangeAsync` methods change the `State` of all the tracked entity classes to `Unchanged`. This topic is covered in section 11.4.

The table shows what happens for a single entity class with no relationships, but most of the commands also use a recursive search of any navigational properties to find any reachable entity classes. Any command that does a recursive search will track each reachable relational entity class and set its `State`.

You have already encountered most of these commands/actions, but a few commands, such as `Attach` and `TrackGraph`, haven't been covered so far. In this section, you visit each command/action. If the command/action has already been described, the section is short. New commands/actions are covered in more detail.

11.3.1 The Add command: Inserting a new row into the database

The `Add`/`AddRange` methods are used to create a new entity in the database by setting the given entity's `State` to `Added`. Section 3.2 covers the `Add` method, and section 6.2.2 has a detailed, step-by-step look at adding an entity class with relationships. To summarize:

- The entity's `State` is set to `Added`.
- The `Add` method looks at all the entities linked to the added entity.
  - If a relationship isn't currently tracked, it is tracked, and its `State` is set to `Added`.
  - If a relationship is tracked, its current `State` is used unless there was a requirement to alter/set a foreign key, in which case its `State` is set to `Modified`.

Also, the `AddAsync`/`AddRangeAsync` methods are available for entities that use a value generator (see section 10.3.3) to set a property. If the value generator has a `NextAsync` method, you must use the `AddAsync`/`AddRangeAsync` methods when that entity is added.

11.3.2 The Remove method: Deleting a row from the database

The `Remove`/`RemoveRange` methods delete the entity from the database by setting the given entity's `State` to `Deleted`. Section 3.5 covered the `Remove` method, and section 8.8.1 covers the different delete behaviors that EF Core supports. In this section, we are looking only at what happens to the `State` of the entity class you delete and the `State` of any of its relationships. If the removed entity has any relationships that are loaded/tracked, the value of the `State` for each relationship entities will be one of the following:

- `State == Deleted`—Typical for a required dependent relationship, such as a `Review` entity class linked to a `Book` entity class
- `State == Modified`—Typical for an optional dependent relationship in which the foreign key is nullable. In this case, the optional relationship is not deleted, but the foreign key that links to the entity that was deleted is set to `null`.
- `State == Unchanged`—Result of deleting a dependent entity class that is linked to a principal class. Nothing changes in the principal class keys/foreign keys when a dependent entity class is deleted.

NOTE You can get some odd `State` settings if you read in an entity class, add a required dependent relationship, and then delete the entity class. For a short time, the required dependent relationship will have a `State` of `Added` because it's the most logical `State` at that time.

But regardless of the `State` of relationships loaded with the entity class you deleted, another stage takes precedence: the `OnDelete` behavior of the deleted entity class. If the `OnDelete` behavior is set to `Cascade`, which is the default for a required dependent relationship, it will delete any required dependent relationships of the deleted entity class. Please see section 8.8.1 for a more detailed explanation.

11.3.3 Modifying an entity class by changing the data in that entity class

One clever thing that EF Core can do is automatically detect that you changed the data in an entity class and turn that change into an update of the database. This feature makes updates simple from the developer's point of view, but it requires quite a bit of work on EF Core's part. The rules are

- For EF Core to detect a change, the entity must be tracked. Entities are tracked if you read them in without an `AsNoTracking` method in the query or when you call a `Add`, `Update`, `Remove`, or `Attach` method with an entity class as a parameter.
- When you call `SaveChanges`/`SaveChangesAsync`, by default, EF Code executes a method called `ChangeTracker.DetectChanges`, which compares the current entity's data with the entity's tracking snapshot. If any properties, backing fields, or shadow properties are different, the entity's `State` is set to `Modified`, and the properties, backing fields, or shadow properties are set to `IsModified`.

Figure 11.2 gives you an idea of how EF Core can detect a change. In this example, the only change is to one of the properties in the first `Book`.

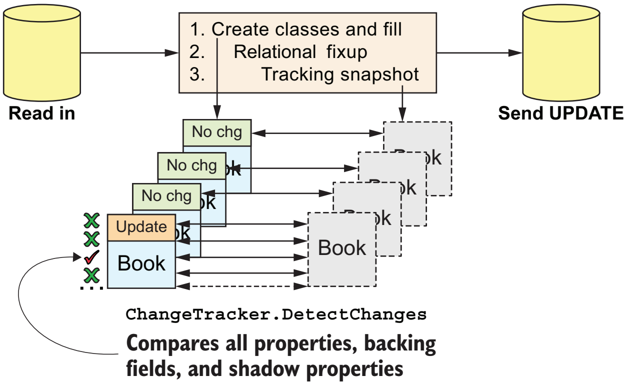


Figure 11.2 `SaveChanges` calls `ChangeTracker.DetectChanges`, which compares each tracked entity with its matching tracking snapshot to detect any differences between the two. `ChangeTracker.DetectChanges` compares all data that is mapped to the database. In this example, only one property in the first `Book` has been changed, represented in the figure by one tick and the title Update above the `Book` entity class.

11.3.4 Modifying an entity class by calling the Update method

Section 11.3.3 shows that EF Core can detect changes in an entity class for you. In chapter 3, however, you encountered an external application that returned a complete entity class in JSON form that had to be updated (see figure 11.3, which is taken from figure 3.3), but that entity class wasn't tracked. In this case, the `ChangeTracker.DetectChanges` method won't work because there is no tracking snapshot to compare. In cases like this one, you can use the `Update` and `UpdateRange` methods.

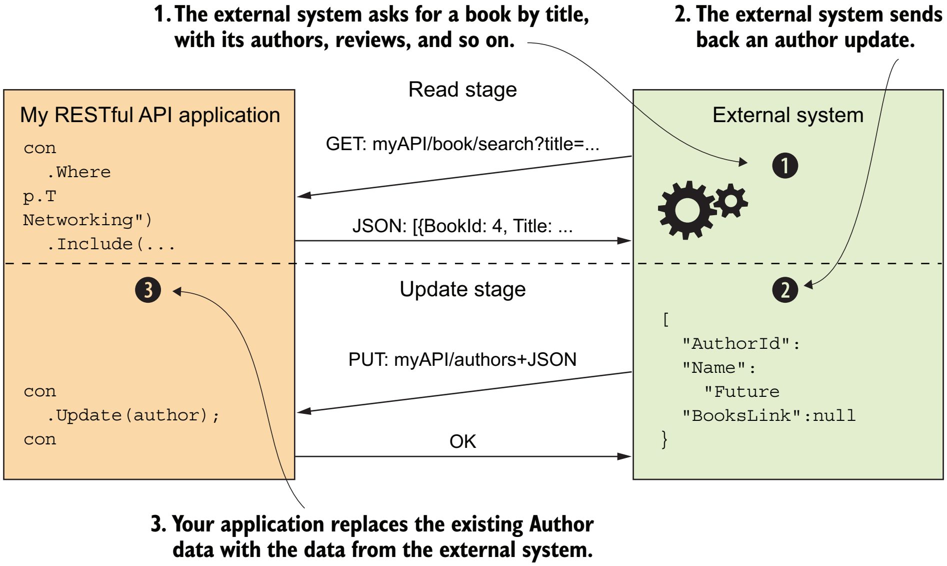


Figure 11.3 An example of a disconnected update, in which you replace all the database information with the new data. The external system on the right returns the content of the `Author` class in JSON format. The ASP.NET Core application on the left converts the send JSON back to an `Author` entity class, and the receiving code uses EF Core's `Update` command to update the Authors table in the database. The `Update` command updates all the properties, backing fields, and shadow properties in the reconstituted entity class.

The `Update` method tells EF Core to update all the properties/columns in this entity by setting the given entity's `State` to `Modified` and sets the `IsModified` property to `true` on all nonrelational properties, including any foreign keys. As a result, the row in the database will have all its columns updated.

If the entity type using the `Update` call has loaded relationships, the `Update` method will recursively look at each related entity class and set its `State`. The rules for setting the `State` on a related entity class depend on whether the relationship entity's primary key is generated by the database and is set (its value isn't the default value for the key's .NET type):

- Database-generated key, not the default value—In this case, EF Core will assume that the relationship entity is already in the database and will set the `State` to `Modified` if a foreign key needs to be set; otherwise, the State will be `Unchanged`.
- Not database-generated key, or the key is the default value—In this case, EF Core will assume that the relationship entity is new and will set its `State` to `Added`.

All that sounds quite complicated, but EF Core generally sets the `State` to the most appropriate setting. If you add an existing entry to an entity class's relationships, for example, its `State` will be `Updated`, but if you add a new entry to an entity class's relationships, its `State` will be `Added`.

11.3.5 The Attach method: Start tracking an existing untracked entity class

The `Attach` and `AttachRange` methods are useful if you have an entity class with existing valid data and want it to be tracked. After you attach the entity, it's tracked, and EF Core assumes that its content matches the current database state. This behavior works well for reconstituting entities with relationships that have been serialized and then deserialized to an entity, but only if the entities are written back to the same database, as the primary and foreign keys need to match.

> **WARNING** Serializing and then deserializing an entity class instance that uses shadow properties needs special handling with the `Attach` method. The shadow properties aren't part of the class, so they'll be lost in any serialization. Therefore, you must save/restore any shadow properties, especially foreign keys, after the `Attach` method has been called.

When you `Attach` an entity, it becomes a normal tracked entity, without the cost of loading it from the database. The `Attach` method does this by setting the entity's `State` to `Unchanged`. As with the `Update` method, what happens to the relationships of the updated entity depends on whether the relationship entity's primary key is generated by the database and is set (its value isn't the default value for the key's .NET type):

- *Database-generated key, and key has a default value*—EF Core will assume that the relationship entity is already in the database and will set the `State` to `Added`.
- *Not a database-generated key, or the key is the not default value*—EF Core will assume that the relationship entity is new and will set its `State` to `Unchanged`.

If you are unsure whether to use `Attach` or `Update` in your code, I recommend you read Arthur Vickers's article "Make sure to call Update when it is needed!" (http://mng.bz/G68O).

11.3.6 Setting the State of an entity directly

Another way to set the `State` of an entity is to set it manually to whatever state you want. This direct setting of an entity's `State` is useful when an entity has many relationships, and you need to specifically decide which state you want each relationship to have. Section 11.3.7 shows a good example.

Because the entity's `State` is read/write, you can set it. In the following code snippet, the `myEntity` instance's `State` is set to Added:

```
context.Entry(myEntity).State = EntityState.Added;
```

You can also set the `IsModified` flag on the property in an entity. The following code snippet sets the `MyString` property's `IsModified` flag to `true`, which sets the entity's `State` to `Modified`:

```
var entity = new MyEntity();
context.Entry(entity).Property("MyString").IsModified = true;
```

> **NOTE** If the entity wasn't tracked before you set the `State`, it'll be tracked afterward.

11.3.7 TrackGraph: Handling disconnected updates with relationships

The `TrackGraph` method is useful if you have an untracked entity with relationships, and you need to set the correct `State` for each entity. The `TrackGraph` method will traverse all the relational links in the entity, calling an action you supplied on each entity it finds. This method is useful if you have a group of linked entities coming from a disconnected situation (say, via some form of serialization), and you want to change only part of the data you've loaded.

> **EF6** The `TrackGraph` method is a welcome addition to EF Core. There's no equivalent command in EF6.x.

Let's expand on the simple example of a RESTful API in chapter 3, in which an author's `Name` property was updated. In that case, the external system sent back only the `Author` entity data. In this example, the external system will send back the whole book, with all its relationships, but it still wants you to update only the author's `Name` property in every `Author` entity class in the relationship.

Listing 11.1 shows the code you'd need to traverse a `Book` entity instance, which you've reconstituted from a JSON copy (not a tracked entity). The `TrackGraph` method will call your lambda `Action` method, given as the second parameter, for every entity, starting with the `Book` entity instance; then it will work through all the relational navigational property's entity instances it can reach.

Listing 11.1 Using `TrackGraph` to set each entity's `State` and `IsModified` flags

Expects an untracked book with its relationships

Calls ChangeTracker.TrackGraph, which takes an entity instance and an Action, which, in this case, you define via a lambda. The Action method is called once on each entity in the graph of entities.

```
var book = … untracked book with all relationships
context.ChangeTracker.TrackGraph(book, e =>
{
    e.Entry.State = EntityState.Unchanged;
```

If the method sets the state to any value other than Detached, the entity will become tracked by EF Core.

```
    if (e.Entry.Entity is Author)
    {
        e.Entry.Property("Name").IsModified = true;
    }
});
context.SaveChanges();
```

Here, you want to set only the Name property of the Author entity to Modified, so you check whether the entity is of type Author.

Sets the IsModified flag on the Name property; also sets the State of the entity to Modified

Calls SaveChanges, which finds that only the Name property of the Author entity has been marked as changed; creates the optimal SQL to update the Name column in the Authors table

`TrackGraph` traverses the entity provided as its first parameter and any entities that are reachable by traversing its navigation properties. The traversal is recursive, so the navigation properties of any discovered entities will also be scanned. The `Action` method you provide as the second parameter is called for each discovered untracked (`State` == `Detached`) entity and can set the `State` that each entity should be tracked in. If the visited entity's `State` isn't set, the entity remains in the `State` of `Detached` (that is, the entity isn't being tracked by EF Core). Also, `TrackGraph` will ignore any entities it visits that are currently being tracked.

Although you could still use the `Update` command for this purpose, doing so would be inefficient because the command would update every table and column in the book's relationships instead of only the authors' names. EF Core's `ChangeTracker.TrackGraph` method provides a better approach.

Figure 11.4 shows the "change only the `Author's Name`" example with an external system returning a serialized version of a `Book` entity. Using `TrackGraph` allows you to target the specific entity and property you want to set the `State` to a new value; in this case, you set the property called `Name` to `IsModified` in any `Author` entity class in the relationships of the `Book` entity.

The result of running this code is that only the `Author` entity instance's `State` is set to `Modified`, whereas the `State` of all the other entity types is set to `Unchanged`. In addition, the `IsModified` flag is set only on the `Author` entity class's `Name` property. In this example, the difference between using an `Updated` method and using the `TrackGraph` code reduces the number of database updates: the `Updated` method would produce 20 column updates (19 of them needlessly), whereas the `TrackGraph` code would change only one column.

11.4 SaveChanges and its use of ChangeTracker.DetectChanges

Section 11.3 was about setting the `State` of the tracked entities so that when you call the `SaveChanges` (or `SaveChangesAsync`) method, the correct updates are applied to the database. In this section, you look at

- How `SaveChanges` finds any updates by using the `ChangeTracker.DetectChanges` method
- What to do if `ChangeTracker.DetectChanges` is taking too long
- How to use the `State` of each tracked entity to log any changes
- How to tap into EF Core's `StateChanged` events

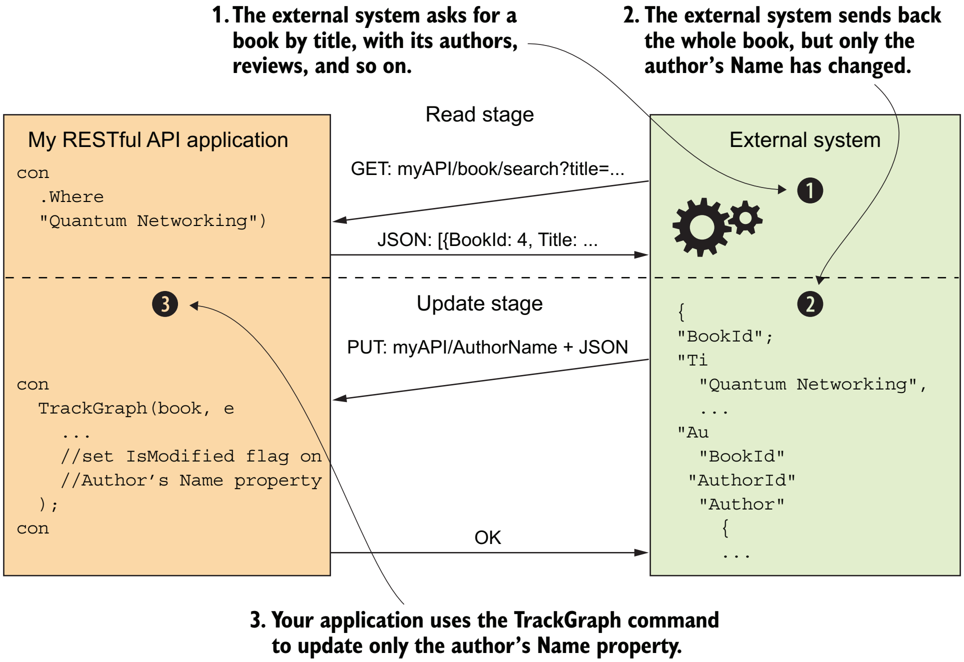


Figure 11.4 An external system that asks for a specific book and gets the JSON containing the book and all its relationships. When the external system wants to update each author's name, it sends back *all* the original JSON, with the changed names, but tells your application that it needs to change only the author's name. Your application uses EF Core's `ChangeTracker.TrackGraph` method to set all the classes to `State Unchanged` but sets the `IsModified flag` on the `Name` property in the `Author` entity class.

11.4.1 How SaveChanges finds all the State changes

Whereas states such as `Added` and `Deleted` are set by the EF Core commands, the "change a property" approach (section 11.3.3) to updates relies on code to compare each entity class with its tracking snapshot. To do so, `SaveChanges` calls a method called `DetectChanges` that is accessed via the `ChangeTracker` property.

Figure 11.5 (repeated from section 11.3.3) shows an example in which four `Book` entities have been read in and one property, the `PublishedOn` property, was changed in the first `Book` entity instance.

This process makes updates easy for you, the developer; you update only the property, backing field, or shadow property, and the change will be detected. But if you have a lot of entities with lots of data, the process can become slow. Section 11.4.2 shows you a solution to use when `ChangeTracker.DetectChanges` is taking too long.

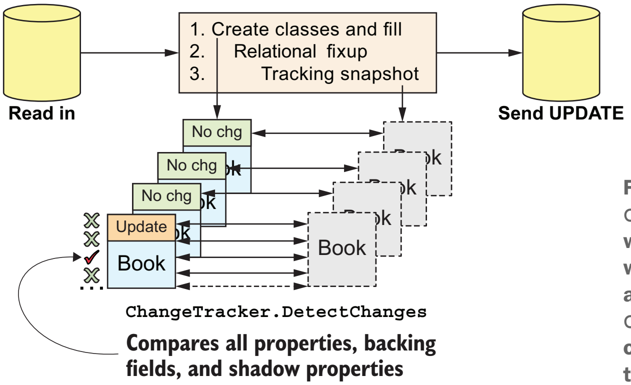


Figure 11.5 `SaveChanges` calls `ChangeTracker.DetectChanges`, which compares each tracked entity with its tracking snapshot to detect any differences between the two. `ChangeTracker.DetectChanges` compares all data that is mapped to the database.

11.4.2 What to do if ChangeTracker.DetectChanges is taking too long

In some applications, you may have a large number of tracked entities loaded. When you're executing mathematical modeling or building artificial intelligence applications, for example, holding a lot of data in memory may be the only way to achieve the level of performance that you require.

The problem is if you have a large amount of tracked entity instances and/or your entities have a lot of data in them. In that case, a call to `SaveChanges/SaveChangesAsync` can become slow. If you are saving a lot of data, the slowness is most likely caused by the database accesses. But if you are saving only a small amount of data, any slowdown is likely due to the time the `ChangeTracker.DetectChanges` takes to compare each entity class instance with its matching tracking snapshot.

EF Core offers you a few ways to replace `ChangeTracker.DetectChanges` with an alternative way to detect changes. These features work by detecting individual updates to the data in your entity classes, cutting out any comparisons of data that hasn't been changed. A rather unscientific test of saving 100,000 tiny entities that had no changes took 350 ms with the normal `ChangeTracker.DetectChanges` method, for example, whereas the approach that detects changes via the class took 2 ms for the same data.

You have four ways to replace the `ChangeTracker.DetectChanges;` each approach has different features and different levels of effort to implement. Table 11.2 summarizes these approaches, with their pros and cons.

Table 11.2 A comparison of the four approaches you can use to stop the `ChangeTracker.DetectChanges` method from looking at an entity, thus saving time

What	Pros	Cons
`INotifyPropertyChanged`	- Can change only the entities that are slow - Handles concurrency exceptions	- Need to edit every property

Table 11.2 A comparison of the four approaches you can use to stop the `ChangeTracker.DetectChanges` method from looking at an entity, thus saving time *(continued)*

What	Pros	Cons
`INotifyPropertyChanged` and `INotifyPropertyChanging`	▪ Can change only the entities that are slow ▪ No tracking snapshot, so uses less memory	▪ Need to edit every property
Proxy change tracking (EF Core 5 feature) `INotifyPropertyChanged`	▪ Easy to code; add virtual to every property ▪ Handles concurrency exceptions	▪ Must change *all* entity types to use proxy
Proxy change tracking (EF Core 5 feature) `INotifyPropertyChanged` and `INotifyPropertyChanging`	▪ Easy to code; add virtual to every property ▪ No tracking snapshot, so uses less memory	▪ Must change *all* entity types to use proxy ▪ Have to create a new entity class via the `CreateProxy<T>` method

Overall, the proxy change tracking feature is easier to code but requires you to change all your entity classes to use proxy change tracking. But if you find a `SaveChanges` performance issue in an existing application, changing all your entity classes might be too much work. For this reason, I focus on the first approach, `INotifyPropertyChanged`, which is easy to add to a few entity classes that have a problem, and the last approach, proxy changed/changing tracking, which is easier but requires you to use it across the whole application.

FIRST APPROACH: INOTIFYPROPERTYCHANGED

EF Core supports the `INotifyPropertyChanged` interface on an entity class to detect whether any property has changed. This interface notifies EF Core that a property has changed, but you have to raise a `PropertyChanged` event, which means the `ChangeTracker.DetectChanges` method isn't used.

To use the `INotifyPropertyChanged` interface you need to create a `NotificationEntity` helper class, shown in the following listing. This class provides a `SetWithNotify` method that you call when any property in your entity class changes.

Listing 11.2 `NotificationEntity` helper class that `NotifyEntity` inherits

```
public class NotificationEntity : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void SetWithNotify<T>(T value, ref T field,
        [CallerMemberName] string propertyName = "")
    {
        if (!Object.Equals(field, value))
        {
            field = value;
```

Automatically gets the propertyName by using System.Runtime.CompilerServices

Only if the field and the value are different do you set the field and raise the event.

Sets the field to the new value

```
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Invokes the PropertyChanged event, but using ?. to stop the method from failing when the new entity is created and the PropertyChangedEventHandler hasn't been filled in by EF Core…

… with the name of the property

The following listing shows an entity class called `NotifyEntity`, which inherits the `NotificationEntity` shown in listing 11.2. You must call the `SetWithNotify` method whenever a noncollection property changes. For collections, you have to use a `ObservableCollection` to raise an event when a navigational collection property is changed.

Listing 11.3 `NotifyEntity` using `NotificationEntity` class for events

```
public class NotifyEntity : NotificationEntity
{
    private int _id;
    private string _myString;
    private NotifyOne _oneToOne;
```

Each noncollection property must have a backing field.

```
    public int Id
    {
        get => _id;
        set => SetWithNotify(value, ref _id);
    }

    public string MyString
    {
        get => _myString;
        set => SetWithNotify(value, ref _myString);
    }

    public NotifyOne OneToOne
    {
        get => _oneToOne;
        set => SetWithNotify(value, ref _oneToOne);
    }
```

If a noncollection property is changed, you need to raise a PropertyChanged event, which you do via the inherited method SetWithNotify.

```
    public ObservableCollection<NotifyMany>
        Collection { get; }
        = new ObservableCollection<NotifyMany>();
}
```

Any collection navigational property must be an Observable collection, so you need to predefine that Observable collection.

You can use any Observable collection, but for performance reasons, EF Core prefers ObservableHashSet<T>.

After you've defined your entity class to use the `INotifyPropertyChanged` interface, you must configure the tracking strategy for this entity class to `ChangedNotifications` (listing 11.4). This configuration tells EF Core not to detect changes via `ChangeTracker.DetectChanges` because it will be notified of any changes via

`INotifyPropertyChanged` events. To configure `INotifyPropertyChanged` events for one entity class, you use the Fluent API command.

Listing 11.4 Setting the tracking strategy for one entity to `ChangedNotifications`

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
      .Entity<NotifyEntity>()
      .HasChangeTrackingStrategy(
          ChangeTrackingStrategy.ChangedNotifications);
}
```

APPROACHES 2 AND 3

I am not covering approach 2 (change and changing events), but the differences from approach 1 are

- The `NotificationEntity` class must create change and changing events.
- You use a different `ChangeTrackingStrategy` setting, such as `ChangingAndChangedNotifications`.

Also not covered is approach 3 (proxy change tracking, `INotifyPropertyChanged`), which works in a similar way to how lazy loading proxy works with virtual properties. Instead, I cover the last approach (described next), which handles both `INotifyPropertyChanged` and `INotifyPropertyChanging`. The main difference is that in approach 3, you can create an instance of an entity class by using the normal constructor approach, whereas the last approach requires you to use the `CreateProxy<TEntity>` method to create an entity class.

LAST APPROACH: PROXY CHANGE TRACKING

The last approach uses proxy change tracking via the `INotifyPropertyChanged` and `INotifyPropertyChanging` events introduced in EF Core 5. These change-tracking events are added to the lazy-loading proxy approach with the virtual properties described in section 2.4.4. To use this approach, you need to do five things:

- Change all your entity classes to have virtual properties.
- Use an `Observable` collection type for navigational collection properties.
- Change your code that creates a new instance of an entity class to use the `CreateProxy<TEntity>` method.
- Add the NuGet library `Microsoft.EntityFrameworkCore.Proxies`.
- Add the method `UseChangeTrackingProxies` when building the application's DbContext options.

Let's start by looking at the structure of the entity class you need to use the proxy change tracking approach, as shown in the following listing.

Listing 11.5 An example entity class set up to use proxy change tracking

```
public class ProxyMyEntity
{
    public virtual int Id { get; set; }
    public virtual string MyString { get; set; }
    public virtual ProxyOptional ProxyOptional { get; set; }

    public virtual ObservableCollection<ProxyMany>
        Many { get; set; }
        = new ObservableCollection<ProxyMany>();
}
```

All properties must be virtual.

For navigational collection properties, you need to use an Observable collection type.

If you read in an entity class via a query, the proxy change tracking will add its extra code to create the `INotifyPropertyChanged` and `INotifyPropertyChanging` events when a property is changed. But if you want to create a new entity class, you can't use the normal `new` command, such as `new Book()`. Instead, you must use the `CreateProxy<TEntity>` method. If you wanted to add a new version of the `ProxyMyEntity` class shown in listing 11.5, for example, you would write

```
var entity = context.CreateProxy<ProxyMyEntity>();
entity.MyString = "hello";
context.Add(entity);
context.SaveChanges();
```

You must use the `CreateProxy<TEntity>` method (first line of the preceding code snippet); otherwise, EF Core won't be able to detect the changing event. (Don't worry; if you forget, EF Core throws an exception with a useful message.)

The final part is making sure that the `Microsoft.EntityFrameworkCore.Proxies` NuGet package is loaded and then updating your DbContext configuration to include the `UseChangeTrackingProxies` method, as shown in the following code snippet:

```
var optionsBuilder =
    new DbContextOptionsBuilder<EfCoreContext>();
optionsBuilder
    .UseChangeTrackingProxies()
    .UseSqlServer(connection);
var options = optionsBuilder.Options;

using (var context = new EfCoreContext(options))
```

NOTE For the third approach, you can turn off the `INotifyPropertyChanging` part of the proxy change tracking by setting the first parameter, `useChangeTrackingProxies`, in the `UseChangeTrackingProxies` method to `false`. Then EF Core would start using the tracking snapshot for comparison.

11.4.3 Using the entities' State within the SaveChanges method

So far, you've learned how to set the `State` of an entity and heard about how `ChangeTracker` can be used to find out what has changed. Now you are going to use the `State` data within the `SaveChanges`/`SaveChangesAsync` to do some interesting things. Here are some of the possible uses of detecting what's about to be changed in the database:

- Automatically adding extra information to an entity—for instance, adding the time when an entity was added or updated
- Writing a history audit trail to the database each time a specific entity type is changed
- Add security checks to see whether the current user is allowed to update that particular entity type

The basic approach is to override the `SaveChanges`/`SaveChangesAsync` methods inside your application's DbContext and execute a method before the base `SaveChanges`/`SaveChangesAsync` is called. We check the `States` before the base `SaveChanges` is called because a) the `State` of every tracked entity will have a value of `Unchanged` once `SaveChanges` is called and b) you want to add/alter some of the entities before they are written to the database. What you do with the `State` information is up to you, but next is an example that logs the last time the entity was added or updated, with the `UserId` of the user who did the add/update.

The following listing provides an interface you can add to any entity class. This defines the properties that you want filled in when the entity is added or updated, and a method that can be used to set the properties to the right values.

Listing 11.6 The `ICreatedUpdated` interface defining four properties and a method

```
public interface ICreatedUpdated
{
    DateTime WhenCreatedUtc { get; }
    Guid CreatedBy { get; }
    DateTime LastUpdatedUtc { get; }
    Guid LastUpdatedBy { get; }

    void LogChange(EntityState state, Guid userId = default);
}
```

Add this interface to any entity class where you want to log when/who it was created or updated.

Holds the datetime when the entity was first added to the database

Holds the UserId who created the entity

Holds the datetime when the entity was last updated

Holds the UserId who last updated the entity

Called when the entity's state is Added or Modified State. Its job is to update the properties based on the state.

The following listing shows an entity class called `CreatedUpdatedInfo` that implements the `ICreatedUpdated` interface that you'll detect when your modified `SaveChanges` method is called (see listing 11.8). The `LogChange` method, which you'll call in your modified `SaveChanges` method, sets the various properties in the entity class.

Listing 11.7 Automatically setting who and when a entity was updated

```
public class CreatedUpdatedInfo : ICreatedUpdated
   {
    public DateTime WhenCreatedUtc { get; private set; }
    public Guid CreatedBy { get; private set; }
    public DateTime LastUpdatedUtc { get; private set; }
    public Guid LastUpdatedBy { get; private set; }

    public void LogChange(EntityEntry entry,
        Guid userId = default)
    {
        if (entry.State != EntityState.Added &&
            entry.State != EntityState.Modified)
            return;

        var timeNow = DateTime.UtcNow;
        LastUpdatedUtc = timeNow;
        LastUpdatedBy = userId;
        if (entry.State == EntityState.Added)
        {
            WhenCreatedUtc = timeNow;
            CreatedBy = userId;
        }
        else
        {
            entry.Property(
                nameof(ICreatedUpdated.LastUpdatedUtc))
                .IsModified = true;
            entry.Property(
                nameof(ICreatedUpdated.LastUpdatedBy))
                .IsModified = true;
        }
    }
}
```

Entity class inherits ICreatedUpdated, which means any addition/update of the entity is logged.

These properties have private setters so that only the LogChange method can change them.

Its job is to update the created and updated properties. It is passed the UserId if available.

This method only handles Added or Modified States.

Obtains the current time so that an add and update time will be the same on create

It always sets the LastUpdatedUtc and LastUpdatedBy.

If it's an add, then you update the WhenCreatedUtc and the CreatedBy properties.

For performance reasons you turned off DetectChanges, so you must manually mark the properties as modified.

The next step is to override all versions of the `SaveChanges` method inside your application's DbContext and then precede the call to the base `SaveChanges` with a call to your `AddUpdateChecks` method shown in listing 11.8. This method looks for entities with a `State` of `Added` or `Modified` and inherits the `ICreatedUpdated` interface. If the method finds an entity (or entities) that fits that criteria, it calls the entity's `LogChange` method to set the two properties to the correct values.

The following listing shows your application's DbContext, called `Chapter11DbContext`, which implements that code. (To keep the code shorter, you only override the `SaveChanges` method. Normally, you'd also override the `SaveChangesAsync` method with two parameters.) Notice too that the code ensures the `ChangeTracker.DetectChanges` method is only called once, because, as you have seen, that method can take some time.

Listing 11.8 Your DbContext looks for added or modified `ICreatedUpdated` entities

```
private void AddUpdateChecks()
{
    ChangeTracker.DetectChanges();
    foreach (var entity in ChangeTracker.Entries()
        .Where(e =>
            e.State == EntityState.Added ||
            e.State == EntityState.Modified))
    {
        var tracked = entity.Entity as ICreatedUpdated;
        tracked?.LogChange(entity);
    }
}

public override int SaveChanges(bool acceptAllChangesOnSuccess)
{
    AddUpdateChecks();
    try
    {
        ChangeTracker.AutoDetectChangesEnabled = false;
        return base.SaveChanges(acceptAllChangesOnSuccess);
    }
    finally
    {
        ChangeTracker.AutoDetectChangesEnabled = true;
    }
}
```

This private method will be called from SaveChanges and SaveChangesAsync.

It calls DetectChanges to make sure all the updates have been found.

It loops through all the tracked entities that have a State of Added or Modified.

If the Added/Modified entity has the ICreatedUpdated, then the tracked isn't null.

So we call the LogChange command. In this example we don't have the UserId available.

You override SaveChanges (and SaveChangesAsync—not shown).

You call the AddUpdateChecks, which contains a call to ChangeTracker.DetectChanges().

Because DetectChanges has been called we tell SaveChanges not to call it again (for performance reasons).

You call the base.SaveChanges that you overrode

Finally to turn the AutoDetectChangesEnabled back on

This is only one example of using `ChangeTracker` to take actions based on the `State` of tracked entities, but it establishes the general approach. The possibilities are endless.

> **NOTE** In chapter 16 I have another example of detecting the `State` of certain entities to update a separate database when a `Book` or its related entities change.

11.4.4 *Catching entity class's State changes via events*

EF Core 2.1 added two events to EF Core: `ChangeTracker.Tracked`, which is triggered when an entity is first tracked, and `ChangeTracker.StateChanged`, which is triggered when the `State` of an already tracked entity is changed. This feature provides a similar effect to calling `ChangeTracker.Entries()`, but by producing an event when something changes. The `ChangeTracker` events are useful for features such as logging changes or triggering actions when a specific entity type's `State` changes. But to start, let's look at the basics of these two events.

The `Tracked` event, which is simpler, is triggered when an entity class is first tracked and tells you whether it came from a query via its `FromQuery` property. That event could occur when you execute an EF Core query (without the `AsNoTracking` method) or start tracking an entity class via an `Add` or `Attach` method. The following listing is a unit test that captures a `Tracked` event when an entity class is `Added` to the context.

Listing 11.9 Example of a `ChangeTracker.Tracked` event and what it contains

```
var logs = new List<EntityTrackedEventArgs>();
context.ChangeTracker.Tracked += delegate(
    object sender, EntityTrackedEventArgs args)
{
    logs.Add(args);
};

//ATTEMPT
var entity = new MyEntity {MyString = "Test"};
context.Add(entity);

//VERIFY
logs.Count.ShouldEqual(1);
logs.Single().FromQuery.ShouldBeFalse();
logs.Single().Entry.Entity.ShouldEqual(entity);
logs.Single().Entry.State
    .ShouldEqual(EntityState.Added);
```

Holds a log of any tracked events

You register your event handler to the ChangeTracker.Tracked event.

This event handler simply logs the EntityTrackedEventArgs.

Creates an entity class

Adds that entity class to context

There is one event.

The entity wasn't tracking during a query.

You can access the entity that triggered the event.

You can also get the current State of that entity.

This listing shows you what information is available in the event data. For a `Tracked` event, you get the `FromQuery` flag, which is `true` if the query was tracked during a query. The `Entry` property gives you information about the entity.

One thing to note in this example is that the `context.Add(entity)` method triggers an `Tracked` event but doesn't trigger a `StateChanges` event. If you want to detect a newly added entity class, you can do so only via the `Tracked` event.

The `StateChanges` event is similar but contains different information. The following listing captures the `StateChanges` event when `SaveChanges` is called. The event contains the entity's `State` before `SaveChanges` was called in the property called `OldState` and the entity's `State` after `SaveChanges` was called in the property called `NewState`.

Listing 11.10 Example of a `ChangeTracker.StateChanges` event and what it contains

Holds a log of any StateChanged events

```
var logs = new List<EntityStateChangedEventArgs>();
context.ChangeTracker.StateChanged += delegate
    (object sender, EntityStateChangedEventArgs args)
{
    logs.Add(args);
};
```

You register your event handler to the ChangeTracker .StateChanged event.

This event handler simply logs the EntityTrackedEventArgs.

```
//ATTEMPT
var entity = new MyEntity { MyString = "Test" };
context.Add(entity);
context.SaveChanges();

//VERIFY
logs.Count.ShouldEqual(1);
logs.Single().OldState.ShouldEqual(EntityState.Added);
logs.Single().NewState.ShouldEqual(EntityState.Unchanged);
logs.Single().Entry.Entity.ShouldEqual(entity);
```

Creates an entity class

Adds that entity class to context

SaveChanges will change the State to Unchanged after the database update.

There is one event.

The State before the change was Added

The State after the change is Unchanged

You get access to the entity data via the Entry property.

The listing shows that you get the before and after `States` of the entity by using the `OldState` and `NewState` properties, respectively. Now that you have seen the two `ChangeTracker` events, let's use them for logging changes to some other form of storage. But in the following listing, I show a class that will turn the two `ChangeTracker` events into logs via NET's `ILogger` interface.

Listing 11.11 Class holding the code to turn `ChangeTracker` events into logs

```
public class ChangeTrackerEventHandler
{
    private readonly ILogger _logger;

    public ChangeTrackerEventHandler(DbContext context,
        ILogger logger)
    {
        _logger = logger;
        context.ChangeTracker.Tracked += TrackedHandler;
        context.ChangeTracker.StateChanged += StateChangeHandler;
    }

    private void TrackedHandler(object sender,
        EntityTrackedEventArgs args)
    {
        if (args.FromQuery)
            return;

        var message = $"Entity: {NameAndPk(args.Entry)}. " +
            $"Was {args.Entry.State}";
        _logger.LogInformation(message);
    }

    private void StateChangeHandler(object sender,
        EntityStateChangedEventArgs args)
    {
        var message = $"Entity: {NameAndPk(args.Entry)}. " +
            $"Was {args.OldState} and went to {args.NewState}";
        _logger.LogInformation(message);
    }
}
```

This class is used in your DbContext to log changes.

You will log to ILogger.

Adds a Tracked event handler

Adds a StateChanged event handler

Handles Tracked events

We do not want to log entities that are read in.

Forms a useful message on Add or Attach

The StateChanged event handler logs any changes.

Now add this code to the constructor of your application DbContext, as shown in the following listing.

Listing 11.12 Adding the `ChangeTrackerEventHandler` to your application DbContext

```
public class Chapter11DbContext : DbContext
{
    private ChangeTrackerEventHandler _trackerEventHandler;

    public Chapter11DbContext(
        DbContextOptions<Chapter11DbContext> options,
        ILogger logger = null)
        : base(options)
    {
        if (logger != null)
            _trackerEventHandler = new
                ChangeTrackerEventHandler(this, logger);
    }
    //… rest of code left out
}
```

Your application DbContext that you want to log changes from

You need an instance of the event handler class while the DbContext exists.

You add a ILogger to the constructor.

If an ILogger is available, you register the handlers.

Creates the event handler class, which registers the event handlers

This example is a simple one, but it does show how powerful the `ChangeTracker` events are. My logged messages are rather simple (see the next listing), but you could easily expand these messages to detail what properties have been modified, include the `UserId` of the user who changed things, and so on.

Listing 11.13 Example output of `ChangeTrackerEventHandler` event logging

Code that triggered that event: context.Add(new MyEntity)

```
Entity: MyEntity {Id: -2147482647}. Was Added
Entity: MyEntity {Id: 1}. Was Added and went to Unchanged
Entity: MyEntity {Id: 1}. Was Unchanged and went to Modified
Entity: MyEntity {Id: 1}. Was Modified and went to Unchanged
```

Code that triggered that event: context.SaveChanges()

Code that triggered that event: entity.MyString = "New string" + DetectChanges

Code that triggered that event: context.SaveChanges()

11.4.5 Triggering events when SaveChanges/SaveChangesAsync is called

EF Core 5 introduced `SavingChanges`, `SavedChanges`, and `SaveChangesFailed` events, which are called before the data is saved to the database, after the data has been successfully saved to the database, and if the save to the database failed, respectively. These events allow you to tap into what is happening in the `SaveChanges` and `SaveChangesAsync` methods. You could use these events to log what was written to the database or alert someone if there was a certain exception inside `SaveChanges` or `SaveChangesAsync`.

To use these events, you need to subscribe to the `SavingChanges` and `SavedChanges` events. The following listing shows you how.

Listing 11.14 How to subscribe to the `SavingChanges/SavedChanges` events

This event will trigger when SaveChanges is called but before it updates the database.

```
context.SavingChanges +=
    delegate(object dbContext,
       SavingChangesEventArgs args)
{
    var trackedEntities =
       ((DbContext)dbContext)
       .ChangeTracker.Entries();

    //… your code goes here
};

context.SavedChanges +=
    delegate(object dbContext,
          SavedChangesEventArgs args)
{
    //… your code goes here
};

context. SaveChangesFailed+=
    delegate(object dbContext,
          SaveChangesFailedEventArgs args)
{
    //… your code goes here
};
```

The SavingChangesEventArgs contains the SaveChanges Boolean parameter acceptAllChangesOnSuccess.

The first parameter is the instance of the DbContext, but you need to cast the object to use it.

The first parameter is the instance of the DbContext that this event is linked to.

This event will trigger when SaveChanges successfully updates the database.

The SavedChangesEventArgs contains the count of entities that were saved to the database.

This event will trigger when SaveChanges has an exception during an update to the database.

The SavingChangesEventArgs contains the exception that happened during the update to the database.

To use these events, you need to know a few things about them:

- Like all C# events, the subscription to these events lasts only as long as the instance of the DbContext exists.
- The events are triggered by both the `SaveChanges` and `SaveChangesAsync` methods.
- The `SavingChanges` event is called before the `ChangeTracker.DetectChanges` method is called, so if you want to implement the code shown in section 11.4.3 to update entities by using their `State`, you need to call the `ChangeTracker.DetectChanges` method first. This approach isn't a good idea, however, because `DetectChanges` would be called twice, which could cause a performance issue.

11.4.6 EF Core interceptors

EF Core 3.0 introduced interceptors that enable you intercept, modify, and/or suppress EF Core operations, including low-level database operations, such as executing a command, as well as higher-level operations, such as calls to `SaveChanges`. These interceptors have some powerful features, such as altering commands being sent to the database.

This feature is advanced, so this section simply signposts the fact that it is available. Also, the EF Core documentation for interceptors is good, provides lots of useful examples, and is about 15 pages long. I refer you to the Microsoft documentation for more information (http://mng.bz/zGJQ).

11.5 Using SQL commands in an EF Core application

EF Core has methods that allow raw SQL commands to be used, either as part of a LINQ query or a database write, such as an SQL UPDATE. These commands are useful when the query you want to perform can't be expressed with LINQ—when it calls an SQL stored procedure, for example, or when a LINQ query results in inefficient SQL being sent to the database.

> **DEFINITION** An SQL stored procedure is a set of SQL commands—which may or may not have parameters—that can be executed. These commands typically read and/or write to the database. The set of SQL commands is stored in the database as a stored procedure and given a name. Then the stored procedure can be called as part of an SQL command.

EF Core's SQL commands are designed to detect SQL injection attacks—attacks in which a malicious user replaces, say, a primary-key value with some SQL commands that extract extra data from your database. EF Core provides two types of SQL commands:

- Methods ending in `Raw`, such as `FromSqlRaw`. In these commands, you provide separate parameters, and those parameters are checked.
- Methods ending in `Interpolated`, such as `FromSqlInterpolated`. The string parameter provided to these methods used C#6's string interpolation with the parameters in the string, such as `$"SELECT * FROM Books WHERE BookId = {myKey}"`. EF Core can check each parameter within the interpolated string type.

> **WARNING** If you build an interpolated string outside the command—such as `var badSQL = $"SELECT … WHERE BookId = {myKey}"`—and then use it in a command like `FromSqlRaw(badSQL)`, EF Core can't check SQL injection attacks. You should use `FromSqlRaw` with parameters or `FromSqlInterpolated` with parameters embedded in a string interpolation.

You can include SQL commands in EF commands in several ways. In addition to showing each group, I will use a mixture of `…Raw` and `…Interpolated` sync versions in the examples. Every command I show has an async version other than the `GetDbConnection` method. The groups of SQL commands that are covered are

- `FromSqlRaw/FromSqlInterpolated` sync/async methods, which allow you to use a raw SQL command in an EF Core query
- `ExecuteSqlRaw/ExecuteSqlInterpolated` sync/async methods, which execute a nonquery command
- `AsSqlQuery` Fluent API method, which maps an entity class to an SQL query
- `Reload/ReloadAsync` command, used to refresh an EF Core-loaded entity that has been changed by an `ExecuteSql…` method

- EF Core's `GetDbConnection` method, which provides low-level database access libraries to access the database directly

EF6 The commands in EF Core for SQL access are different from the way that EF6.x provides SQL access to the database.

11.5.1 FromSqlRaw/FromSqlInterpolated: Using SQL in an EF Core query

The `FromSqlRaw/FromSqlInterpolated` methods allow you to add raw SQL commands to a standard EF Core query, including commands that you wouldn't be able to call from EF Core, such as stored procedures. Here's an example of calling a stored procedure that returns only books that have an average review vote of the given value.

Listing 11.15 Using a `FromSqlInterpolated` method to call an SQL stored procedure

```
int filterBy = 5;
var books = context.Books
    .FromSqlInterpolated(
        $"EXECUTE dbo.FilterOnReviewRank @RankFilter = {filterBy}")
    .IgnoreQueryFilters()
    .ToList();
```

You start the query in the normal way, with the DbSet<T> you want to read.

The FromSqlInterpolated method allows you to insert an SQL command.

Uses C#6's string interpolation feature to provide the parameter

You need to remove any query filters; otherwise, the SQL won't be valid.

There are a few rules about an SQL query:

- The SQL query must return data for all properties of the entity type (but there is a way around this rule; see section 11.5.5).
- The column names in the result set must match the column names that properties are mapped to.
- The SQL query can't contain related data, but you can add the `Include` method to load related navigational properties (see listing 11.16).

You can add other EF Core commands after the SQL command, such as `Include`, `Where`, and `OrderBy`. The following listing shows an SQL command that filters the results by the average star rating with an `Include` of the book's `Reviews` and `AsNoTracking` command added.

Listing 11.16 Example of adding extra EF Core commands to the end of an SQL query

```
double minStars = 4;
var books = context.Books
    .FromSqlRaw(
       "SELECT * FROM Books b WHERE " +
         "(SELECT AVG(CAST([NumStars] AS float)) " +
         "FROM dbo.Review AS r " +
```

The SQL calculates the average votes and uses it in an SQL WHERE.

```
          "WHERE b.BookId = r.BookId) >= {0}", minStars)
    .Include(r => r.Reviews)
    .AsNoTracking()
    .ToList();
```

The Include method works with the FromSql because you are not executing a store procedure.

You can add other EF Core commands after the SQL command.

In this case, you use the normal sql parameter check and substitution method— {0}, {1}, {2}, and so on.

WARNING If you're using model-level query filters (see section 6.1.7), the SQL you can write has limitations. `ORDER BY` won't work, for example. The way around this problem is to apply the `IgnoreQueryFilters` method after the `Sql` command and re-create the model-level query filter in your SQL code.

11.5.2 ExecuteSqlRaw/ExecuteSqlInterpolated: Executing a nonquery command

In addition to putting raw SQL commands in a query, you can execute nonquery SQL commands via EF Core's `ExecuteSqlRaw`/`ExecuteSqlInterpolated` methods. Typical commands are SQL `UPDATE` and `DELETE`, but any nonquery SQL command can be called. The following listing shows an SQL `UPDATE` command, which takes two parameters.

Listing 11.17 The `ExecuteSqlCommand` method executing an SQL `UPDATE`

```
var rowsAffected = context.Database
    .ExecuteSqlRaw(
        "UPDATE Books " +
        "SET Description = {0} " +
        "WHERE BookId = {1}",
        uniqueString, bookId);
```

The ExecuteSqlRaw is in the context.Database property.

The ExecuteSqlRaw will execute the SQL and return an integer, which in this case is the number of rows updated.

The SQL command is a string, with places for the parameters to be inserted.

Provides two parameters referred to in the command

The `ExecuteSqlRaw` method returns an integer, which is useful for checking that the command was executed in the way you expected. In this example, you'd expect the method to return `1` to show that it found and updated a row in the Books table that had the primary key you provided.

11.5.3 AsSqlQuery Fluent API method: Mapping entity classes to queries

EF Core 5 provided a way to map an entity class to an SQL query via a the `AsSqlQuery` Fluent API method. This feature allows you to hide your SQL code inside the application's DbContext's configuration, and developers can use this `DbSet<T>` property in queries as though it were a normal entity class mapped to an entity. It's a read-only entity class, of course, but see the following note if you need a read/write version.

NOTE EF Core 5 added the ability to configure an entity class to be mapped to both a table (for create, update, and delete) and a view (for read). See http://mng.bz/0rY6.

As an example, you will create an entity class called `BookSqlQuery` that returns three values for a `Book` entity class: `BookId`, `Title`, and the average votes for this `Book` in a property called `AverageVotes`. This class is shown in the following listing.

Listing 11.18 The `BookSqlQuery` class to map to an SQL query

```
public class BookSqlQuery
{
    [Key]
    public int BookId { get; set; }

    public string Title { get; set; }

    public double? AverageVotes { get; set; }
}
```

The primary key of the book that is returned

The title of the book

The average votes for this Book based on the Review's NumStars property

Now you need to configure this entity class to an SQL query, using the `AsSqlQuery` Fluent API method, as shown in the following listing.

Listing 11.19 Configuring the `BookSqlQuery` entity class to an SQL query

```
public class BookDbContext : DbContext
{
    //… other DbSets removed for clarity

    public DbSet<BookSqlQuery> BookSqlQueries { get; set; }

    protected override void
        OnModelCreating(ModelBuilder modelBuilder)
    {
        //… other configrations removed for clarity

        modelBuilder.Entity<BookSqlQuery>().ToSqlQuery(
        @"SELECT BookId
            ,Title
            ,(SELECT AVG(CAST([r0].[NumStars] AS float))
        FROM Review AS r0
        WHERE t.BookId = r0.BookId)  AS AverageVotes
        FROM Books AS t");
    }
}
```

You add a DbSet<T> for the BookSqlQuery entity class to make querying easy.

The ToSqlQuery method maps the entity class to an SQL query.

Returns the three values for each Book

You can add LINQ commands, such as `Where` and `OrderBy`, in the normal way, but the returned data follows the same rules as the `FromSqlRaw` and `FromSqlInterpolated` methods (section 11.5.1).

11.5.4 Reload: Used after ExecuteSql commands

If you have an entity loading (tracked), and you use an `ExecuteSqlRaw/ExecuteSqlInterpolated` method to change the data on the database, your tracked entity is out of date. That situation could cause you a problem later, because EF Core doesn't know that the values have been changed. To fix this problem, EF Core has a method called `Reload/ReloadAsync`, which updates your entity by rereading the database.

In the following listing, you load an entity, change its content via the `ExecuteSqlCommand` method, and then use the `Reload` method to make sure that the entity's content matches what's in the database.

Listing 11.20 Using the `Reload` method to refresh the content of an existing entity

```
var entity = context.Books.
    Single(x => x.Title == "Quantum Networking");
var uniqueString = Guid.NewGuid().ToString();

context.Database.ExecuteSqlRaw(
        "UPDATE Books " +
        "SET Description = {0} " +
        "WHERE BookId = {1}",
        uniqueString, entity.BookId);

context.Entry(entity).Reload();
```

Loads a Book entity in the normal way

Uses ExecuteSqlRaw to change the Description column of that same Book entity

When calling the Reload method, EF Core rereads that entity to make sure that the local copy is up to date.

At the end of this code, the entity instance will match what's in the database.

11.5.5 GetDbConnection: Running your own SQL commands

When EF Core can't provide the query features you want, you need to drop back to another database access method that can. A few low-level database libraries require a lot more code to be written but provide more-direct access to the database, so you can do almost anything you need to do. Normally, these low-level database libraries are database-server-specific. In this section, you use a NuGet library called Dapper (see https://github.com/StackExchange/Dapper). Dapper is a simple object mapper for .NET, sometimes known as a micro-ORM. Dapper is simple but fast. It uses the ADO.NET library to access the database and adds autocopying of columns to class properties.

The following listing uses Dapper to read specific columns into a nonentity class called `RawSqlDto` that has properties called `BookId`, `Title`, and `AverageVotes`, so you can load only the columns you want. In this example, you use Dapper to query the same database that your application's DbContext is linked to. The Dapper query returns a single `RawSqlDto` class with data in the three properties for the Books row, where the `BookId` column (the primary key) has a value of `4`.

Listing 11.21 Obtaining a `DbConnection` from EF Core to run a Dapper SQL query

```
var connection = context.Database.GetDbConnection();
string query = "SELECT b.BookId, b.Title, " +
               "(SELECT AVG(CAST([NumStars] AS float)) " +
               "FROM dbo.Review AS r " +
               "WHERE b.BookId = r.BookId) AS AverageVotes " +
               "FROM Books b " +
               "WHERE b.BookId = @bookId";

var bookDto = connection
    .Query<RawSqlDto>(query, new
    {
        bookId = 4
    })
    .Single();
```

Gets a DbConnection to the database, which the micro-ORM called Dapper can use

Creates the SQL query you want to execute

Calls Dapper's Query method with the type of the returned data

Provides parameters to Dapper to be added to the SQL command

PERFORMANCE TIP `FromSqlRaw`/`FromSqlInterpolated` methods must return all the columns mapped to the entity calls, and even if you add a LINQ `Select` after the `FromSqlRaw`/`FromSqlInterpolated` method, it still returns all the columns. As a result, Dapper is likely to be faster at loading a few columns from the database than any of the EF Core `RawSql`... methods.

Don't be afraid to mix EF Core and Dapper, especially if you have a performance problem. I use Dapper with EF Core in part 3 to get a performance improvement because I wrote an improved SQL query that sorted on average review stars. The downside of Dapper is that it doesn't know anything about navigational properties, so working with linked entities takes more code in Dapper than it does in EF Core.

11.6 Accessing information about the entity classes and database tables

Sometimes, it's useful to get information about how the entity classes and properties are mapped to the database tables and columns. EF Core provides two sources of information, one that emphasizes the entity classes and one that focuses more on the database:

- `context.Entry(entity).Metadata`—Has more than 20 properties and methods that provide information on the primary key, foreign key, and navigational properties
- `context.Model`—Has a set of properties and methods that provides a similar set of data to the `Metadata` property, but focuses more on the database tables, columns, constraints, indexes, and so on

Here are some examples of how you might use this information to automate certain services:

- Recursively visiting an entity class and its relationships so that you can apply some sort of action in each entity class, such as resetting its primary-key values

- Obtaining the settings on an entity class, such as its delete behavior
- Finding the table name and column names used by an entity class so that you can build raw SQL with the correct table and column names

> **EF6** EF6.x provided some model information, but it was complex to use and incomplete. EF Core has a comprehensive, easy-to-use set of `Model` information, but there isn't much documentation other than the methods' comments.

The following sections provide examples of using these sources.

11.6.1 Using context.Entry(entity).Metadata to reset primary keys

In section 6.2.3, you learned how to copy an entity class with certain relationships by resetting the primary keys manually. I needed a similar feature for a client's application, so I built a service that resets the primary keys automatically as a good example of using `context.Entry(entity).Metadata`.

The example in section 6.2.3 copied an `Order` entity with two `LineItem` entities, but the `Book` entity class shouldn't be copied. The following listing is a copy of the listing from section 6.2.3.

Listing 11.22 Creating an `Order` with two `LineItems` ready to be copied

```
var books = context.SeedDatabaseFourBooks();
var order = new Order
{
    CustomerId = Guid.Empty,
    LineItems = new List<LineItem>
    {
        new LineItem
        {
            LineNum = 1, ChosenBook = books[0], NumBooks = 1
        },
        new LineItem
        {
            LineNum = 2, ChosenBook = books[1], NumBooks = 2
        },
    }
};
context.Add(order);
context.SaveChanges();
```

Create an Order with two LinItems that you want to copy.

For this test, add four books to use as test data.

Set CustomerId to the default value so that the query filter lets you read the order back.

Adds the first LineNum linked to the first book

Adds the second LineNum linked to the second book

Writes this Order out to the database

In the version in chapter 6, you read in the `Order` and `LineItems` entity classes, and then reset the primary keys manually. But in this example, you build a class called `PkResetter` to perform this task automatically. The following listing shows this code in the `PkResetter` class.

Listing 11.23 Using metadata to visit each entity and reset its primary key

```
public class PkResetter
{
    private readonly DbContext _context;
    private readonly HashSet<object> _stopCircularLook;

    public PkResetter(DbContext context)
    {
        _context = context;
        _stopCircularLook = new HashSet<object>();
    }

    public void ResetPksEntityAndRelationships(object entityToReset)
    {
        if (_stopCircularLook.Contains(entityToReset))
            return;

        _stopCircularLook.Add(entityToReset);

        var entry = _context.Entry(entityToReset);
        if (entry == null)
            return;

        var primaryKey = entry.Metadata.FindPrimaryKey();
        if (primaryKey != null)
        {
            foreach (var primaryKeyProperty in primaryKey.Properties)
            {
                primaryKeyProperty.PropertyInfo
                    .SetValue(entityToReset,
                    GetDefaultValue(
                        primaryKeyProperty.PropertyInfo.PropertyType));
            }
        }

        foreach (var navigation in entry.Metadata.GetNavigations())
        {
            var navProp = navigation.PropertyInfo;

            var navValue = navProp.GetValue(entityToReset);
            if (navValue == null)
                continue;

            if (navigation.IsCollection)
            {
                foreach (var item in (IEnumerable)navValue)
                {
                    ResetPksEntityAndRelationships(item);
                }
            }
            else
            {
                ResetPksEntityAndRelationships(navValue);
            }
```

Used to stop circular recursive steps

This method will recursively look at all the linked entities and reset their primary keys.

If the method has already looked at this entity, the method exits.

Remembers that this entity has been visited by this method

Deals with an entity that isn't known by your configuration

Gets the primary-key information for this entity

Resets every property used in the primary key to its default value

Gets all the navigational properties for this entity

Gets a property that contains the navigation property

Gets the navigation property value

If null, skips the navigation property

If the navigation property is collection, visits every entity

Recursively visits each entity in the collection

If a singleton, visits that entity

```
            }
        }
    }
```

That listing might seem to be a lot of code to reset the three primary keys, but it will work with any entity-class configuration, so you can use it anywhere. Here is a list of the various `Metadata` properties and methods used in listing 11.23:

- *Find the entity's primary key*—`entry.Metadata.FindPrimaryKey()`
- *Get the primary key's properties*—`primaryKeyProperty.PropertyInfo`
- *Find the entity's navigational relationships*—`Metadata.GetNavigations()`
- *Get a navigational relationship's property*—`navigation.PropertyInfo`
- *Checking whether the navigational property is a collection*—`navigation.IsCollection`

NOTE The `PkResetter` class assumes that the primary keys and the navigational properties are stored in a property, but in fact, these values could be in backing fields or shadow properties. This simplification was used to make the code shorter and easier to read.

11.6.2 Using context.Model to get database information

The `context.Model` property gives you access to the `Model` of the database that EF Core builds on first use of an application's DbContext. The `Model` contains some data similar to `context.Entry(entity).Metadata`, but it also has specific information of the database schema. Therefore, if you want to do anything with the database side, `context.Model` is the right information source to use.

I used the `context.Model` source to build the EfCore.EfSchemaCompare library that I mentioned in section 9.5.3. But for a smaller example, you'll produce a method that returns an SQL command to delete a collection of entities with a common foreign key. The reason for doing this is to improve the delete performance of a group of dependent entities.

If you deleted a group of dependent entities via EF Core, you would typically read in all the entities to delete, and EF Core would delete each entity with a separate SQL command. The method in the following listing produces a single SQL command that deletes all the dependent entities in one SQL command without the need to read them in. This process, therefore, is much quicker than EF Core, especially on large collections.

Listing 11.24 Using `context.Model` to build a quicker dependent delete

This method provides a quick way to delete all the entities linked to a principal entity.

Gets the Model information for the given type, or null if the type isn't mapped to the database

```
public string BuildDeleteEntitySql<TEntity>
    (DbContext context, string foreignKeyName)
    where TEntity : class
{
    var entityType = context.Model.FindEntityType(typeof(TEntity));
```

```
    var fkProperty = entityType?.GetForeignKeys()
        .SingleOrDefault(x => x.Properties.Count == 1
            && x.Properties.Single().Name == foreignKeyName)
        ?.Properties.Single();

    if (fkProperty == null)
        throw new ArgumentException($"Something wrong!");

    var fullTableName = entityType.GetSchema() == null
        ? entityType.GetTableName()
        : $"{entityType.GetSchema()}.{entityType.GetTableName()}";

    return $"DELETE FROM {fullTableName} " +
           $"WHERE {fkProperty.GetColumnName()}"
           + " = {0}";
}
```

Looks for a foreign key with a single property with the given name

If any of those things doesn't work, the code throws an exception.

Forms the full table name, with a schema if required

Forms the main part of the SQL code

Adds a parameter that the ExecuteSqlRaw can check

Having found the right entity/table and checked that the foreign key name matches, you can build the SQL. As the listing shows, you have access to the table's name and schema, plus the column name of the foreign key. The following code snippet shows the output of the `BuildDeleteEntitySql` method in listing 11.24 with a `Review` entity class for the `TEntity` and a foreign-key name of `BookId`:

```
DELETE FROM Review WHERE BookId = {0}
```

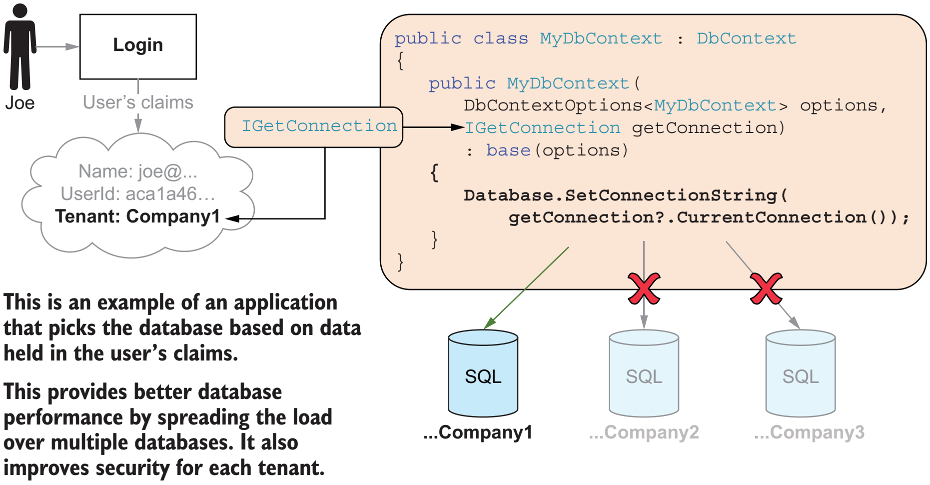
The SQL command is applied to the database by calling the `ExecuteSqlRaw` method, with the SQL string as the first parameter and the foreign-key value as the second parameter.

> **NOTE** The `BuildDeleteEntitySql` class assumes that the foreign key is singular, but a foreign key could be a composite key with multiple values. This simplification was used to make the code shorter and easier to read.

Although this example is simple, it does show that using the `Model` methods allows you to obtain information entity classes with its relationships and match those entity classes to the database's schema.

11.7 Dynamically changing the DbContext's connection string

EF Core 5 makes it easier to change the connection string in an instance of an application's DbContext. Now it provides a method called `SetConnectionString` that allows you to change the connection string at any time so that you can change the database you are accessing at any time. I typically use this feature to pick different databases based on the person who is logged in, where the user is located, and so on. This process is known as *database sharding*, and it offers better performance because the user's data is spread over multiple databases. It can also add some security by placing all the data for one group of users in one database. Figure 11.6 shows the `SetConnectionString` method used to implement a database sharding system with EF Core.



```
public class MyDbContext : DbContext
{
   public MyDbContext(
      DbContextOptions<MyDbContext> options,
      IGetConnection getConnection)
      : base(options)
   {
      Database.SetConnectionString(
          getConnection?.CurrentConnection());
   }
}
```

This is an example of an application that picks the database based on data held in the user's claims.

This provides better database performance by spreading the load over multiple databases. It also improves security for each tenant.

Figure 11.6 A user logging in to an ASP.NET Core application. The application uses the user's details to work out what database they should access and adds a claim called `Tenant` to hold that information. This works because when the application's DbContext is created, its constructor code runs to set the database connection string. This code uses the injected `IGetConnection` class, which returns a connection string based on the `Tenant` claim that maps the user to the correct database.

EF Core 5 made one other important change: the connection string can be `null` when you first create the application's DbContext. (Before EF Core 5 the connection string could not be `null`.) The connection string can be `null` until you need to access the database. This feature is useful because on startup, there would be no tenant information, so the connection string would be `null`. With the EF Core 5 change, however, your EF Core configuration code can run without needing a connection string. Another example is my `EfCore.GenericServices` library, which needs to scan the entities used in a database on startup. Now that library works even if the connection string is `null`.

11.8 Handling database connection problems

With relational database servers, especially in the cloud, a database access can fail because the connection times out or certain transient errors occur. EF Core has an execution strategy feature that allows you to define what should happen when a timeout occurs, how many timeouts are allowed, and so on. Providing an execution strategy can make your application less likely to fail due to connection problems or transient internal errors.

> **EF6** EF Core's execution strategy is an improvement on the EF6.x execution strategy, as EF Core can handle retries in a transaction.

The SQL Server database provider includes an execution strategy that's specifically tailored to SQL Server (including SQL Azure). It's aware of the exception types that can be retried and has sensible defaults for maximum retries, delay between retries, and so on. This listing shows how to apply this strategy to the setup of SQL Server, with the execution strategy shown in bold.

Listing 11.25 Setting up a DbContext with the standard SQL execution strategy

```
var connection = @"Server=(localdb)\mssqllocaldb;Database=… etc.”;
var optionsBuilder =
    new DbContextOptionsBuilder<EfCoreContext>();

optionsBuilder.UseSqlServer(connection,
    option => option.EnableRetryOnFailure());
var options = optionsBuilder.Options;

using (var context = new EfCoreContext(options))
{
   … normal code to use the context
```

Normal EF Core queries or `SaveChanges` calls will automatically be retried without your doing anything. Each query and each call to `SaveChanges` is retried as a unit if a transient failure occurs. But database transactions need a little more work.

11.8.1 Handling database transactions with EF Core's execution strategy

Because of the way that an execution strategy works, you need to adapt any code that uses a database transaction in which you have multiple calls to `SaveChanges` within an isolated transaction. (See section 4.7.2 for information on how transactions work.) The execution strategy works by rolling back the whole transaction if a transient failure occurs and then replaying each operation in the transaction; each query and each call to `SaveChanges` is retried as a unit. For all the operations in the transaction to be retried, the execution strategy must be in control of the transaction code.

The following listing shows both the addition of the SQL Server `EnableRetryOnFailure` execution strategy and the use of the execution strategy (in bold) with a transaction. The transaction code is written in such a way that if a retry is needed, the whole transaction is run again from the start.

Listing 11.26 Writing transactions when you've configured an execution strategy

```
var connection = @"Server=(localdb)\mssqllocaldb;Database=… etc.”;
var optionsBuilder =
    new DbContextOptionsBuilder<EfCoreContext>();

optionsBuilder.UseSqlServer(connection,
    option => option.EnableRetryOnFailure());
var options = optionsBuilder.Options
```

Configures the database to use the SQL execution strategy, so you have to handle transactions differently

```
using (var context = new Chapter09DbContext(options))
{
    var strategy = context.Database
        .CreateExecutionStrategy();
    strategy.Execute(() =>
    {
        try
        {
            using (var transaction = context
                .Database.BeginTransaction())
            {
                context.Add(new MyEntity());
                context.SaveChanges();
                context.Add(new MyEntity());
                context.SaveChanges();
                transaction.Commit();
            }
        }
        catch (Exception e)
        {
            //Error handling to go here
            throw;
        }
    });
}
```

Creates an IExecutionStrategy instance, which uses the execution strategy you configured the DbContext with

The important thing is to make the whole transaction code into an Action method it can call.

The rest of the transaction setup and running your code are the same.

WARNING The code in listing 11.26 is safe when it comes to a retry. By *safe*, I mean that the code will work properly. But in some cases, such as when data outside the execution strategy retry action is altered, the retry could cause problems. An obvious example is an `int count = 0` variable defined outside the scope of the retry action that's incremented inside the action. In this case, the value of the `count` variable would be incremented again if a retry occurred. Bear this warning in mind when you design transactions if you're using the execution strategy retry facility.

11.8.2 Altering or writing your own execution strategy

In some cases, you might need to change the execution strategy for your database. If there's an existing execution strategy for your database provider (such as SQL Server), you can change some options, such as the number of retries or the SQL errors to be retried.

If you want to write your own execution strategy, you need to implement a class that inherits the interface `IExecutionStrategy`. I recommend that you look at the EF Core internal class called `SqlServerExecutionStrategy` as a template. You can find this template at http://mng.bz/A1DK.

After you've written your own execution strategy class, you can configure it into your database by using the `ExecuteStrategy` method in the options, as shown in bold in the next listing.

Listing 11.27 Configuring your own execution strategy into your DbContext

```
var connection = this.GetUniqueDatabaseConnectionString();
var optionsBuilder =
    new DbContextOptionsBuilder<Chapter09DbContext>();

optionsBuilder.UseSqlServer(connection,
    options => options.ExecutionStrategy(
        p => new MyExecutionStrategy()));

using (var context = new Chapter09DbContext(optionsBuilder.Options))
{
   … etc.
```

Summary

- You can use EF Core's entity `State` property, with a little help from a per-property `IsModified` flag, to define what will happen to the data when you call `SaveChanges`.
- You can affect the `State` of an entity and its relationships in several ways. You can use the `DbContext's` methods `Add`, `Remove`, `Update`, `Attach`, and `TrackGraph`; set the `State` directly; and track modifications.
- The `DbContext`'s `ChangeTracker` property provides several ways to detect the `State` of all the entities that have changed. These techniques are useful for marking entities with the date when an entity was created or last updated, or logging every `State` change for any of the tracked entities.
- The `Database` property has methods that allow you to use raw SQL command strings in your database accesses.
- You can access information about the entities and their relationships via the `Entry(entity).Metadata` and the database structure via the `Model` property.
- EF Core contains a system that allows you to provide a retry capability. This system can improve reliability by retrying accesses if there are connection or transient errors in your database.

For readers who are familiar with EF6:

- EF Core has changed the ways that the entity's `State` is set based on lessons learned from EF6.x. Now, it is more likely to set the entity's `State` to the correct value for the action you're using.
- EF Core introduces a new method called `TrackGraph`, which will traverse a graph of linked entities and call your code to set each entity's `State` to the value you require.
- The way you use raw SQL commands in EF Core is different from the way it's done in EF6.x.

- EF Core’s `Entry(entity).Metadata` and `Model` properties are a tremendous improvement over EF6.x’s access to the model metadata. Now you can access every aspect of the database model.
- EF Core’s execution strategy is an improvement on the EF6.x execution strategy, as EF Core can handle retries in a database transaction.

Part 3

Using Entity Framework Core in real-world applications

In parts 1 and 2, you learned about EF Core in some detail, and at every step, I tried to provide examples of using each feature or approach. Now, in part 3, you are going to build a more complex version of the Book App and then performance-tune it. There will be some new information, such as looking at Cosmos DB in chapter 16 and unit testing in chapter 17, but the focus of part 3 is on using EF Core rather than learning about it.

I am a freelance contractor. My clients want their requirements turned into robust, secure, high-performance applications—and they want them quickly! To provide these applications, I use approaches and libraries that are robust, secure, and high-performance. The first two chapters in part 3 cover various approaches I have learned over the years that allow me to build applications quickly. As Kent Beck said, "Make it work, make it right, make it fast."

Having built an application in chapters 12 and 13, we move on to performance tuning. The initial Book App has around 700 real books in it, but for performance testing, we clone that data to 100,000 books and more. That number of books exposes some database performance issues, and over two and a half chapters, you will improve the Book App's performance by using several techniques.

Chapter 16 is about using Cosmos DB to add a final performance tune of the Book App. This chapter exposes the differences between a relational (SQL) database and a NoSQL database so that you are better informed about where and how to use either type of database.

Finally, chapter 17 covers unit testing, with the focus on EF Core. Unit testing when a database is involved requires careful thought, especially if you don't want the unit test to run slowly. I share several techniques and approaches, and I provide a NuGet package that I built, called EfCore.TestSupport. This library contains setup methods that help you unit test EF Core applications safely and quickly.

12

Using entity events to solve business problems

This chapter covers

- Understanding the types of events that work well with EF Core
- Using domain events to trigger extra business rules
- Using integration events to synchronize two parts of your application
- Implementing an Event Runner and then improving it

In software, the term *event* covers a wide range of architectures and patterns. Generally, it means "Action A triggers action B." You saw some C# events in chapter 11, such as events in which an entity state changes (section 11.4.4). But this chapter is about another, quite different type of event, which I call an *entity event* because it is held in your entity classes. Using an entity event is like putting a message in the entity class for someone to read later.

The purpose of entity events is to trigger business logic when something changes in an entity class. In section 12.1.1 I show an example where a change in an address's details causes the sales tax on a quote to be updated. This example is implemented by detecting a change to the address details and sending an entity event (message) that runs some business logic that updates the sales tax for quotes at that address.

In addition to the entity events, you need parts that make them work. At the heart of the entity-event approach is code that I call the *Event Runner*, whose job is to read in all the entity events and run the specific business code (referred to as *event handlers*) associated with each entity event. Each event handler contains the specific business logic for that entity event, and each entity-event message provides the data that the event handler needs.

The Event Runner runs before the `SaveChanges` and `SaveChangesAsync` methods are called. The best way is to override the `SaveChanges` and `SaveChangesAsync` methods and then run the Event Runner into the methods. I refer to these `SaveChanges` and `SaveChangesAsync` methods as being event-enhanced.

12.1 Using events to solve business problems

I came up with the name entity events, but much cleverer people coined the terms *domain events* and *integration events* to define two uses of entity events. In this chapter, you'll learn about domain events and integration events, as well as the situations in which they can be used. Then you will implement event-enhanced `SaveChanges` and `SaveChangesAsync` methods that you can use in your applications.

12.1.1 Example of using domain events

I was introduced to domain events by one of my clients. The client had used an event system discussed by Jimmy Bogard in his article "A better domain events pattern" (see http://mng.bz/oGNp), which described how to add domain events to EF Core. I had read this article some years before and didn't get it, but my client did and used domain events successfully. Seeing entity events being used in a real application persuaded me of their usefulness, and I went on to use domain events to solve several business requirements and performance issues in the client's application. The following example is taken from one of those business requirements.

My client's company sells bespoke constructions in the United States, and every project starts with a quote to send to the client. The construction could be anywhere in the United States, and the state where the work is done defines the sales tax. As a result, the sales tax had to be recalculated when any of the following things happened:

- *A new quote was created.* By default, a new quote doesn't have a location, so the business rule was to give it the highest sales tax until the location was specified.
- *The job location was set or changed.* The sales tax had to be recalculated, and it was the sales team's job to select a location from a list of known locations.
- *A location's address changed.* All the quotes linked to that location had to be recalculated to make sure that the sales tax was correct.

Now, you could add business logic for all these actions, but doing that would make the frontend more complex, and you might miss one area where a location changed and then the sales tax was wrong. The solution was to use events that triggered if a quote's location was added or updated, and it worked well. A change in the `Location` entity class created a domain event to trigger an event handler that recalculated the sales tax for a quote (or quotes). Each domain event needed a slightly different piece of business logic, plus a common service to calculate the tax. Figure 12.1 shows an example of what might happen if the address of a location changes.

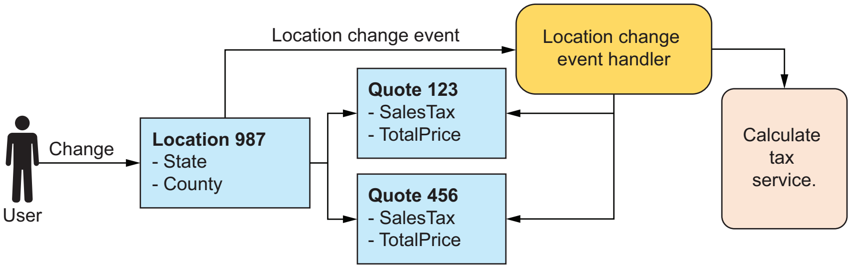


Figure 12.1 Rather than add code at the frontend to run some business logic when a location changes, you can catch the change in the entity class and add a domain event to the entity class. When `SaveChanges` is called, a piece of code added to `SaveChanges` looks at any domain events and runs the appropriate event handler to make sure that all the open `Quotes` have their `SalesTax` recalculated.

I won't delve into how this example works now, as this section describes where and why events are useful. Suffice it to say that in section 12.4, you write code to handle entity events and improve that code as you go deeper into this approach.

12.1.2 Example of integration events

The second use of an entity event is a more complex situation. In chapter 13, you will learn multiple ways to improve the performance of your EF Core database accesses. One of these approaches is to precalculate the data you need to show to the user and store it in another database used only for displaying data to the user. This approach improves read performance and scalability.

The normal SQL commands for the Book App, for example, calculate the average star rating of a book by dynamically calculating the average across all the `Book`'s `Reviews`. That technique works fine for a small number of `Books` and `Reviews`, but with large numbers, sorting by average review ratings can be slow. In chapter 16, you will use a Query Responsibility Segregation (CQRS) database pattern to store the precalculated data in a separate, read-side database. The problem is making sure that the write-side SQL database and the read-side Cosmos DB database are always in step.

I use this solution in chapter 16: when writing the SQL database, run a transaction that contains both the update to the SQL database and the update to the read-side Cosmos DB database. If either database fails, both databases will fail, which means that they can't get out of step. Figure 12.2 shows how this solution might work.

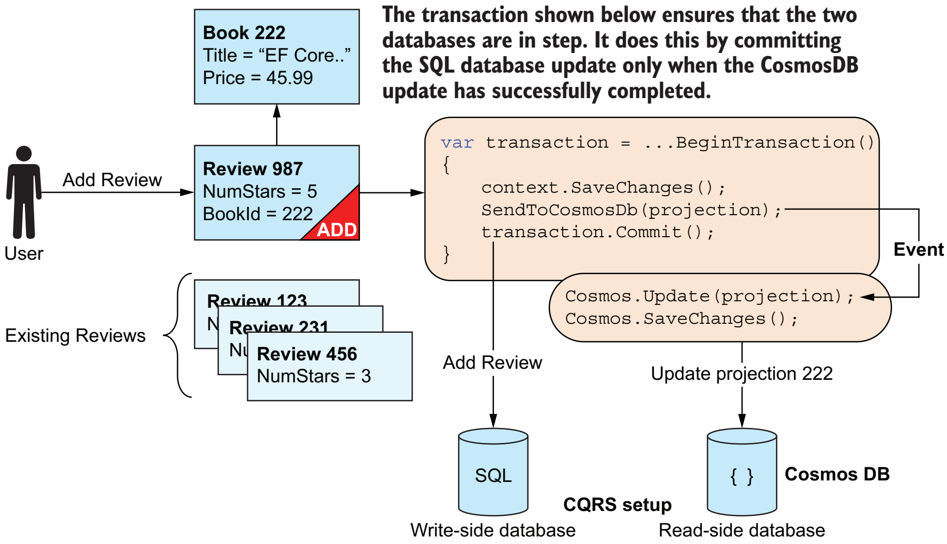


Figure 12.2 A CQRS database using a relational database as the primary data store, with a Cosmos DB database acting as a read-side database to improve performance. The issue is how to ensure that the two databases are in step—in this case, the `Book` projection in the Cosmos DB matches what the SQL database has. This figure is an example of using integration events to cross the boundary of the code handling the relational database and the code handling the Cosmos DB database.

12.2 Defining where domain events and integration events are useful

The two examples you have seen use events in different ways; the sales-tax event example is focused within a specific part of the code relating to customers and quotes, and the CQRS example is linking two quite different parts of an application. These two types of events have been given names, primarily by the Domain-Driven Design (DDD) community, but you will see that events can be used in normal, non-DDD entity types too.

NOTE I cover how to apply a DDD approach to EF Core entity classes in chapter 13. But in this chapter, you will learn how to use events in non-DDD entity types.

DDD talks a lot about a bounded context, which represents a defined part of software where particular terms, definitions, and rules apply in a consistent way. A bounded context is about applying the Separation of Concerns (SoC) principle at the macro level. The part 3 Book App, for example, is broken into different bounded contexts: one handles displaying the books by using a SQL database, another provides a way to display the books by using a NoSQL database, and yet another handles processing a user's order. So using the term bounded context, you can categorize the two event types as follows:

- The sales-tax example is referred to as a domain event because it is working exclusively within a single bounded context.
- The CQRS example is referred to as an integration event because it crosses from one bounded context to another.

NOTE I cover bounded contexts in more detail in chapter 13.

12.3 Where might you use events with EF Core?

I don't want to suggest that you should do everything by using entity events, but I do think that entity events are a good approach to learn. Where would you use entity events? The answer is best provided by some examples:

- Setting or changing an `Address` triggers a recalculation of the sales-tax code of a `Quote`.
- Creating an `Order` triggers a check on reordering `Stock`.
- Updating a `Book` triggers an update of that `Book`'s `Projection` on another database.
- Receiving a `Payment` that pays off the debt triggers the closing of the `Account`.
- Sending a `Message` to an `external service`.

Each example has two entity class names in an italic monospace font. These entity classes are different but not tightly linked: `Address/Quote`, `Order/Stock`, `Book/Projection`, `Payment/Account`, and `Message/external service`. When I say that the classes are not tightly linked, I mean that the second class isn't dependent on the first class. If the `Address` entry were deleted, for example, the `Quote` entry wouldn't be deleted.

NOTE A good indication that domain events could help is when your business logic is going to be working on two different groups of data.

In all these cases, the first class could be handled in the standard way (that is, not using entity event), and a domain event could trigger an event handler to handle the update to the second class. Conversely, events aren't useful when the entity classes are already closely linked. You wouldn't use events to set up each `LineItem` in an `Order`, for example, because the two classes are closely linked to each other.

Another time where events can be useful is when you want to add a new feature to some existing code and don't want to alter the existing methods and business logic. If the new feature doesn't change the existing code, you might have a case for using events even if the two entity classes are closely linked. Chapter 15 has a good example that improves the performance of the existing Book App. I don't want to change the existing code, which works, but I want to add some cached values to the `Book` entity class, and using domain events is a great solution.

Having said all that, it's likely that you won't use lots of domain events. There were only 20 domain events in the system from which the sales-tax example came, for example, but some of these events were critical to the features and especially the performance of the application.

Integration events are even rarer; they are useful only when you have two bounded contexts that need to work together. But if you need to synchronize two different parts of your application, integration events are among the best approaches you can use.

Overall, I find events like these to be so useful that I have built a library, `EfCore.GenericEventRunner`, to easily add entity events (both domain and integration) to an application when I need them. But before I get into how to implement such a system, consider the pros and cons of using domain and integration events.

12.3.1 Pro: Follows the SoC design principle

The event systems already described provide a way to run separate business rules on a change in an entity class. In the location-change/sales-tax example, the two entities are linked in a nonobvious way; changing the location of a job causes a recalculation of the sales tax for any linked quotes. When you apply the SoC principle, these two business rules should be separated.

You could create some business logic to handle both business rules, but doing so would complicate a simple update of properties in an address. By triggering an event if the `State`/`County` properties are changed, you can keep the simple address update and let the event handle the second part.

12.3.2 Pro: Makes database updates robust

The design of the code that handles domain events is such that the original change that triggers the event and the changes applied to entity classes via the called event handler are saved in the same transaction. Figure 12.3 shows this code in action.

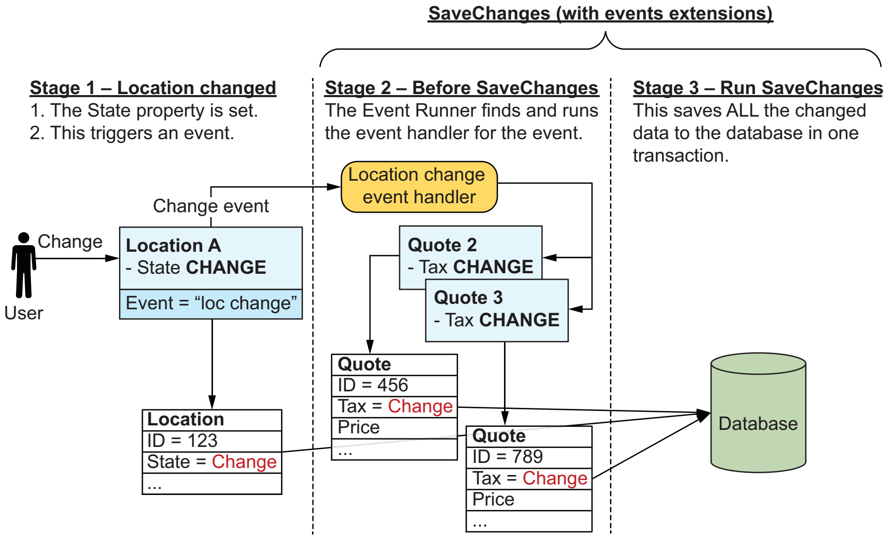


Figure 12.3 The domain events system saves both the initial `Location` update that triggered the event and the changes made to `Quote` entity classes in one transaction. The database will contain all the changes in one go, so the two types of updates can’t get out of date.

As you will see in section 12.5, the integration event implementation is robust too. If the integration event fails, the database update will be rolled back, ensuring that the local database and the external service and different database are in step.

12.3.3 Con: Makes your application more complex

One of the downsides of using events is that your code is going to be more complicated. Even if you use a library such as `EfCore.GenericEventRunner` to manage the events, you will still have to create your events, add the events to your entity classes, and write your event handlers, which requires more code than building services for your business logic, as covered in chapter 4.

But the trade-off of events that need more code is that the two business logic parts are decoupled. Changes to the address become a simple update, for example, while the event makes sure that the tax code is recalculated. This decoupling reduces the business complexity that the developer has to deal with.

12.3.4 Con: Makes following the flow of the code more difficult

It can be hard to understand code that you didn't write or wrote a while back. One helpful VS/VS Code feature that I use is Go to Implementation, which lets me jump to a method's code so that I can dig down through the code to understand how each part works before I change it.

You can do the same thing when you use events, but that technique does add one more level of indirection before you get to the code. For the sales-tax-change example in figure 12.1, you would need to click the `LocationChangedEvent` class to find the `LocationChangedEventHandler` that has the business code you're looking for—only one more step, but a step you don't need if you don't use events.

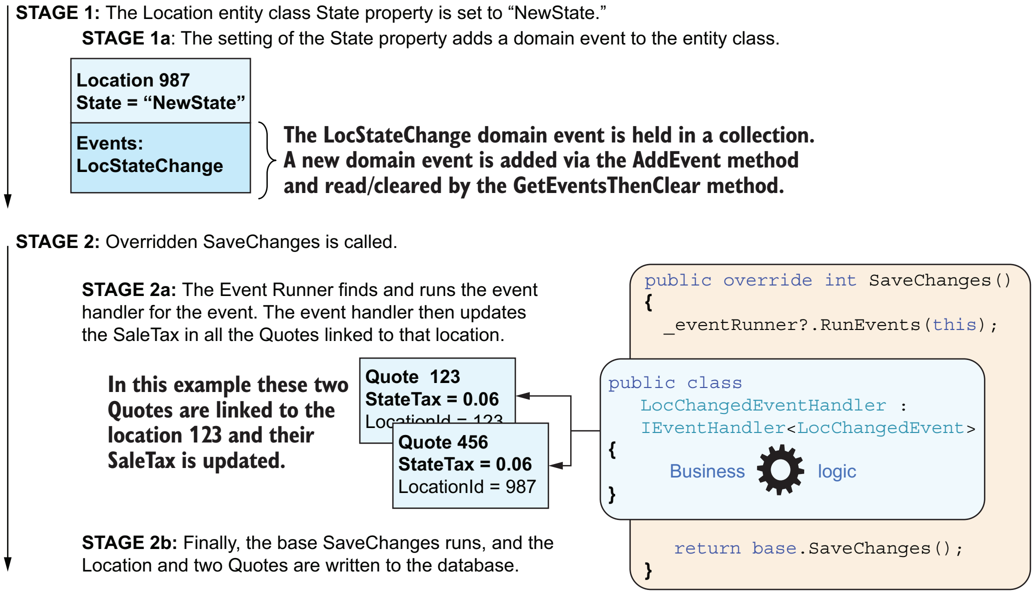
12.4 Implementing a domain event system with EF Core

In this section, you are going to implement a domain event system in EF Core. First, you'll add the ability to hold entity events in your entity classes. Then you'll override the DbContext's `SaveChanges` so that you have extra logic to extract the entity events and to find and run each corresponding event handler.

Figure 12.4 shows the code and steps needed to implement a domain events system, using the example described in figure 12.1, where a `Location`'s `State` property changes. In this example, two `Quotes` are linked to that location, so their `SalesTax` property should be updated to the correct sales tax at that location.

To implement this domain event system, add the following code to your application:

1. You create some domain events classes to be triggered.
2. Add code to the entity classes to hold the domain events.
3. Alter the code in the entity class to detect a change on which you want to trigger an event.



STAGE 1: The Location entity class State property is set to "NewState."

STAGE 1a: The setting of the State property adds a domain event to the entity class.

The LocStateChange domain event is held in a collection. A new domain event is added via the AddEvent method and read/cleared by the GetEventsThenClear method.

STAGE 2: Overridden SaveChanges is called.

STAGE 2a: The Event Runner finds and runs the event handler for the event. The event handler then updates the SaleTax in all the Quotes linked to that location.

In this example these two Quotes are linked to the location 123 and their SaleTax is updated.

STAGE 2b: Finally, the base SaveChanges runs, and the Location and two Quotes are written to the database.

```
public override int SaveChanges()
{
  _eventRunner?.RunEvents(this);
```

```
public class
   LocChangedEventHandler :
   IEventHandler<LocChangedEvent>
{
      Business     logic
}
```

```
    return base.SaveChanges();
}
```

Figure 12.4 Stage 1 shows that a change of a `Location`'s `State` property will cause a domain event to be added to the `Location` entity class. In stage 2, when the overridden `SaveChanges` method is called, it will read in any domain events in the tracked entities and then find and run the appropriate event handler for each domain event. In this example, the event handler updates the `SalesTax` property in all the `Quotes` linked to that `Location`.

4. Create some event handlers that are matched to the events. These event handlers may alter the calling entity class or access the database or business logic to execute the business rules it is designed to handle.
5. Build an Event Runner that finds and runs the correct event handler that matches each found event.
6. Add the Event Runner to the DbContext, and override the `SaveChanges` (and `SaveChangesAsync`) methods in your application's DbContext.
7. When the Event Runner has finished, run the base `SaveChanges`, which updates the database with the original changes and any further changes applied by the event handlers.
8. Register the Event Runner and all the event handlers.

Next, you will follow these steps to build each part of this approach.

12.4.1 Create some domain events classes to be triggered

There are two parts to creating an event. First, an event must have an interface that allows the Event Runner to refer to it. This interface can be empty, representing an event. (I call this interface `IDomainEvent` in this example.) I use this interface to represent a domain event inside the Event Runner.

Each application event contains data that is specific to the business needs. The following listing shows the `LocationChangedEvent` class, which needs only the `Location` entity class.

Listing 12.1 The `LocationChangedEvent` class, with data that the event handler needs

```
public class LocationChangedEvent : IDomainEvent
{
    public LocationChangedEvent(Location location)
    {
        Location = location;
    }

    public Location Location { get; }
}
```

The event class must inherit the IDomainEvent. The Event Runner uses the IDomainEvent to represent every domain event.

The event handler needs Location to do the Quote updates.

Each event should send over the data that the event handler needs to do its job. Then it is the event handler's job to run some business logic, using the data provided by the event.

12.4.2 Add code to the entity classes to hold the domain events

The entity class must hold a series of events. These events aren't written to the database but are there for the Event Runner to read via a method. The following listing shows a class that an entity can inherit to add the event feature to its capabilities.

Listing 12.2 The class that entity classes inherit to create events

```
public class AddEventsToEntity : IEntityEvents
{
    private readonly List<IDomainEvent>
        _domainEvents = new List<IDomainEvent>();

    public void AddEvent(IDomainEvent domainEvent)
    {
        _domainEvents.Add(domainEvent);
    }

    public ICollection<IDomainEvent>
        GetEventsThenClear()
    {
        var eventsCopy = _domainEvents.ToList();
        _domainEvents.Clear();
        return eventsCopy;
    }
}
```

The IEntityEvents defines the GetEventsThenClear method for the Event Runner.

The list of IDomainEvent events is stored in a field.

The AddEvent is used to add new events to the _domainEvents list.

This method is called by the Event Runner to get the events and then clear the list.

The entity class can call the `AddEvent` method, and the Event Runner can get the domain events via the `GetEventsThenClear` method. Getting the domain events also clears the events in the entity class, because these messages will cause an event handler to be executed, and you want the event handler to run only once per domain event. Remember that domain events are nothing like C# events; domain events are messages passed to the Event Runner via the entity classes, and you want a message to be used only once.

12.4.3 Alter the entity class to detect a change to trigger an event on

An event is normally something being changed or something reaching a certain level. EF Core allows you to use backing fields, which make it easy to capture changes to scalar properties. The following listing shows the `Location` entity class that creates a domain event when the `State` property changes.

Listing 12.3 The `Location` entity class creates a domain event if the `State` is changed

```
public class Location : AddEventsToEntity
{
    public int LocationId { get; set; }
    public string Name { get; set; }

    private string _state;

    public string State
    {
        get => _state;
        set
        {
            if (value != _state)
                AddEvent(
                    new LocationChangedEvent(this));
            _state = value;
        }
    }

}
```

This entity class inherits the AddEventsToEntity to gain the ability to use events.

These normal properties don't generate events when they are changed.

The backing field contains the real value of the data.

The setter is changed to send a LocationChangedEvent if the State value changes.

This code will add a LocationChangedEvent to the entity class if the State value changes.

NOTE Collection navigational properties are a little harder to check for changes, but DDD-styled entity classes (covered in chapter 13) make this check much simpler.

12.4.4 Create event handlers that are matched to the domain events

Event handlers are key to using events in your application. Each event handler contains some business logic that needs to be run when the specific event is found. For the Event Runner to work, every event handler must have the same signature, which is defined by an interface I created for this example, called `IEventHandler<T> where T : IDomainEvent`. The following listing shows the event handler that updates the `SalesTax` in every `Quote` that is linked to the `Location` that changed.

Listing 12.4 The event handler updates the sales tax on `Quotes` linked to this `Location`

```
public class LocationChangedEventHandler
    : IEventHandler<LocationChangedEvent>
{
    private readonly DomainEventsDbContext _context;
    private readonly
        ICalcSalesTaxService _taxLookupService;

    public LocationChangedEventHandler(
        DomainEventsDbContext context,
        ICalcSalesTaxService taxLookupService)
    {
        _context = context;
        _taxLookupService = taxLookupService;
    }

    public void HandleEvent
        (LocationChangedEvent domainEvent)
    {
        var salesTaxPercent = _taxLookupService
            .GetSalesTax(domainEvent.Location.State);

        foreach (var quote in _context.Quotes.Where(
            x => x.WhereInstall == domainEvent.Location))
        {
            quote.SalesTaxPercent = salesTaxPercent;
        }
    }
}
```

This class must be registered as a service via DI.

Every event handler must have the interface IEventHandler<T>, where T is the event class type.

This specific event handler needs two classes registered with DI.

The Event Runner will use DI to get an instance of this class and will fill in the constructor parameters.

The method from the IEventHandler<T> that the Event Runner will execute

Uses another service to calculate the right sales tax

Sets the SalesTax on every Quote that is linked to this Location

The key point here is that the event handler is registered as a service so that the Event Runner can get an instance of the event handler class via dependency injection (DI). The event handler class has the same access to DI services that normal business logic does. In this case, the `LocationChangedEventHandler` injects the application's DbContext and the `ICalcSalesTaxService` service.

12.4.5 Build an Event Runner that finds and runs the correct event handler

The Event Runner is the heart of the event system: its job is to match each event to an event handler and then invoke the event handler's method, providing the event as a parameter. This process uses NET Core's `ServiceProvider` to get an instance of the event handler, which allows the event handlers to access other services. Figure 12.5 provides a visual representation of what the Event Runner does.

> **NOTE** If you don't have NET Core's DI feature available in your application, you could replace the DI by handcoding a `switch` statement with code to create each event manager. This technique is harder to manage, but it will work.

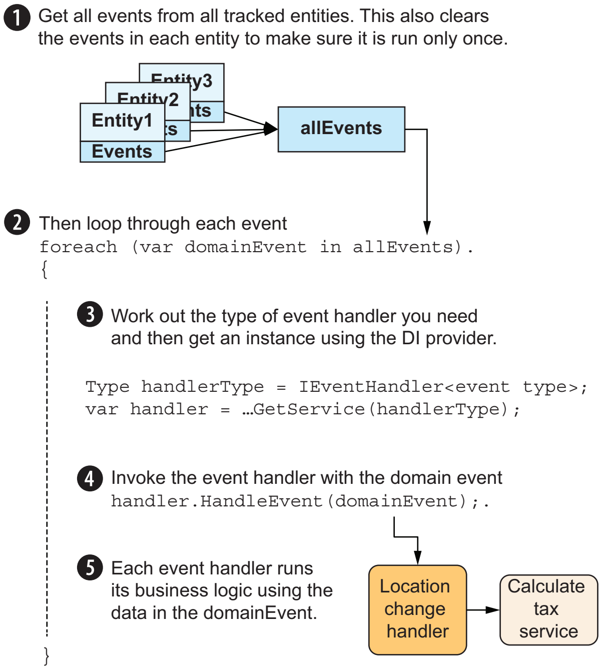


Figure 12.5 The Event Runner gathers the events from each tracked entity that has the `IEntityEvents` interface; then, for each event, it gets an instance of the matching event handler class and invokes the handler with the event as a parameter. Finally, each event handler runs its business logic, using the data found in the event.

The following listing shows the Event Runner code. This code is fairly complex, because the design of the Event Runner requires the use of generic classes.

Listing 12.5 The Event Runner that is called from inside the overridden `SaveChanges`

The Event Runner needs an interface so that you can register it with the DI.

```
public class EventRunner : IEventRunner
{
    private readonly IServiceProvider _serviceProvider;

    public EventRunner(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    public void RunEvents(DbContext context)
    {
        var allEvents  = context.
            ChangeTracker.Entries<IEntityEvents>()
            .SelectMany(x => x.Entity.GetEventsThenClear());

        foreach (var domainEvent in allEvents)
        {
            var domainEventType = domainEvent.GetType();
            var eventHandleType = typeof(IEventHandler<>)
                .MakeGenericType(domainEventType);
```

The Event Runner needs the ServiceProvider to get an instance of the event handlers.

Reads in all the events and clears the entity events to stop duplicate events

Loops through each event found

Gets the interface type of the matching event handler

Uses the DI provider to create an instance of the event handler and returns an error if one is not found

```
            var eventHandler =
                _serviceProvider.GetService(eventHandleType);
            if (eventHandler == null)
                throw new InvalidOperationException(
                    $"Could not find an event handler”)
```

Creates the EventHandlerRunner that you need to run the event handler

```
            var handlerRunnerType = typeof(EventHandlerRunner<>)
                .MakeGenericType(domainEventType);
            var handlerRunner = ((EventHandlerRunner)
                Activator.CreateInstance(
                    handlerRunnerType, eventHandler));

            handlerRunner.HandleEvent(domainEvent);
        }
    }
}
```

Uses the EventHandlerRunner to run the event handler

The following listing shows the `EventHandlerRunner` and `EventHandlerRunner<T>` classes. You need these two classes because the definition of an event handler is generic, so you can't call it directly. You get around this problem by creating a class that takes the generic event handler in its constructor and has a nongeneric method (the abstract class called `EventHandlerRunner`) that you can call.

Listing 12.6 The `EventHandlerRunner` class that runs the generic-typed event handler

```
internal abstract class EventHandlerRunner
{
    public abstract void HandleEvent
        (IDomainEvent domainEvent);
}
```

By defining a nongeneric method, you can run the generic event handler.

```
internal class EventHandlerRunner<T> : EventHandlerRunner
    where T : IDomainEvent
{
```

Uses the EventHandlerRunner<T> to define the type of the EventHandlerRunner

```
    private readonly IEventHandler<T> _handler;

    public EventHandlerRunner(IEventHandler<T> handler)
    {
        _handler = handler;
    }
```

The EventHandlerRunner class is created with an instance of the event handler to run.

```
    public override void HandleEvent
        (IDomainEvent domainEvent)
    {
        _handler.HandleEvent((T)domainEvent);
    }
}
```

Method that overrides the abstract class's HandleEvent method

12.4.6 Override SaveChanges and insert the Event Runner before SaveChanges is called

Next, you override `SaveChanges` and `SaveChangesAsync` so that the Event Runner is run before the base `SaveChanges` and `SaveChangesAsync` run. Any changes the event handlers make to entities are saved with the original changes that caused the events. This point is really important: both the changes made to entities by your nonevent code are saved with any changes made by your event handler code. If a problem occurs with the data being saved to the database (a concurrency exception was thrown, for example), neither of the changes would be written to the database, so the two types of entity changes—nonevent code changes and event-handler code changes—won't become CQRS out of step. The following listing shows how you inject the Event Runner via your application's DbContext constructor and then use that Event Runner inside the overridden `SaveChanges` method.

Listing 12.7 Your application's DbContext with `SaveChanges` overridden

```
public class DomainEventsDbContext : DbContext
{
    private readonly IEventRunner _eventRunner;

    public DomainEventsDbContext(
        DbContextOptions<DomainEventsDbContext> options,
        IEventRunner eventRunner = null)
        : base(options)
    {
        _eventRunner = eventRunner;
    }

    //… DbSet<T> left out


    public override int SaveChanges
        (bool acceptAllChangesOnSuccess)
    {
        _eventRunner?.RunEvents(this);
        return base.SaveChanges(acceptAllChangesOnSuccess);
    }

    //… overridden SaveChangesAsync left out
}
```

Holds the Event Runner that is injected by DI via the class's constructor

The constructor now has a second parameter DI fills in with the Event Runner.

You override SaveChanges so that you can run the Event Runner before the real SaveChanges.

Runs the Event Runner

Runs the base.SaveChanges

NOTE There are two version of `SaveChanges` and `SaveChangesAsync`, but you need to override only one of each of them. You need to override only the `int SaveChanges(bool acceptAllChangesOnSuccess)`, for example, because the `SaveChanges` with no parameters calls the `SaveChanges` with the `acceptAllChangesOnSuccess` parameter set to `true`.

12.4.7 Register the Event Runner and all the event handlers

The last part is registering the Event Runner and the event handlers with the DI provider. The Event Runner relies on the DI to provide an instance of your event handlers, using their interfaces; also, your application's DbContext needs the Event Runner injected by DI into the `IEventRunner` parameter of its constructor. When Event Runner and the event handlers are registered, along with any services that the event handlers need (such as the sales tax calculator service), the Event Runner will work. In this simple example, you can register the few classes and interfaces manually by using the following NET Core DI provider, as shown in the following listing.

Listing 12.8 Manually registering the Event Runner and event handlers In ASP.NET Core

```
public void ConfigureServices(IServiceCollection services)
{
    //… other registrations left out

    services.AddTransient<IEventRunner, EventRunner>();

    services.AddTransient<IEventHandler<LocationChangedEvent>,
            LocationChangedEventHandler>();
    services.AddTransient<IEventHandler<QuoteLocationChangedEvent>,
        QuoteLocationChangedEventHandler>();

    services.AddTransient<ICalcSalesTaxService,
        CalcSalesTaxService>();
}
```

Registers the Event Runner, which will be injected into your application's DbContext

You register interfaces/classes with the NET dependency injection provider—in this case, in a ASP.NET Core app.

Registers all your event handlers

You need to register any services that your event handlers will use.

Although manual registration works, a better way is to automate finding and registering the event handlers. Listing 12.9 shows an extension method that will register the Event Runner and all the event handlers in each assembly you provide. The following code snippet shows how it is called:

```
services.RegisterEventRunnerAndHandlers(
    Assembly.GetAssembly(
        typeof(LocationChangedEventHandler)));
```

The following listing shows the `RegisterEventRunnerAndHandlers` code.

Listing 12.9 Automatically registering the Event Runner and your event handlers

```
public static void RegisterEventRunnerAndHandlers(
    this IServiceCollection services,
    params Assembly[] assembliesToScan)
{
    services.AddTransient<IEventRunner, EventRunner>();
```

The method needs the NET Core's service collection to add to.

You provide one or more assemblies to scan.

Registers the Event Runner

```
    foreach (var assembly in assembliesToScan)
    {
        services.RegisterEventHandlers(assembly);
    }
}

private static void RegisterEventHandlers(
    this IServiceCollection services,
    Assembly assembly)
{
    var allGenericClasses = assembly.GetExportedTypes()
        .Where(y => y.IsClass && !y.IsAbstract
           && !y.IsGenericType && !y.IsNested);
    var classesWithIHandle =
        from classType in allGenericClasses
        let interfaceType = classType.GetInterfaces()
            .SingleOrDefault(y =>
                y.IsGenericType &&
                y.GetGenericTypeDefinition() ==
                    typeof(IEventHandler<>))
        where interfaceType != null
        select (interfaceType, classType);

    foreach (var tuple in classesWithIHandle)
    {
        services.AddTransient(
            tuple.interfaceType, tuple.classType);
    }
}
```

Calls a method to find and register event handler in an assembly

Finds and registers all the classes that have the IEventHandler<T> interface

Finds all the classes that could be an event handler in the assembly

Finds all the classes that have the IEventHandler<T> interface, plus the interface type

Registers each class with its interface

> **NOTE** The `RegisterEventRunnerAndHandlers` code won't register the `CalcSalesTaxService` service because it looks only for event handlers. But the `CalcSalesTaxService` class is a normal service—that is, a class with a nongeneric interface, like any other service. Chapter 5, especially section 5.7.3, shows how to register these types of services.

And that's it! You have added the domain events feature to your application, and you are ready to go. You will use domain events in chapter 15 as one way to improve database query performance by updating cache values when `Reviews` are added or removed. You can also see this feature in action in the Book App by clicking the SQL (cached) menu link.

12.5 Implementing an integration event system with EF Core

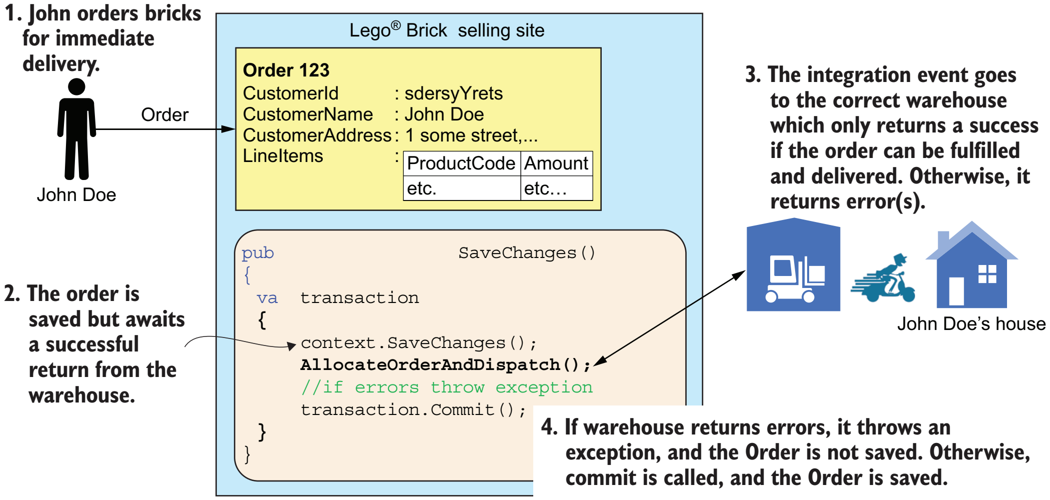
Now that you've seen how domain events work, we'll move on to integration events. Integration events are simpler to implement than domain events but harder to design because they work across bounded contexts (see section 12.2).

There are many ways to implement integration events, but this book is about EF Core, so this section concentrates on using an integration event within a database transaction in the `SaveChanges` method. The purpose is to ensure that the database is updated only if the integration event was successful.

I provided one example in section 12.2: combining the update of a SQL database with the corresponding update of a CQRS read-side database. This example works because the core tries to update a CQRS read-side database only if the SQL update succeeded, and it commits the SQL update only if the CQRS read-side database was successful; that way, the two databases contain the same data. You can generalize this example into two parts, both of which must work for the action to be successful:

- Don't send the integration event if the database update didn't work.
- Don't commit the database update unless the integration event worked.

Now let's implement some code that follows the integration event approach. As a simple example, suppose that you are building a new service that sends customers their orders of Lego bricks by courier on the same day. You don't want to disappoint your customers, so you must be sure that your warehouse has the items in stock and has a courier that can deliver the order immediately. The overall system is depicted in figure 12.6.



1. John orders bricks for immediate delivery.

Order 123
CustomerId : sdersyYrets
CustomerName : John Doe
CustomerAddress : 1 some street,...
LineItems :

ProductCode	Amount
etc.	etc…

```
pub                          SaveChanges()
{
  va  transaction
  {
      context.SaveChanges();
      AllocateOrderAndDispatch();
      //if errors throw exception
      transaction.Commit();
  }
}
```

2. The order is saved but awaits a successful return from the warehouse.

3. The integration event goes to the correct warehouse which only returns a success if the order can be fulfilled and delivered. Otherwise, it returns error(s).

4. If warehouse returns errors, it throws an exception, and the Order is not saved. Otherwise, commit is called, and the Order is saved.

Figure 12.6 Using an integration event to make sure that you can deliver an order before you save it. To implement this event, you override `SaveChanges` in your application's DbContext and handle any new `Order` by using a transaction. The `Order` is validated and saved; then an integration event sends the content of the order to the warehouse. If the warehouse has all the items needed and has a courier available to get the order to the customer, it returns a success message, which causes the `Order` to be committed to the database. If errors occur, the `Order` is not written to the database, and the errors are shown to the customer.

You have two options for detecting and handling your integration event in your application's DbContext:

- You inject the service directly into your application's DbContext, which works out for itself whether a specific event has happened by detecting the `State` of the entities. A second part is called only if the first method says that it needs to be called.
- You could use an approach similar to the Event Runner that you used for domain events, but a different event type is run within a transaction after the base `SaveChanges` is called.

In most cases, you won't have many integration events, so the first option is quicker; it bypasses the event system you added to the entity for the domain events and does its own detection of the event. This approach is simple and keeps all the code together, but it can become cumbersome if you have multiple events to detect and process.

The second option is an expansion of the Event Runner and domain events, which uses a similar creation of an integration event when something changes in the entity. In this specific case, the code will create an integration event when a new `Order` is created.

Both options require an event handler. What goes in the event handler is the business logic needed to communicate with the system/code and to understand its responses. The first option was used in the Lego example, where the event handler detected the event itself. You need to add two sections of code to implement this example:

- Build a service that communicates with the warehouse.
- Override `SaveChanges` (and `SaveChangesAsync`) to add code to create the integration event and its feedback.

12.5.1 Building a service that communicates with the warehouse

You know that integration events cross boundaries in an application. In the Lego example, the design suggests that the website where customers place orders is separate from the warehouse, which means some form of communication, maybe via some RESTful API. In this case, you would build a class that communicates with the correct warehouse and returns either a success or a series of errors. The following listing is one way to implement the code that communicates with the external warehouse.

Listing 12.10 The `Warehouse` event handler that both detects and handles the event

```
public class WarehouseEventHandler : IWarehouseEventHandler
{
    private Order _order;

    public bool NeedsCallToWarehouse(DbContext context)
    {
```

This method detects the event and returns true if there is an Order to send to the warehouse.

```
        var newOrders = context.ChangeTracker
            .Entries<Order>()
            .Where(x => x.State == EntityState.Added)
            .Select(x => x.Entity)
            .ToList();

        if (newOrders.Count > 1)
            throw new Exception(
                "Can only process one Order at a time");

        if (!newOrders.Any())
            return false;

        _order = newOrders.Single();
        return true;
    }

    public List<string> AllocateOrderAndDispatch()
    {
        var errors = new List<string>();

        //... code to communicate with warehouse

        return errors;
    }
}
```

Obtains all the newly created Orders

The business logic handles only one Order per SaveChanges call.

If there isn't a new Order, returns false

If there is an Order, retains it and returns true

This method will communicate with the warehouse and returns any errors the warehouse sends back.

Adds the code to communicate with the warehouse

Returns a list of errors. If the list is empty, the code was successful.

12.5.2 Overriding SaveChanges to handle the integration event

As stated earlier, you are using an integration event implementation that detects the event itself, rather than adding an event to the entity class, so the code inside the overridden `SaveChanges` and `SaveChangesAsync` is specific to the integration event. The following code listing shows the code to implement the Lego example.

Listing 12.11 DbContext with overridden `SaveChanges` and `Warehouse` event handler

```
public class IntegrationEventDbContext : DbContext
{
    private readonly IWarehouseEventHandler
        _warehouseEventHandler;

    public IntegrationEventDbContext(
        DbContextOptions<IntegrationEventDbContext> options,
        IWarehouseEventHandler warehouseEventHandler)
        : base(options)
    {
        _warehouseEventHandler = warehouseEventHandler;
    }

    public DbSet<Order> Orders { get; set; }
    public DbSet<Product> Products { get; set; }

    public override int SaveChanges
        (bool acceptAllChangesOnSuccess)
```

Holds the instance of the code that will communicate with the external warehouse

Injects the warehouse event handler via DI

Overrides SaveChanges to include the warehouse event handler

```
    {
        if (!_warehouseEventHandler.NeedsCallToWarehouse(this))
            return
                base.SaveChanges(acceptAllChangesOnSuccess);

        using(var transaction = Database.BeginTransaction())
        {
            var result =
                base.SaveChanges(acceptAllChangesOnSuccess);

            var errors = _warehouseEventHandler
                .AllocateOrderAndDispatch();

            if (errors.Any())
            {
                throw new OutOfStockException(
                    string.Join('.', errors));
            }

            transaction.Commit();
            return result;
        }
    }

    //… overridden SaveChangesAsync left out
}
```

If the event handler doesn't detect an event, it does a normal SaveChanges.

There is an integration event, so a transaction is opened.

Calls the base SaveChange to save the Order

Calls the warehouse event handler that communicates with the warehouse

If the warehouse returned errors, throws an OutOfStockException

Returns the result of the SaveChanges

If there were no errors, the Order is committed to the database.

NOTE When you use transactions in which the retry-on-failure option is enabled, you need to wrap the transaction in an execution strategy (see section 11.7.1).

12.6 Improving the domain event and integration event implementations

The code shown so far implements a fully running domain events and integration-events system that you can use, but it omits several features that would be useful. Async event handlers would be important features to add, for example. In this section, you explore some extra features that you might like to add to your event handling. Here are some of the additional features I found when I created a generalized event library called `EfCore.GenericEventRunner`:

- Generalizing events (events running before, during, and after the call to `SaveChanges`)
- Adding support for async event handlers
- Handling multiple event handers for the same event
- Handling event sagas, in which one event kicks off another event

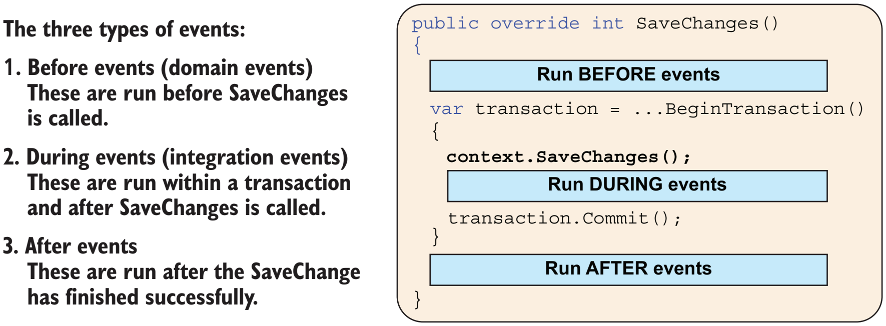
The following sections add the features in this list to the domain and integration designs you have worked on so far. The aim is to build a generalized event library that you can use in any application that events would help.

NOTE Full implementations of the new features are not provided due to space constraints. The aim is to show what sort of improvements you could add to the Event Runner. Links to the implementation in the `EfCore.GenericEventRunner` library are provided where appropriate.

12.6.1 Generalizing events: Running before, during, and after the call to SaveChanges

If you are going to build a library to handle events, it's worth looking at all the types of events you might want to handle. You have already seen the domain event and an integration event, but for the integration event example, you handcoded an integration event system because doing that was easier. But when you are willing to write a library, it's worth putting in the work to handle integration events too.

Another event type might be useful—one that runs when `SaveChanges` or `SaveChangesAsync` has finished successfully. You could send an email when you are sure that an `Order` has been checked and successfully added to the database. That example uses three event types, which I call *Before* (domain events), *During* (integration events), and *After* events (figure 12.7).



```
public override int SaveChanges()
{
  var transaction = ...BeginTransaction()
  {
    context.SaveChanges();
    transaction.Commit();
  }
}
```

Figure 12.7 A study of the various events around the call to `SaveChanges` or `SaveChangesAsync` suggests three important positions: 1. Before SaveChanges is called, which allows you to alter entities before they are saved; 2. within a transaction where `SaveChanges` has been called but the transaction hasn't been committed yet, allowing you to roll back the saved data if the outgoing event fails; and 3. after `SaveChanges/SaveChangesAsync` finishes successfully, which allows you to run code that is valid only if the data was successfully saved.

To implement the Before, During, and After event system, you must add two more Event Runners (see listing 12.5): one called within a transaction to handle the integration events, and one after `SaveChanges/SaveChangesAsync` has finished successfully (figure 12.7). You also need three event-handler interfaces—Before, During, and After—so that the correct event handler is run at the same time.

The full implementation is rather long and not repeated here. You can find the code in the `RunEventsBeforeDuringAfterSaveChanges` method that implements the Before, During, and After event system in the `EfCore.GenericEventRunner` library. See http://mng.bz/K4A0.

12.6.2 Adding support for async event handlers

In many of today's multiuser applications, async methods will improve scalability, so you need to have async versions of the event handlers. Adding an async method requires an extra event handler interface for an async event handler version. Also, the Event Runner code must be altered to find an async version of the event handler when the `SaveChangesAsync` is called. Listing 12.12 shows the updated `RunEvents` method in the Event Runner from listing 12.5 updated as an async `RunEventsAsync` method.

> **NOTE** To help you see the changes added to the version from listing 12.5, I've added comments for only the changed code.

Listing 12.12 The original `RunEvents` method updated to run async event handlers

```
public async Task RunEventsAsync(DbContext context)
{
    var allEvents  = context.
        ChangeTracker.Entries<IEntityEvents>()
        .SelectMany(x => x.Entity.GetEventsThenClear());

    foreach (var domainEvent in allEvents)
    {
        var domainEventType = domainEvent.GetType();
        var eventHandleType = typeof(IEventHandlerAsync<>)
            .MakeGenericType(domainEventType);

        var eventHandler =
            _serviceProvider.GetService(eventHandleType);
        if (eventHandler == null)
            throw new InvalidOperationException(
                "Could not find an event handler”);

        var handlerRunnerType =
            typeof(EventHandlerRunnerAsync<>)
            .MakeGenericType(domainEventType);
        var handlerRunner = ((EventHandlerRunnerAsync)
            Activator.CreateInstance(
                handlerRunnerType, eventHandler));

        await handlerRunner.HandleEventAsync(domainEvent);
    }
}
```

The RunEvent becomes an async method, and its name is changed to RunEventAsync.

The code is now looking for a handle with an async type.

Needs a async EventHandlerRunner to run the event handler

Is cast to a async method

Allows the code to run the async event handler

12.6.3 Handling multiple event handers for the same event

You might define more than one event handler for an event. Your `LocationChangedEvent`, for example, might have one event handler to recalculate the tax code and another event handler to update the company's map of ongoing projects. In the current implementations of the Event Runners, the .NET Core DI method `GetService` would throw an exception because it can return only one service. The solution is simple. Use the .NET Core DI method `GetServices` method and then loop through each event handler found:

```
var eventHandlers =
    _serviceProvider.GetServices(eventHandleType);
if (!eventHandlers.Any())
    throw new InvalidOperationException(
        "Could not find an event handler”);
foreach(var eventHandler in eventHandlers)
{
    //… use code from listing 12.5 that runs a single event handler
```

12.6.4 Handling event sagas in which one event kicks off another event

In my client's system, we found that one event could cause a new event to be created. The `LocationChangedEvent` event updated the `SalesTax`, which, in turn, caused a `QuotePriceChangeEvent`. These updates are referred to as *event sagas* because the business logic consists of a series of steps that must be executed in a certain order for the business rule to be completed.

Handling event sagas requires you to add a looping arrangement that looks for events being created by other events. The following listing shows the updated `RunEvents` method in the Event Runner from listing 12.5, with only the new looping code having comments.

Listing 12.13 Adding looping on events to the `RunEvents` method in the Event Runner

```
public void RunEvents(DbContext context)
{
    bool shouldRunAgain;
    int loopCount = 1;
    do
    {
        var allEvents = context.
            ChangeTracker.Entries<IEntityEvents>()
            .SelectMany(x => x.Entity.GetEventsThenClear());

        shouldRunAgain = false;
        foreach (var domainEvent in allEvents)
        {
            shouldRunAgain = true;

            var domainEventType = domainEvent.GetType();
            var eventHandleType = typeof(IEventHandler<>)
                .MakeGenericType(domainEventType);
```

Controls whether the code should loop around again to see whether there are any new events

Counts how many times the Event Runner loops around to check for more events

This do/while code keeps looping while shouldRunAgain is true.

shouldRunAgain is set to false. If there are no events, it will exit the do/while loop.

There are events, so shouldRunAgain is set to true.

```
                var eventHandler =
                    _serviceProvider.GetService(eventHandleType);
                if (eventHandler == null)
                    throw new InvalidOperationException(
                        "Could not find an event handler");

                var handlerRunnerType = typeof(EventHandlerRunner<>)
                    .MakeGenericType(domainEventType);
                var handlerRunner = ((EventHandlerRunner)
                    Activator.CreateInstance(
                        handlerRunnerType, eventHandler));

                handlerRunner.HandleEvent(domainEvent);
            }
            if (loopCount++ > 10)
                throw new Exception("Looped to many times");
        } while (shouldRunAgain);
    }
```

This check catches an event handler that triggers a circular set of events.

Stops looping when there are no events to handle

Summary

- A domain event class carries a message that is held inside an entity class. The domain event defines the type of event and carries event-specific data, such as what data has changed.
- Event handlers contain business logic that is specific to a domain event. Their job is to run the business logic, using the domain event data to guide what it does.
- The domain events version of the `SaveChanges` and `SaveChangesAsync` methods captures all the domain events in the tracked-entities classes and then runs matching event handlers.
- The integration events versions of the `SaveChanges` and `SaveChangesAsync` methods use a transaction to ensure that both the database and integration event handler succeed before the database is updated. This requirement allows you to synchronize two separate parts of your application.
- In section 12.4, you implemented a domain events system by creating domain event classes, event handlers, and an Event Runner. Using these three parts and overriding the `SaveChanges` and `SaveChangesAsync` methods allows you to use domain events in your applications.
- In section 12.5, you updated the domain events system in section 12.4 to handle integration events, which requires calling an external service within a database transaction.
- In section 12.5, you added enhancements to the Event Runner, such as supporting event handlers that use async methods.

13

Domain-Driven Design and other architectural approaches

This chapter covers

- Three architectural approaches applied to the part 3 Book App
- The differences between normal and DDD-styled entity classes
- Eight ways you can apply DDD to your entity classes
- Three ways to handle performance problems when using DDD

Although this book is about EF Core, I want to include something about software architecture, as readers of the first edition of this book found it useful. You were introduced to the layered architecture in part 1. Now, in part 3, in which we are building a much more complex Book App, I'll change the Book App's software architecture to improve the separation of parts of the code and make the entity classes' data more secure.

The most important of these architectural changes is swapping to the use of Domain-Driven Design (DDD), from Eric Evan's book of the same name (Addison-Wesley Professional, 2003). The first version of EF Core added one new feature that

EF6 didn't have—backing fields—and that new feature makes following the DDD approach possible. Since the first edition of this book came out, I have used DDD a lot, both in client applications and in building libraries to handle DDD entity classes.

I share my experiences and code to help you learn how DDD can help you with developing applications. The use of DDD on entity classes is broken into eight sections so that you can understand how each part of DDD helps improve the application. Finally, I cover ways to deal with slow performance on updates when there are lots of entries in a relationship using DDD-styled entities.

13.1 A good software architecture makes it easier to build and maintain your application

One problem with building software applications is that they can become harder and harder to develop as they get bigger because you need to alter the existing code to add a new feature. All sorts of issues arise, such as finding and understanding the existing code, deciding on the best way to add the new feature, and making sure that you haven't broken anything.

The architecture you choose for your application is one of the ways you can make it easier to write and update the code. Software principles, such as Separation of Concerns (SoC) and DDD, also play a part in making the application easier to fix and extend. A good application design provides a pattern that guides your coding, as well as some rules that encourage you down a good development route.

Neal Ford coined the term *evolutionary architecture* in *Building Evolutionary Architectures* (O'Reilly, 2017) to recognize the fact that, nowadays, applications need to grow and change to keep providing the user the right features and the best experience. In section 13.2, I describe the architecture/software principles I chose for the Book App in part 3—principles that make adding features much easier.

13.2 The Book App's evolving architecture

In parts 1 and 2, the Book App uses one database containing about 50 books. Its purpose is to provide a simple application to show how various EF Core features can be used in a real application. Therefore, the layered architecture (see section 5.2) is a good fit.

The part 1 and 2 Book App is small enough that I could have put all the code inside the ASP.NET Core application, but I didn't. Instead, I used a layered architecture with most of the interesting code in the data layer and the service layer. Here is why I didn't put all the code in the ASP.NET Core application:

- It would be harder to find something inside the single ASP.NET Core project.
- It would be harder to test because the code would be hardcoded to ASP.NET Core.

In addition to using the layered architecture, I used the SoC software principle (see section 2.7) to break the software into smaller parts because I knew that I would be adding features. Here are two of many examples that show why SoC is so useful:

- I build the main Book App query in sections (Select, Filter, Sort, Page) to make it easier to understand, test, and refactor.
- In part 2, I moved the EF Core Fluent API code into per-entity configuring classes to made it easier to find, show, and refactor a configuration for a specific entity class.

The layered architecture works well in parts 1 and 2, where the focus is on how EF Core works, but part 3 focuses on performance-tuning your EF Core applications. This part uses multiple databases (SQL and Cosmos DB), two ways to access the database (EF Core and Dapper), and multiple performance-tuning techniques. This means there are different versions of the code to display the books in the Book App's database. To manage all these different query approaches and to show you some new ways to design applications, I am adopting three new architectural/software principles to build part 3's Book App:

- A *modular monolith* approach, which enforces SoC by using .NET projects
- DDD principles, both architecturally and on the entity classes
- A *clean architecture*, as described by Robert C. Martin (known as Uncle Bob)

I provide introductions to these three architectural/software principles in sections 13.2.1 through 13.2.3. Figure 13.1 provides an overall view of the part 3 Book App architecture.

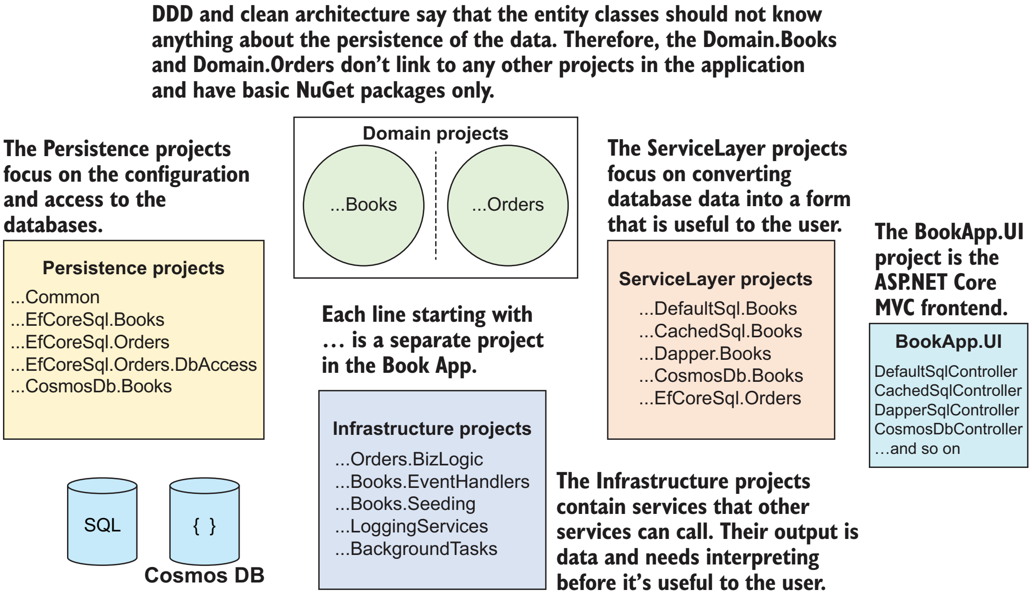


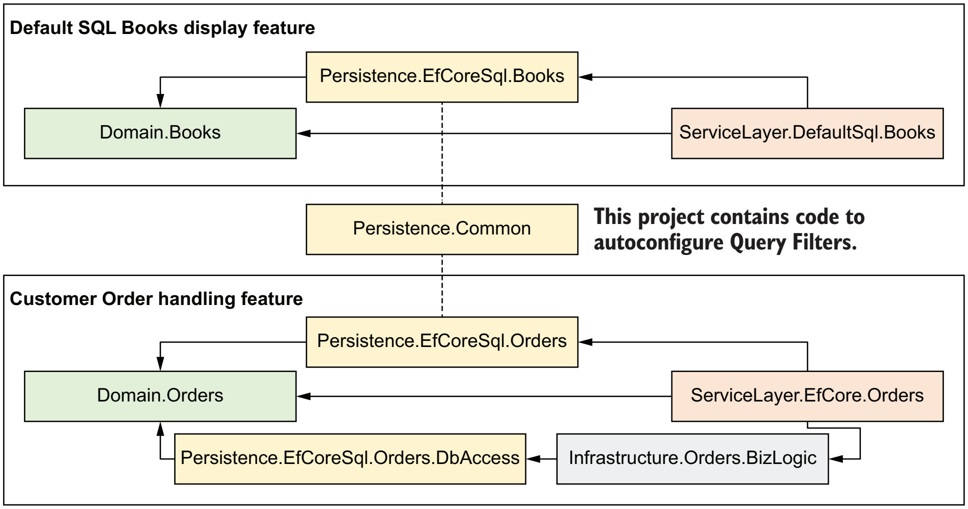
Figure 13.1 An overview of how the part 3 Book App is constructed, with five groups: Domain, Persistence, Infrastructure, ServiceLayer, and the ASP.NET Core BookApp.UI. This structure is designed to enforce the SoC rules by limiting what the developer can access from a project. The aim is to break the code into discrete "features" to make it easier to understand and refactor the code.

13.2.1 Building a modular monolith to enforce the SoC principles

My goal is to make the application modular. By *modular*, I mean that the code for a specific feature, such as displaying the books, is easy to identify, and has no links to feature code that it doesn't need. I achieve this goal by creating small projects that implement code for a specific job and link only to projects that contain the code required for a specific feature (see figure 13.2).

The layered architecture in parts 1 and 2 has a service layer project containing code for lots of Book App features mixed together, including code for displaying books, creating an order, seeding the database, and running background services. This architecture becomes a mass of interconnected code (known as a *ball of mud*) and is hard to refactor. In part 3's modular monolith design, each of these features has its own project, which makes it (nearly) impossible for features to share code except via a lower layer.

It's not obvious in figure 13.1, but various projects are linked to create features that are as self-contained as possible. Figure 13.2 shows two features, one for handling books and one for handling user's orders. These two features are separate (apart from the Persistence.Common project) and have only one shared project.



This project contains code to autoconfigure Query Filters.

Figure 13.2 The modular monolith approach follows the SoC principle by breaking the application's code into small projects, each of which has a specific job to do. This approach also follows the clean architecture layer approach with four layers: Domain, Persistence, Infrastructure, and ServiceLayer, as shown in the names in each project. Some of the project names, such as ServiceLayer.DefaultSql .Books, may not make sense now, but they will become clear when you get to chapters 15 and 16.

The main aim of applying a modular approach to a monolith is to overcome the typical "ball of mud" monolith structure. If it helps, you can think of features in a modular monolith architecture as being the same as microservices, but the features communicate by means of simple method calls, whereas microservices communicate over some sort of protocol, with possible failures.

DEFINITION The *microservices architecture* arranges an application as a collection of loosely coupled services that communicate by using some form of message passing, such as HTTP messages.

Using a modular approach provides numerous benefits:

- It's easy to see all the code involved in a certain command, such as displaying a book.
- A change in a feature should have no effect, or minimal effect, on other features.
- You should be able to move a feature to another application, such as a microservice, with minimal problems for the rest of the application.

At the same time, you want your code to be DRY (don't repeat yourself), so there will be some shared code, such as the Persistence.Common project shown in figure 13.2. But I recommend that you don't create lots of generic code that is used in lots of projects, because changing that generic code could break code in another feature. If your generic code is that good, turn it into a library. (That's what I do.)

This section ends our overview of the overall architecture of the part 3 Book App. The rest of this chapter looks at DDD, because DDD can profoundly affect how you build and use your EF Core code and entity classes.

NOTE Because I made significant changes in chapters 15 and 16, I added a new section 13.8 to this chapter to share my experiences of using the three architectural approaches as I added new features that doubled the size of the Book App from its chapter 13 start.

13.2.2 Using DDD principles both architecturally and on the entity classes

DDD details many approaches for defining, building, and managing software applications. But I want to point out three DDD principles in particular:

- The part 3 Book App's entity classes follow the DDD rules for what DDD calls *entities* (and what EF Core calls *entity classes*). The main rule is that a DDD entity is in total control of the data in that entity: all the properties are made read-only, and there are constructors/methods to create/update the entities' data. Giving the entity total control of its data makes your entity classes much more powerful; each entity class has clearly defined constructors/methods for the developer to use.
- DDD says that entities, which contain both data and domain (business) logic, should not know anything about how the entities are persisted to a database (mentioned in section 4.3.1). I talk more about that topic in section 13.2.3, which covers the use of clean architecture.

- DDD talks about *bounded contexts*, which separate your application into distinct parts. The idea is to create bounded contexts that are separate so that they are easier to understand, and then set up clearly defined communication between the bounded contexts. In the part 3 Book App, I created a bounded context around the display and editing of books, and another bounded context covering the ordering of books.

13.2.3 Applying a clean architecture as described by Robert C. Martin

Clean architecture is a software design approach that separates the different parts of your code into layers, arranged as a series of rings like those of an onion. These layers, plus some rules, are there to organize your code so that your entity classes and business logic are kept isolated from higher layers in the rings. I couldn't get all the projects into figure 13.1 by using a series of rings, but the arrangement of the Book App does follow the clean-architecture approach.

NOTE Here is a link to the clean architecture definition written by Robert C. Martin (Uncle Bob): http://mng.bz/9N71.

Clean architecture incorporates several other architectures, including hexagonal and onion. The purpose of this architecture is to define rules that state how different layers communicate. Clean architecture has a dependency rule, for example, stating that code in inner rings can't explicitly link to outer rings. Clean architecture matches DDD's rule to separate the entities from the database (DDD persistence) code and helps keep the code separated in rings, which I have defined as Domain, Persistence, Infrastructure, ServiceLayer, and the ASP.NET Core BookApp.UI.

13.3 Introduction to DDD at the entity class level

DDD is a massive topic with many facets, but this book is about EF Core. Therefore, I focus on EF Core entity classes, keeping the database parts out of DDD entities and using DDD's bounded context pattern to define how your code accesses the database.

With its focus on patterns and designs that make building applications better and more relevant, Eric Evans's 2003 *Domain-Driven Design* is a pivotal book in software development. What *Domain-Driven Design* doesn't do is give you a detailed set of steps on how you should implement with DDD. I think that is good, because if the book had given detailed steps, it would be outdated by now, but the principles are just as relevant today.

Because the Evans book didn't include detailed implementation plans, however, lots of people have come up with different ways to implement DDD. At one end, the entity classes are carefully crafted to contain only business code; all database parts, such as primary and foreign keys, are hidden. At the other end are designs in which—due to the developer's desire to move all the business code inside the entity class—the entity class contains reads and writes to the database. I'm going to describe the approach that most people take to DDD, plus some code to reduce the code you must write. (I don't show the strict DDD style, with all keys hidden, but you can follow that approach with shadow properties.)

First, however, let's look at the major changes between a normal entity class and a DDD entity class, which will help you understand the differences between what has been shown in this book so far and how DDD works. We'll start with a simple update to a `Book`'s `PublishedOn` property as an example of updating the database; you first saw this update in section 3.3. The code is trivial, making it easier to see the differences in the two approaches. Figure 13.3 shows the original, non-DDD design on the left and the DDD design on the right.

The DDD version in figure 13.3 requires slightly more code, but as you will see in section 13.4, that extra code lets these DDD entity classes become much more valuable parts of your code. Nevertheless, with potentially hundreds of creates and updates, those few extra lines add up, which is why I am always trying to find ways to reduce the code I need to write (see section 13.5).

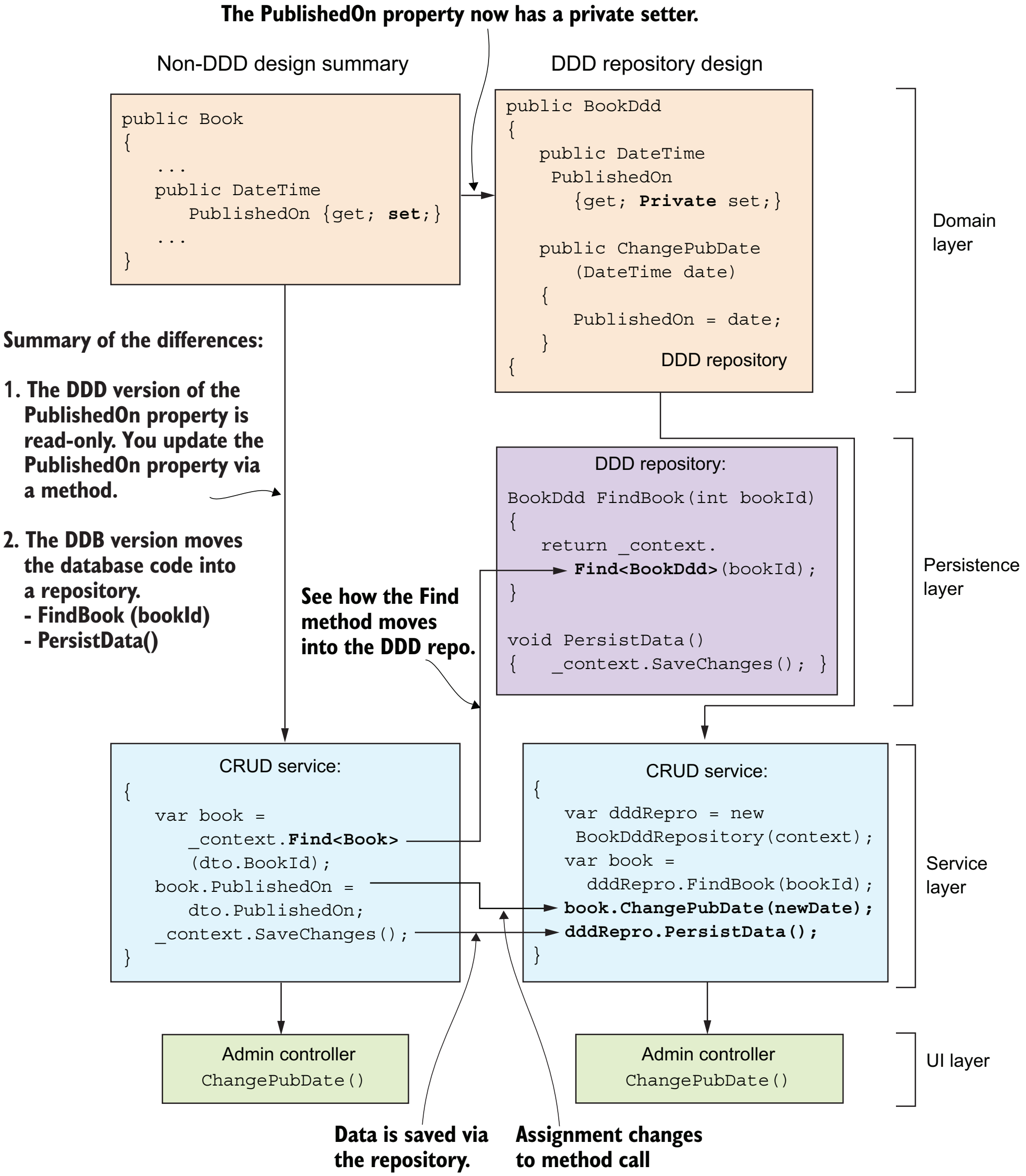
13.4 Altering the Book App entities to follow the DDD approach

In this section, you are going to change the `Book` entity class and associated entity classes to follow the DDD approach. You will make these changes in stages so that you can understand how and why you are making them. Here are the steps in the process of changing your code to the DDD approach:

- Changing the properties in the `Book` entity to read-only
- Updating the `Book` entity properties via methods in the entity class
- Controlling how the `Book` entity is created
- Understanding the differences between an entity and a value object
- Minimizing the relationships between entity classes
- Grouping entity classes (DDD name: *aggregates*)
- Deciding when the business logic shouldn't be run inside an entity
- Applying DDD's bounded context to your application's DbContext

13.4.1 Changing the properties in the Book entity to read-only

DDD says that the entity class is in charge of the data it contains; therefore, it must control how the data is created or changed. For the entity class to control its data, you make all the properties in the entity read-only. After that, a developer can set the data in the entity class only via the class's constructor (section 13.4.3) or via the methods in the entity class (section 13.4.2). The entity can ensure that it is always in a valid state. With a non-DDD `Book` entity class, I could create a `Book` without an `Author`, but the business rules state that a valid book has at least one `Author`. To get this level of control, you need to make all the properties read-only so that a developer must use the defined methods/constructors. Listing 13.1 shows the `Book` entity class with the properties changed to read-only.



The PublishedOn property now has a private setter.

```
public Book
{
   ...
   public DateTime
      PublishedOn {get; set;}
   ...
}
```

```
public BookDdd
{
   public DateTime
    PublishedOn
      {get; Private set;}

   public ChangePubDate
      (DateTime date)
   {
      PublishedOn = date;
   }
{
```

Summary of the differences:

1. **The DDD version of the PublishedOn property is read-only. You update the PublishedOn property via a method.**
2. **The DDB version moves the database code into a repository.**
   - **- FindBook (bookId)**
   - **- PersistData()**

See how the Find method moves into the DDD repo.

```
BookDdd FindBook(int bookId)
{
   return _context.
      Find<BookDdd>(bookId);
}

void PersistData()
{   _context.SaveChanges(); }
```

```
{
   var book =
      _context.Find<Book>
      (dto.BookId);
   book.PublishedOn =
      dto.PublishedOn;
   _context.SaveChanges();
}
```

```
{
   var dddRepro = new
    BookDddRepository(context);
   var book =
     dddRepro.FindBook(bookId);
   book.ChangePubDate(newDate);
   dddRepro.PersistData();
}
```

Data is saved via the repository.

Assignment changes to method call

Figure 13.3 Comparing the non-DDD design for updating a `Book`'s publication date in the Book App (left) with the DDD design (right). The code required for the update has the same parts, but the DDD version moves all the EF Core code to the Persistence layer. If you also "hide" the application's DbContext in the DDD version, you can ensure that the developer can access the database only via the DDD repository.

Listing 13.1 Making the `Book` entity class's properties read-only

```
public class Book
{
    public int BookId { get; private set; }
    public string Title { get; private set; }
    //… other non-collection properties left out

    private HashSet<Review> _reviews;
    public IReadOnlyCollection<Review>
        Reviews => _reviews?.ToList();

    private HashSet<BookAuthor> _authorsLink;
    public IReadOnlyCollection<BookAuthor>
        AuthorsLink => _authorsLink?.ToList();
    //… other collection properties left out
}
```

Noncollection properties have their setter set to private.

A collection is stored in a backing field.

The property collection returns the appropriate backing fields as read-only collections.

WARNING If you are using AutoMapper, it will ignore the private access scope on your setter and update the property, which is *not* what you want to happen when using DDD. To stop this update, you need to add the `IgnoreAllPropertiesWithAnInaccessibleSetter` method after the call to AutoMapper's `CreateMap<TSource,TDestination>` method.

13.4.2 Updating the Book entity properties via methods in the entity class

With all the properties converted to read-only, you need another way to update the data inside an entity. The answer is to add methods inside the entity class that can update the properties. I call these methods *access methods*. Creating access methods is extra work, so why does DDD say you should do this? Here are the main benefits:

- You can use an entity like a black box. The access methods and constructors are its API: it's up to the entity to make sure that the data inside the entity is always in a valid state.
- You can put your business rules in the access method. The method can return errors to users so that they can fix the problem and retry, or for a software problem, you can throw an exception.
- If there isn't a method to update a specific property, you know that you're not allowed to change that property.

Some simple methods only change a property, but many methods contain the business rules for your application. One example in the `Book` entity is adding and removing a promotional price. In the part 3 `Book` entity class, you replace the `PriceOffer` entity class with two methods that run the business rules for adding and removing a promotional price. The rules are

- The sale price of a book is contained in the `ActualPrice` property.
- The full price of a book is contained in the `OrgPrice` property.

- The `PromotionalText` property should be `null` if there is no promotion, but it must have the promotion message if there is a promotion.

It would be easy for someone to disobey these rules, but turning the rules into an access method means that no one can get them wrong. Also, the rules are in one place, so they're easy to change if necessary. These access methods are some of DDD's most powerful techniques.

Listing 13.2 shows the `AddPromotion` and `RemovePromotion` access methods in the Book entity. These methods ensure that the rules for adding and removing a promotional price are followed.

NOTE The `IStatusGeneric` interface and `StatusGenericHandler` class come from a small open source NuGet library called `GenericServices.StatusGeneric`, which I use in many of my own libraries and applications.

Listing 13.2 Example of a DDD access method that contains business logic/validation

```
public IStatusGeneric AddPromotion(
    decimal actualPrice, string promotionalText)
{
    var status = new StatusGenericHandler();
    if (string.IsNullOrWhiteSpace(promotionalText))
    {
        return status.AddError(
            "You must provide text to go with the promotion.",
            nameof(PromotionalText));
    }

    ActualPrice = actualPrice;
    PromotionalText = promotionalText;

    return status;
}

public void RemovePromotion()
{
    ActualPrice = OrgPrice;
    PromotionalText = null;
}
```

The AddPromotion returns a status. If that status has errors, the promotion is not applied.

The parameters came from the input.

Creates a status that is successful unless errors are added to it

Ensures that the promotionalText has some text in it

The AddError method adds an error and returns immediately.

The error contains a user-friendly message and the name of the property that has the error.

If no errors occur, the ActualPrice and PromotionalText are updated.

The status, which is successful, is returned.

This removes an existing promotion. Because there are no possible errors it returns void.

Removes the promotion by resetting the ActualPrice and the PromotionalText

NOTE The name of the property in the `AddError` method in listing 13.2 is `PromotionalText`, not `promotionalText`, because we are providing the name of the property that the ASP.NET Core frontend was using when it called the `AddPromotion` method.

13.4.3 Controlling how the Book entity is created

In line with the DDD approach, in which the entity controls the setting of data in it, you need to think about the creation of an entity. As far as I know, Eric Evans doesn't define this process, but creating an entity class is an important issue, especially as all the properties are read-only. Therefore, you need to provide at least one constructor or a static create factory method for a developer to use to create a new instance of the entity.

In the `Book` entity class, it's possible to create an invalid instance, because the business rules state that a `Book`'s `Title` must not be empty and that there should be at least one `Author`. A constructor can't return errors, so you create a static create factory method that returns a status containing errors if the `Book`'s `Title` is empty or if no `Author` is provided. If there aren't any errors, the status contains a `Result` property containing the newly created `Book`, as shown in the following listing.

Listing 13.3 The static create factory to create a valid `Book` or return the errors

```
private Book() { }

public static IStatusGeneric<Book> CreateBook(
    string title, DateTime publishedOn,
    decimal price,
    ICollection<Author> authors)
{
    var status = new StatusGenericHandler<Book>();
    if (string.IsNullOrWhiteSpace(title))
        status.AddError(
            "The book title cannot be empty.");

    var book = new Book
    {
        Title = title,
        PublishedOn = publishedOn,
        OrgPrice = price,
        ActualPrice = price,
    };
    if (authors == null)
        throw new ArgumentNullException(nameof(authors));

    byte order = 0;
    book._authorsLink = new HashSet<BookAuthor>(
        authors.Select(a =>
            new BookAuthor(book, a, order++)));

    if (!book._authorsLink.Any())
        status.AddError(
            "You must have at least one Author for a book.");

    return status.SetResult(book);
}
```

Creating a private constructor means that people can't create the entity via a constructor.

The static CreateBook method returns a status with a valid Book (if there are no errors).

These parameters are all that are needed to create a valid Book.

Creates a status that can return a result—in this case, a Book

Adds an error. Note that it doesn't return immediately so that other errors can be added.

Sets the properties

The authors parameter, which is null, is considered to be a software error and throws an exception.

Creates the BookAuthor class in the order in which the Authors have been provided

Sets the status's Result to the new Book instance. If there are errors, the value is null.

If there are no Authors, add an error.

For simple entity classes, you can use a public constructor with specific parameters, but any entities that have business rules and return error messages should use a static factory in the entity class.

13.4.4 Understanding the differences between an entity and a value object

DDD talks about an *entity* (the `Book` entity being an example), but it also talks about a *value object*. The difference is what uniquely defines an instance of each. Eric Evans says, "Tracking the identity of entities is essential," but "Make [value objects] express the meaning by the attributes [properties] it conveys" (*Domain-Driven Design*, p. 98–99). Here are two examples that might help:

- An entity isn't defined by the data inside it. I expect that more than one person named John Smith has written a book, for example. Therefore, the Book App would need a different `Author` entity for each author named John Smith.
- A value object is defined by the data inside it. If I have an address to send an order to, and another address with the same road, city, state, zip code, and country, was created, the two instances of the address are said to be equal.

From an EF Core perspective, a DDD entity is an EF Core entity class, which is saved to the database with some form of primary key. The primary key ensures that the entity is unique in the database, and when EF Core returns a query including entity classes (and the query doesn't include any form of the `AsNoTracking` method), it uses a single instance for each entity class that has the same primary key (see section 6.1.3).

You can implement a value object by using EF Core's owned type (see section 8.9.1). The main form of an owned type is a class with no primary key; the data is added to the table it is included in.

NOTE The Book App doesn't include any value objects, so I can't use it as an example. Please look at listing 8.15 for a good example of using owned types in an entity class.

13.4.5 Minimizing the relationships between entity classes

Eric Evans says, "It is important to constraint relationships as much as possible" (*Domain-Driven Design*, p. 83). He goes on to say that added two-way relationships between entities mean you need to understand both entities when working on either entity, which makes the code harder to understand. His recommendation (and mine) is to minimize the relationships. A `Book`, for example, has a navigational property of all the `Reviews` for a `Book`, but the `Review` does not have a navigational property back to the `Book` (see section 8.2).

It's easy to minimize navigational relationships between entity classes. In section 8.2, I look at the relationships between the `Book` entity class and the `Review` entity class. I concluded that the `Book` entity needed a navigational collection of `Reviews` linked to it, but the `Review` entity didn't need a navigational link back to the `Book` entity. In other words, understanding the `Book` entity requires some idea of what the `Review` entity does, but when dealing with the `Review` entity, I had to understand only what the `Review` entity does.

13.4.6 Grouping entity classes

Another important DDD pattern called *aggregates* offers some guidance on handling related entities. The aggregates principle says that you should group entities that can be considered to be "one unit for the purpose of data changes" (*Domain-Driven Design*, p. 126). One of the entities in an aggregate is the *root aggregate*, and any changes in the other aggregates are made via this root aggregate.

Figure 13.4 shows aggregate entities around the DDD version of the `Book` entity class used in the Book App. Any changes to the `Review` or `BookAuthor` entities linked to a `Book` entity can be changed only via access methods or constructors in the `Book` entity. The `Author` entity is outside the `Book` aggregate because it can be linked to multiple `Book`s.

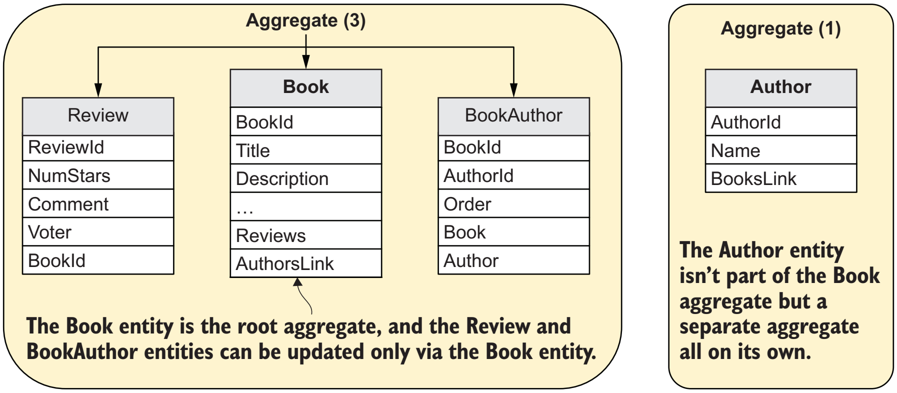


Figure 13.4 DDD's aggregates concept groups entities that can be managed as though they were one group of data. One of the entities is the root aggregate (the Book entity in the left aggregate and Author in the right aggregate). All updates to the `Book`'s `Reviews` or `BookAuthor` entities are done via the `Book` entity. This technique reduces the amount of entities you need to deal with and allows the root entity to ensure that all the other aggregates are set up correctly (that a `Book` has at least one `BookAuthor` link, for example).

> **NOTE** The `BookAuthor` entity in figure 13.4 breaks Evans's DDD aggregate rule because a nonroot aggregate should not be referenced from outside the aggregate. (The `Author` entity has a backlink to the `BookAuthor` entity.) But the `BookAuthor` entity contains book-specific data: the `Order` property, which defines the order in which the `Authors` should be cited. These features of the `BookAuthor` entity make it an aggregate of the `Book` entity.

The aggregate rule simplifies the handling of entities classes because one root entity can handle multiple aggregates in its group. Also, the root entity can validate that the other, nonroot aggregates are set up correctly for the root aggregate, such as the `Book` create factory's checking that there is at least one `Author` for a `Book` entity.

This rule also calls for using an access method in the `Book` entity to add, update, or remove `Review` entities' links to the `Book` entity instance. The following listing shows the two access methods for adding or removing `Reviews`.

Listing 13.4 The access methods that control the aggregate entity `Review`

```
public void AddReview(int numStars,
    string comment, string voterName)
{
    if (_reviews == null)
        throw new InvalidOperationException(
            "Reviews collection not loaded");

    _reviews.Add(new Review(
        numStars, comment, voterName));
}

public void RemoveReview(int reviewId)
{
    if (_reviews == null)
        throw new InvalidOperationException(
            "Reviews collection not loaded");

    var localReview = _reviews.SingleOrDefault(
        x => x.ReviewId == reviewId);

    if (localReview == null)
        throw new InvalidOperationException(
            "The review wasn't found");

    _reviews.Remove(localReview);
}
```

Adds a new review with the given parameters

This code relies on the _reviews field to be loaded, so it throws an exception if it isn't.

Creates a new Review, using its internal constructor

Removes a Review, using its primary key

Finds the specific Review to remove

Not finding the Review is considered to be a software error, so the code throws an exception.

The found Review is removed.

One additional change you make is marking the `Review` entity class's constructor as `internal`. That change stops a developer from adding a `Review` by creating an instance outside the `Book` entity.

13.4.7 Deciding when the business logic shouldn't be run inside an entity

DDD says that you should move as much of your business logic inside your entities, but the DDD aggregates rule says that the root aggregate should work only with other entities in the aggregate group. If you have business logic that includes more than one DDD aggregate group, you shouldn't put (all) the business logic in an entity; you need to create some external class to implement the business logic.

An example of a situation that requires more than one aggregate group in the business logic is processing a user's order for books. This business logic involves the `Book` entity, which is in the `Book/Review/BookAuthor` aggregate group, and the `Order/LineItem` aggregate group.

You saw a solution to the order-for-books problem in section 4.4.3. The DDD version uses similar code, but the final stage of building the `Order` is carried in a static factory inside the `Order` entity because the `Order` is the root aggregate in the `Order/LineItem` aggregate group. The following listing shows the external business class called `PlaceOrderBizLogic`.

NOTE Because you saw some of this code in listing 4.2, I left out similar parts of the code. The purpose is to focus on the changes in the DDD parts, especially creating the `Order` via an `Order` static factory.

Listing 13.5 `PlaceOrderBizLogic` class working across `Book` and `Order` entities

```
public async Task<IStatusGeneric<Order>>
    CreateOrderAndSaveAsync(PlaceOrderInDto dto)
{
    var status = new StatusGenericHandler<Order>();

    if (!dto.AcceptTAndCs)
    {
        return status.AddError("accept T&Cs…");
    }
    if (!dto.LineItems.Any())
    {
        return status.AddError("No items in your basket.");
    }

    var booksDict = await _dbAccess
        .FindBooksByIdsAsync
            (dto.LineItems.Select(x => x.BookId));

    var linesStatus = FormLineItemsWithErrorChecking
        (dto.LineItems, booksDict);
    if (status.CombineStatuses(linesStatus).HasErrors)
        return status;

    var orderStatus = Order.CreateOrder(
        dto.UserId, linesStatus.Result);

    if (status.CombineStatuses(orderStatus).HasErrors)
        return status;

    await _dbAccess.AddAndSave(orderStatus.Result);

    return status.SetResult(orderStatus.Result);
}
```

This method returns a status with the created Order, which is null if there are no errors.

The PlaceOrderInDto contains a TandC bool, and a collection of BookIds and number of books.

This status is used to gather errors, and if there are no errors, the code returns an Order.

Validate the user's input

This method creates a list of bookIds and numbers of books (see end of listing 4.2).

The _dbAccess contains the code to find each book (see listing 4.3)

If any errors were found while checking each order line, returns the error status

Calls the Order static factory. It is the Order's job to form the Order with LineItems.

Again, any errors will abort the Order and return errors.

The _dbAccess contains the code to add the Order and call SaveChangesAsync.

Returns a successful status with the created Order entity

The biggest change from the code in chapter 4 is that the `Order` entity takes over the final stage of building the `Order`. The following listing shows the `Order` static factory method.

Listing 13.6 This static factory creates an `Order` with the `LineItems`, with error checks

```
public static IStatusGeneric<Order> CreateOrder
    (Guid userId,
    IEnumerable<OrderBookDto> bookOrders)
{
    var status = new StatusGenericHandler<Order>();
    var order = new Order
    {
        UserId = userId,
        DateOrderedUtc = DateTime.UtcNow
    };

    byte lineNum = 1;
    order._lineItems = new HashSet<LineItem>(
        bookOrders
        .Select(x => new LineItem( x, lineNum++)));

    if (!order._lineItems.Any())
        status.AddError("No items in your basket.");

    return status.SetResult(order);
}
```

The OrderBookDto lives in the Order domain and carries the info that the Order needs.

This static factory creates the Order with lineItems.

The Order uses the UserId to show orders only to the person who created it.

Creates a status to return with an optional result of Order

Sets the standard properties in an order

Creates each of the LineItems in the same order in which the user added them

Double-checks that the Order is valid

Returns the status with the Order. If there are errors, the status sets the result to null.

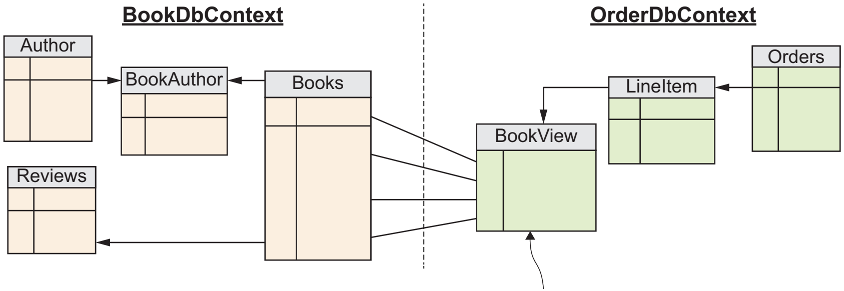
13.4.8 Applying DDD's bounded context to your application's DbContext

In section 13.2.2, I said that bounded contexts "separate your application into distinct parts" and that bounded contexts have "clearly defined communication." In figure 13.1, you saw two Persistence projects, `Persistence.EfCoreSql.Books` and `Persistence.EfCoreSql.Orders`, which are independent. But code previously described for placing a user's order needed information about the `Book`, so how do we manage this situation?

In this specific case, the solution is to use an SQL View in the `Order`'s DbContext that maps to the Books table in the `Book`'s DbContext, as shown in figure 13.5. That way, you can make the `Persistence.EfCoreSql.Books` and `Persistence.EfCoreSql.Orders` independent while letting both access the database data.

Using an SQL View is an excellent solution in this case because it follows many of the DDD rules. First, the `BookView` contains only the data that the `Orders` side needs, so the developer isn't distracted by irrelevant data. Second, when an entity class is configured as a View, EF Core marks that entity class as read-only, enforcing the DDD rule that only the `Books` entity should be able to change the data in the Books table.

NOTE Another benefit is that a class mapped to an SQL View won't add migration code to alter that table. You can apply EF Core migrations from both the



The OrderDbContext has a class called BookView, which is configured as an SQL View mapped to specific columns in the Books table.

Figure 13.5 To follow the DDD bounded-context approach, the Domain.Books should be independent of the Domain.Orders. But at the database level, both Domains need a `Book` entity. The solution in this case is to create a `BookView` entity in the Domain.Orders that contains only the specific properties to create and display an order. Then we configure the `BookView` class as an SQL View mapped to the Books table.

`BookDbContext` and the `OrderDbContext` contexts to the database, and only the `BookDbContext`'s `Book` entity will affect the Books table (see section 9.4.3).

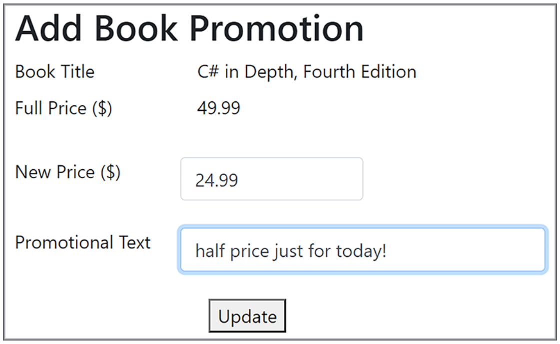
Although the SQL View works well for this purpose, it creates a link between the two bounded contexts. Be careful if you change the `Book` entity and then migrate the database.

> **NOTE** Passing data between bounded contexts is a big topic that I can't cover here. I recommend the old but still-relevant article "Strategies for Integrating Bounded Contexts," in which Philip Brown gives a good overview of many ways to communicate between bounded contexts (http://mng.bz/96Bg).

13.5 Using your DDD-styled entity classes in your application

The DDD approach is to keep the focus on the domain mode—that is, on the entities and their relationships. Conversely, it doesn't want the database (DDD persistence) parts to distract the developer who is working on the domain design. The idea is that the entity and its relationships (navigational properties in EF Core terms) are all the developer needs to consider when solving domain issues.

Having updated the design of your entity classes to one that follows the DDD approach, now you want to use these classes in your application. Querying the `Book` hasn't changed, but how you create and update your DDD-styled entity classes has changed. In section 13.4, you altered your entity classes to use constructors or static create factories to create and access methods to update. In this section, you are going to look at how you would use these new approaches in an application. The examples come from the Book App (which is an ASP.NET Core MVC application) in part 3. Figure 13.6 shows the page that the admin user uses to add a promotion to a book.



Add Book Promotion

Book Title: C# in Depth, Fourth Edition

Full Price ($): 49.99

New Price ($): 24.99

Promotional Text: half price just for today!

Figure 13.6 Web page used to add a promotion to a book. This example shows the title and full price of the book, and invites the admin user to give the new price and some text to go next to the price. When the Update button is clicked, the `AddPromotion` access method is called with the new data, and if there are no errors, `SaveChanges` is called to update the `Book`.

Next, you will implement the code to put in an ASP.NET Core controller to display the page shown in figure 13.6 and update the `Book` entity when the user has provided their input and clicked the Update button. You will use two approaches: the standard approach described by Evans and a library designed to work directly with DDD access methods (which I refer to as a *class-to-method-call* library). The following list allows you to compare the two approaches:

- Calling the `AddPromotion` access method via a repository pattern
- Calling the `AddPromotion` access method via a class-to-method-call library

You will build code that adds a new `Review` to the `Book` entity class. Updates to relationships require you to decide how to handle the update. You will implement the `AddReview` example in two ways so that you can compare the two approaches:

- Adding a `Review` to the `Book` entity class via a repository pattern
- Adding a `Review` to the `Book` entity class via a class-to-method-call library

13.5.1 Calling the AddPromotion access method via a repository pattern

Evans's book uses a repository pattern to handle the database accesses. Microsoft's definition of the repository pattern says, "Repositories are classes or components that encapsulate the logic required to access data sources. They centralize common data access functionality, providing better maintainability and decoupling the infrastructure or technology used to access databases from the domain model layer."

There are many ways to build a repository pattern. I have chosen to use a generic repository, which will work with any entity. The following listing shows the generic repository that you need for the `AddPromotion` example.

Listing 13.7 A generic repository that handles some basic database commands

```
public class GenericRepository<TEntity>
    where TEntity : class
{
```

The generic repository will work with any entity class.

```
    protected readonly DbContext Context;

    public GenericRepository(DbContext context)
    {
        Context = context;
    }

    public IQueryable<TEntity> GetEntities()
    {
        return Context.Set<TEntity>();
    }

    public async Task<TEntity> FindEntityAsync(int id)
    {
        var entity = await Context.FindAsync<TEntity>(id);

        if (entity == null)
            throw new Exception("Could not find entity");

        return entity;
    }

    public Task PersistDataAsync()
    {
        return Context.SaveChangesAsync();
    }
}
```

The repository needs the DbContext of the database.

Returns an IQueryable query of the entity type

This method finds and returns a entity with a integer primary key.

Finds an entity via its single, integer primary key

A rudimentary check that the entity was found

The found entity is returned.

Calls SaveChanges to update the database

Using this repository, you can find a specific `Book` entity and call that `Book`'s `AddPromotion` access method, using the data provided by the admin user. The following listing shows the code using the `GenericRepository<Book>` that would go in the ASP.NET Core `AdminController`. This controller has two methods, both named `AddPromotion`, but with different parameters and attributes. The first `AddPromotion` method is called to display the page shown in figure 13.6. The second `AddPromotion` method is called when the user clicks the Update button and handles the update of the `Book` entity with the promotion.

> **NOTE** If you aren't familiar with ASP.NET Core, please look at section 5.7, which gives you a step-by-step look at how ASP.NET Core controllers work.

The following listing shows the `AdminController` with its constructor and the two methods. Note that only the new code that uses the repository has comments.

Listing 13.8 Handling the `AddPromotion` update by using a repository pattern

```
public class AdminController : Controller
{
    private readonly GenericRepository<Book> _repository;

    public AdminController(
        GenericRepository<Book> repository)
    {
        _repository = repository;
    }
```

The Generic-Repository<Book> is injected into the Controller.

```
    public async Task<IActionResult> AddPromotion(int id)
    {
        var book = await _repository.FindEntityAsync(id);

        var dto = new AddPromotionDto
        {
            BookId = id,
            Title = book.Title,
            OrgPrice = book.OrgPrice
        };

        return View(dto);
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> AddPromotion(AddPromotionDto dto)
    {
        if (!ModelState.IsValid)
        {
            return View(dto);
        }

        var book = await _repository
            .FindEntityAsync(dto.BookId);
        var status = book.AddPromotion(
            dto.ActualPrice, dto.PromotionalText);

        if (!status.HasErrors)
        {
            await _repository.PersistDataAsync();
            return View("BookUpdated", "Updated book…");
        }

        //Error state
        status.CopyErrorsToModelState(ModelState, dto);
        return View(dto);
    }
}
```

Uses the repository to read in the Book entity

Copies over the parts of the Book you need to show the page

Calls the AddPromotion access method with the two properties from the dto

The access method returned no errors, so you persist the data to the database.

13.5.2 Calling the AddPromotion access method via a class-to-method-call library

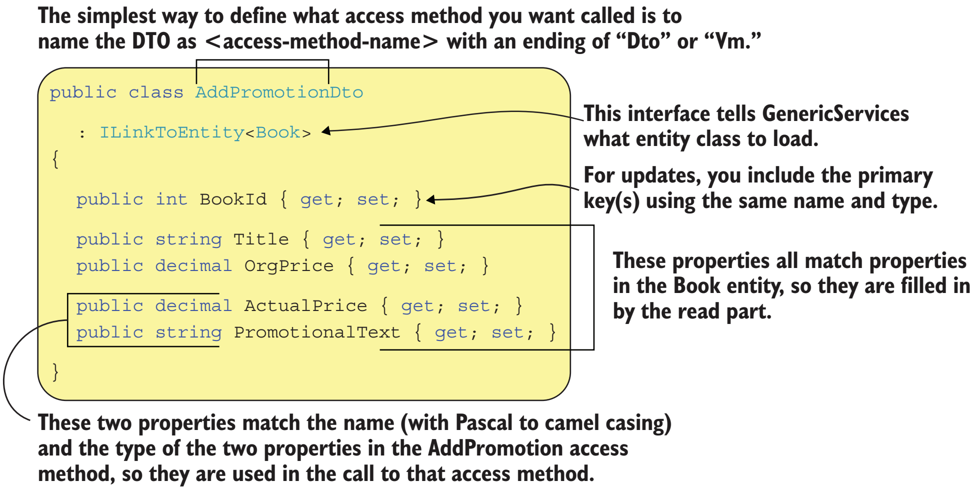
Although calling DDD access methods by using a repository system works, this approach has some repetitious code, such as in the first stage, where you copy properties into a DTO/ViewModel (referred to as the DTO from now on) to show to the user, and in the second stage, where returned data in the DTO is turned into an access method call. What would happen if you had a way to automate this process?

Well, in early 2018, after I had finished the first edition of this book, I found a way to automate both parts of the DDD CRUD and built an open source library called `EfCore.GenericServices` (referred to as `GenericServices` from now on). The `GenericServices` library automates most CRUD operations from normal entity classes with settable properties and DDD entity classes with their constructors and access methods.

One benefit of using this library is that it reduces the amount of code you have to write in comparison to the repository approach. The library saves you about five lines in ASP.NET Core, and you don't have to write the repository. Another benefit is that the code you use is the same for every update; only the DTO is different. The library allows you to copy/paste frontend code and then change only the DTO type to swap to another access method, constructor, or static factory.

> **NOTE** I designed `GenericServices` to deal with most, but not all, CRUD code. It's great at dealing with simple to moderate CRUD situations, but it can't cover every circumstance. For more complex CRUD code, I write the code by hand. You can learn more about this library at http://mng.bz/jBoP.

The rest of this section shows how you can implement the `AddPromotion` example by using `GenericServices`. First, you will look at the DTO in figure 13.7, which defines what entity the library needs to load, what properties to load for the read part, and what access method to call.



The simplest way to define what access method you want called is to name the DTO as <access-method-name> with an ending of "Dto" or "Vm."

```
public class AddPromotionDto

   : ILinkToEntity<Book>
{

   public int BookId { get; set; }

   public string Title { get; set; }
   public decimal OrgPrice { get; set; }

   public decimal ActualPrice { get; set; }
   public string PromotionalText { get; set; }

}
```

This interface tells GenericServices what entity class to load.

For updates, you include the primary key(s) using the same name and type.

These properties all match properties in the Book entity, so they are filled in by the read part.

These two properties match the name (with Pascal to camel casing) and the type of the two properties in the AddPromotion access method, so they are used in the call to that access method.

Figure 13.7 The DTO defines what entity class is read and updated by using the `ILinkToEntity<T>` interface. On a read, it will fill in all the properties in the DTO that have the same name/type as the linked entity class—in this case, the `Book` entity. The name of the DTO is used to find the access method to call, and the properties are found by matching names (with Pascal to camel casing) and their types.

The following listing shows the use of the `GenericServices` library instead of a repository (listing 13.8). Note that I've commented only the new code that uses `GenericServices`.

Listing 13.9 Handling the `AddPromotion` update by using `GenericServices`

```
//public class AdminController : Controller
{
    private readonly ICrudServicesAsync _service;

    public AdminController(
        ICrudServicesAsync service)
    {
        _service = service;
    }

    public async Task<IActionResult> AddPromotion(int id)
    {
        var dto = await _service
            .ReadSingleAsync<AddPromotionDto>(id);

        return View(dto);
       }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> AddPromotion(AddPromotionDto dto)
    {
        if (!ModelState.IsValid)
        {
            return View(dto);
        }

        await _service.UpdateAndSaveAsync(dto);

        if (!_service.HasErrors)
        {
            return View("BookUpdated", service.Message);
        }

        //Error state
        _service.CopyErrorsToModelState(ModelState, dto);
        return View(dto);
    }
}
```

The ICrudServicesAsync service comes from GenericServices and is injected via the Controller's constructor.

The ReadSingleAsync<T> reads into the DTO, using the given primary key.

The UpdateAndSaveAsync method calls the access method, and if no errors occur, it saves the access method to the database.

As you can see, the code is much smaller, with only one line in each ASP.NET Core action method. My own before-and-after analysis suggests that the `GenericServices` library reduces the time it takes to build a backend ASP.NET Core application by 10–20%.

13.5.3 Adding a Review to the Book entity class via a repository pattern

When you're updating navigational properties, you need to handle another step: preloading the navigational property. In listing 13.4, the access methods to add a `Review` to or remove a `Review` from the `Book` entity require the _reviews backing field to be filled before the addition or removal, so you need to update the repository that reads in the `Book` entity with its `Reviews` collection included. Because this task is specific to the `Book` entity, you create a `BookRepository` class that inherits the `GenericRepository`. The following listing shows this new `BookRepository`.

Listing 13.10 Add the `LoadBookWithReviewsAsync` method to the repository

```
public class BookRepository : GenericRepository<Book>
{
    public BookRepository(DbContext context)
        : base(context)
    { }

    public async Task<Book>
        LoadBookWithReviewsAsync(int bookId)
    {
        var book = await GetEntities()
            .Include(b => b.Reviews)
            .SingleOrDefaultAsync(
                b => b.BookId == bookId);
        if (book == null)
            throw new Exception("Could not find book");
        return book;
    }
}
```

The book repository inherits the generic repository to get the general commands.

The GenericRepository needs the application's DbContext.

Loads a Book with Reviews

Uses the GenericRepository's GetEntities to get a IQueryable<Book> query

Makes sure that the Review collection is loaded with the book

Selects the Book with the given BookId

A rudimentary check that the entity was found

Returns the book with the Reviews collection loaded

This repository replaces the `GenericRepository<Book>` in listing 13.9. This code snippet shows how you would call the `LoadBookWithReviewsAsync` method in ASP.NET Core's `POST` action method:

```
var book = await _repository
    .LoadBookWithReviewsAsync(dto.BookId);
book.AddReview(
    dto.NumStars, dto.Comment, dto.VoterName);
await _repository.PersistDataAsync();
```

13.5.4 Adding a Review to the Book entity class via a class-to-method-call library

For preloading navigational properties, the `GenericServices` library provides an `IncludeThen` attribute that you add to the DTO. This attribute allows you to define the name of navigational properties to `Include` or `ThenInclude`. The following listing shows the `AddReviewDto` class with its `IncludeThen` attribute.

Listing 13.11 The `AddReviewDto` class with an attribute to load the `Reviews`

```
[IncludeThen(nameof(Book.Reviews))]
public class AddReviewDto
```

The IncludeThen attribute includes the Book's Reviews navigational property.

The name of the DTO shows that it should call the AddReview access method.

```
    : ILinkToEntity<Book>
{
    public int BookId { get; set; }

    public string Title { get; set; }

    public string VoterName { get; set; }
    public int NumStars { get; set; }
    public string Comment { get; set; }
}
```

The entity that this DTO is linked to is the Book entity class.

The primary key of the Book is filled in by the read and used by the update method.

The Title is read in on a read and used to confirm to the user what book they are adding a Review to.

These three properties are used as parameters in the AddReview access method.

After you add the `IncludeThen` attribute, any read of an entity will include the navigational properties. You use `GenericServices`' `ReadSingleAsync<T>` and `UpdateAndSaveAsync(dto)` methods the same way that you would access methods that do not have navigational properties to update.

13.6 The downside of DDD entities: Too many access methods

Matthew Krieger read one of my articles about using DDD with EF Core and left this comment: "Here is the big thing I cannot get my head around: Won't you end up with lots of access methods?" He was right. In real applications, you can end up with lots of access methods. When you are building a large application, the time it takes to write an access method grows if you have hundreds to write.

Two of my clients used DDD, and both went for an approach that allowed some properties to be updated directly—that is, not using access methods. One client wanted to use JSON Patch to update the entities because it sped up the building of the frontend pages. Another client was using DDD but updating some properties by letting AutoMapper "punch through" the private setter and set the value. (See my warning in section 13.4.1.)

> **DEFINITION** JSON Patch is a way to send alterations to data by using a JSON object that conforms to the IETF RFC 6902 specification. See http://jsonpatch.com for more information.

Both clients were attempting to speed up development, and saying "That's not the right way to use DDD" wasn't the right suggestion. We agreed on this approach: if the property has no business rules (other than validation attributes), the setter on that property could be made public. I refer to entity classes that use this approach as *hybrid* DDD entities.

As an example, if you look at the `Book` entity class, the `Title` property and the `Publisher` property have no business logic but should not be empty, so the setter of these two properties could be made public without having any effect on the business rules. Making the properties' setter public would save you from writing two more access methods and allow JSON Patch or AutoMapper to update these properties. Some DDD practitioners might criticize this hybrid DDD suggestion as not following the DDD pattern, but I call it a pragmatic solution. ☺

NOTE My `GenericServices` library can detect and use hybrid DDD classes. If a DDD class has properties with public setters, it registers the entity class as a hybrid DDD class. Hybrid DDD classes enable `GenericServices` to use JSON Patch or AutoMapper to set these properties directly without needing an access method to be written. See my article at http://mng.bz/Wrj1.

13.7 Getting around performance issues in DDD-styled entities

So far, you have looked at the ways to apply the DDD approach to entity classes in EF Core. But when you start building real applications, you sometimes need to improve performance. Typically, the performance issues in an application involve queries, and DDD doesn't affect them at all. But if you have database write performance issues, you might feel the need to bypass DDD. Instead of ditching DDD, you have three ways to keep using DDD with minimal bending of the rules.

As an example, we look at the performance of adding or removing a `Review`. So far, you have loaded all the reviews before running add/remove access methods. If you have only a few `Reviews`, you have no problem, but if your site is like Amazon, where products can have thousands of reviews, loading all of them to add one new `Review` is going to be too slow.

In section 3.4.3, I describe a way to add a single `Review` to a `Book` by creating that `Review` and setting its `BookId` foreign key to the primary key of the `Book`. This approach means that you don't have to include all the `Reviews`, so the update will be quick. But all the solutions in this section break the DDD rule that the entity classes shouldn't know anything about the database code. So in this section, you look at three solutions.

Each solution requires one change: a way to set the `BookId` foreign key in the `Review` entity. This change immediately breaks the rule that DDD entities shouldn't know about the database, but I can't see any way around this part, although the last approach I describe is close.

We start with the following listing, which shows the updated `Review` constructor. Note that the `Review` has an internal access modifier, which means that it can be created only in the Domain.Books project. The use of an internal access modifier and the optional `BookId` parameter in the constructor will become clearer as we solve this issue.

Listing 13.12 The updated `Review` public constructor with optional foreign key

A new, optional property is added for setting the Review foreign key.

The Review constructor is internal, so only entity classes can create a Review.

```
internal Review(
    int numStars, string comment, string voterName,
    int bookId = 0)
{
    NumStars = numStars;
    Comment = comment;
    VoterName = voterName;
```

Standard properties

Sets the standard properties

```
        if (bookId != 0)
            BookId = bookId;
    }
```

If a foreign-key parameter was provided, the BookId foreign key is set.

> **ALTERNATIVE** The other option is to expose a navigational property linking the `Review` back to the `Book` entity. This option keeps the entity from knowing about foreign keys but breaks the DDD rule on minimizing relationships. Pick which rule you want to break.

After you have changed the `Review` entity, you can use any of three options:

- Allow database code into your entity classes.
- Make the `Review` constructor public and write nonentity code to add a `Review`.
- Use domain events to ask an event handler to add a `Review` to the database.

13.7.1 Allow database code into your entity classes

One solution is for the `AddReview` access method to have access to the application's DbContext. You can provide the application's DbContext by adding an extra parameter to the `AddReview`/`RemoveReview` access methods or using EF Core's service injection as shown in section 6.1.10. Listing 13.13 shows the two access methods to add/remove a `Review`. The DbContext is provided in the access methods via a parameter.

> **NOTE** I couldn't use this solution in the part 3 Book App because clean architecture bans adding any substantial libraries, especially anything having to do with databases, to the Domain projects. But I have used this solution in other applications.

Listing 13.13 Providing the application's DbContext to the access methods

```
public void AddReview(
    int numStars, string comment, string voterName,
    DbContext context)
{
    if (BookId == default)
        throw new Exception("Book must be in db");

    if (context == null)
        throw new ArgumentNullException(
            nameof(context),
            "You must provide a context");

    var reviewToAdd = new Review(
        numStars, comment, voterName,
        BookId);

    context.Add(reviewToAdd);
}

public void RemoveReview (
    int reviewId,
    DbContext context)
```

The access method takes the normal AddReview inputs ...

... but a new parameter is added, which is EF Core DbContext.

This method works only on a Book that is already in the database.

This method works only if an DbContext instance is provided.

Creates the Review and sets the Review BookId foreign key

Uses the DbContext Add method to mark the new Review to be added to the database

The access method takes the normal RemoveReview input of the ReviewId.

```
{
    if (BookId == default)
        throw new Exception("Book must be in db");

    if (context == null)
        throw new ArgumentNullException(
            nameof(context),
            "You must provide a context");

    var reviewToDelete = context.Set<Review>()
        .SingleOrDefault(x => x.ReviewId == reviewId);

    if (reviewToDelete == null)
        throw new Exception("Not found");
    if (reviewToDelete.BookId != BookId)
        throw new Exception("Not linked to book");

    context.Remove(reviewToDelete);
}
```

This method works only on a Book that is already in the database.

This method works only if an DbContext instance is provided.

Reads in the review to delete

A rudimentary check that the review entity was found

If not linked to this Book, throw an exception.

Deletes the review

This solution breaks the following DDD rules:

- The add/remove review access methods contain database features.
- The `Review` entity knows about a database feature: the `BookId` foreign key

NOTE `GenericServices` supports injection of a DbContext via a parameter. When `GenericServices` calls DDD constructors, static factories, or access methods, it looks for parameters of type `DbContext` or the type of the application's DBContext and fills them in with the DBContext that `GenericServices` was registered with.

13.7.2 Make the Review constructor public and write nonentity code to add a Review

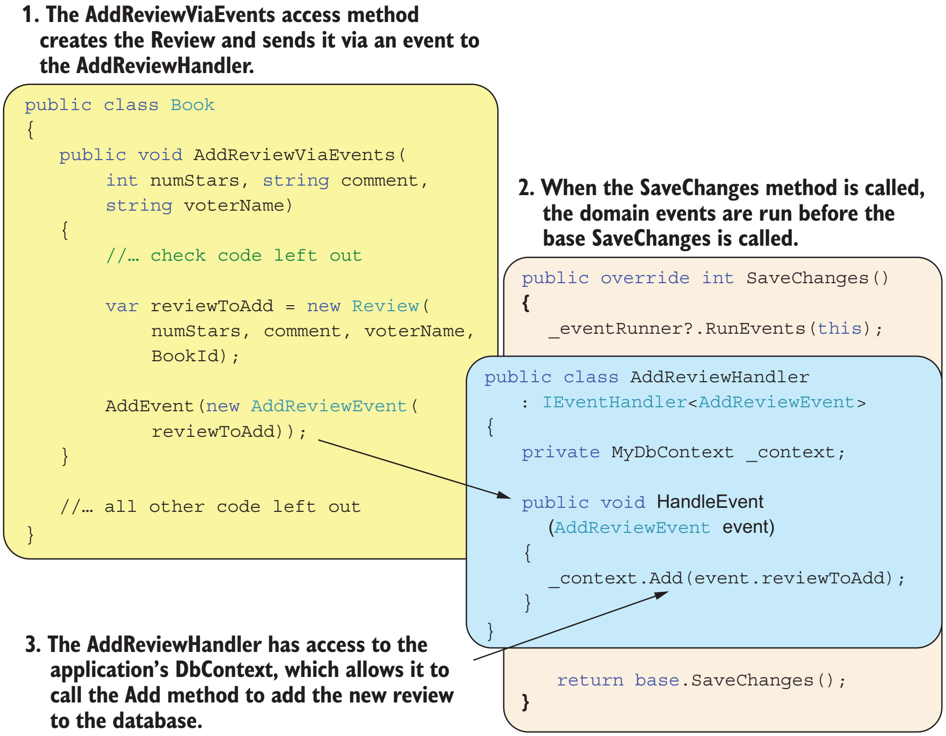
This solution removes the database features introduced in section 13.7.1 from the `Book`'s access methods and places them in another project (most likely BizLogic). The solution makes the `Book` entity cleaner, but it does require the `Review` constructor's access modifier to be changed to `public`. The downside is that anyone can create a `Review` entity instance.

The code to add/remove a `Review` is the same as shown in listing 13.4, but now it is in its own class. This solution breaks the following DDD rules:

- The `Book` entity isn't in charge of the `Review` entities linked to it.
- The `Review` has a public constructor, so any developer can create a `Review`.
- The `Review` entity knows about a database feature: the `BookId` foreign key.

13.7.3 Use domain events to ask an event handler to add a review to the database

The last solution is to use a domain event (see chapter 12) to send a request to event handlers that add or remove a `Review`. Figure 13.8 shows the `AddReviewViaEvents` access method in the `Book` entity on the left and the `AddReviewHandler` being run by `SaveChanges` (or `SaveChangesAsync`) on the right.



1. The AddReviewViaEvents access method creates the Review and sends it via an event to the AddReviewHandler.

```
public class Book
{
   public void AddReviewViaEvents(
       int numStars, string comment,
       string voterName)
   {
       //… check code left out

       var reviewToAdd = new Review(
           numStars, comment, voterName,
           BookId);

       AddEvent(new AddReviewEvent(
           reviewToAdd));
   }

   //… all other code left out
}
```

2. When the SaveChanges method is called, the domain events are run before the base SaveChanges is called.

```
public override int SaveChanges()
{
   _eventRunner?.RunEvents(this);

     return base.SaveChanges();
}
```

```
public class AddReviewHandler
   : IEventHandler<AddReviewEvent>
{
   private MyDbContext _context;

   public void HandleEvent
     (AddReviewEvent event)
   {
     _context.Add(event.reviewToAdd);
   }
}
```

3. The AddReviewHandler has access to the application's DbContext, which allows it to call the Add method to add the new review to the database.

Figure 13.8 A solution using events to add a single `Review` without loading all the `Reviews` in a `Book` entity. The Book `entity` has an access method called `AddReviewsViaEvents`, which creates the review and sends it in a domain event to an event handler. When your event-enhanced `SaveChanges/SaveChangesAsync` method is called, it finds and runs the `AddReviewHandler` while providing the domain event as a parameter. The event handler can access the application's DbContext, so it can call the `Add` method to add that new `Review` to the database. Then the base `SaveChanges/SaveChangesAsync` updates the database with the changes.

Figure 13.8 shows only the `AddReview` example, but `RemoveReview` would send the `ReviewId` to a `RemoveReviewHandler`, whose job is to find and delete that `Review`. This approach has the least divergence from the DDD approach because it leaves the `Book` entity in charge of managing the `Reviews` linked to the `Book`. Also, the `Review` entity can keep its internal access modifier so that no code outside the entity classes project can create a `Review`. But it still has the downside that all the solutions have: the `Review` entity knows about a database foreign key.

13.8 Three architectural approaches: Did they work?

The experience of building and enhancing part 3's Book App was a great test of applying the three architectural approaches during development. The Book App started with 9 projects, but by the end of chapter 16, it had 23 projects—a big change, with lots of refactoring to support new features. This section summarizes my experience of using these approaches on the initial part 3 Book App up to the end of chapter 16.

13.8.1 A modular monolith approach that enforces SoC by using projects

I was aware of the modular monolith approach but hadn't used it in an application before. My experience was that it worked well; in fact, it was much better than I thought it would be. I would use this approach again for any medium-size to large application. Following the modular monolith approach meant each project was small and focused, and giving the project a name that said what it did made navigating the code easy.

Having used the layered architecture (see section 5.2) for some time, I know that the service layer can get really big and hard to work on (sometimes referred to as the *big ball of mud* problem). I try to mitigate this problem by grouping related code into folders, but I'm never quite sure whether the code in folder A links to code in the other folders. When I'm using a layered architecture, if I'm in a hurry, I tend to write something new instead of refactoring the old code. I can't take the time to work out whether code is used elsewhere or uses something I don't know about.

By contrast, the modular monolith approach provides small, focused projects. I know that all the code in the project is doing one job, and the only links are to other projects that are relevant to this project. That approach makes the code easier to understand, and I'm much more inclined to refactor the old code, as I'm less likely to break something else.

One lazy thing I found myself doing was referring to the book display project that contained the original Book App code from part 1. That layer has some useful classes and `enum`s that could be used in other book-display projects. I was breaking the modular monolith rules by referring to a project that had a lot of code that wasn't relevant to the linked project. I should have pulled those common classes into a separate project, but I was racing to finish my book, and it was easy to reference the first book display project (as on a real job!). The modular monolith approach helps separate the code, but it relies on the developer to follow the rules.

> **NOTE** I had to go back to the Book App in chapter 16 to add new versions of some display features, so I took the opportunity to create a project called `BookApp.ServiceLayer.DisplayCommon.Books`, which holds all the common code. That project removes the linking between query features and makes the code much easier to understand and refactor.

Here are a few tips for using the modular monolith approach:

- Use a hierarchal naming rule for your projects. A name like `BookApp.Persistence.EfCoreSql.Books`, for example, makes it easier to find things.
- Don't end a project name with the name of a class. Instead, use something like `...Books`, not `...Book`. I named some projects `...Book`, which required me to prefix each `Book` class with its complete namespace—in this case, `BookApp.Domain.Books.Book`.
- You're going to get project names wrong. I called one project `BookApp.Infrastructure.Books.EventHandlers`, but as the Book App grew and the project expanded, I had to change it to `BookApp.Infrastructure.Books.CachedValues`.
- If you change the name of a project in Visual Studio by selecting the project and typing the new name, you don't change the folder name. I found that situation confusing in GitHub, so I made sure to rename the folder as well, which meant editing the solution file (there is a nice tool that can do this for you; see https://github.com/ModernRonin/ProjectRenamer).
- You need Visual Studio 16.8.0 or later if you are going to have lots of projects in your application because Visual Studio 16.8 is much quicker than older versions at dealing with lots of projects in a solution. (VS Code has always been fast with lots of projects.)

13.8.2 DDD principles, both architecturally and on the entity classes

I am familiar with using DDD, and as I expected, it worked well. Here is a list of DDD features that made the development of the Book App easier:

- Each entity class contained all the code needed to create or update that entity and any aggregate entities. If I needed to change anything, I knew where to look, and I knew that there wasn't another version of this code elsewhere.
- The DDD access methods were especially useful when I used domain events in chapter 15.
- The DDD access methods were even more useful when I used integration events in chapter 16 because I had to capture every possible change to the `Book` entity and its aggregates, which was easy to do by adding integration events to every access method and static create factory method in the `Book` entity. If I couldn't capture all changes in that way, I would have to detect changes by using the entities' `State`, and I know from experience that detecting changes is hard to implement.
- The DDD bounded context that allowed two different EF Core DbContexts, `BookDbContext` and `OrderDbContext`, also worked well. Migrating the two parts of the same database (see section 9.4.5) worked fine.

13.8.3 Clean architecture as described by Robert C. Martin

It wasn't my first time using clean architecture, as I had worked on a client's application that used a clean architecture approach, but it was the first time I'd started from scratch with this approach. I was much more aware of where I should place the different parts of the application. Overall, I found the clean architecture layers to be useful, but I had to change one thing, which I will describe at the end of this section.

By the end of chapter 16, the part 3 Book App consisted of five layers, starting at the center and working out:

- *Domain*—Holding entity classes
- *Persistence*—Holding the DbContexts and other database classes
- *Infrastructure*—Holding a mixture of projects, such as seeding the database and event handlers
- *ServiceLayers*—Holding code to adapt the lower layers to the frontend
- *UI*—Holding the ASP.NET Core application

Figure 13.9 shows these five layers, with the number of projects in each layer after all the chapters were written.

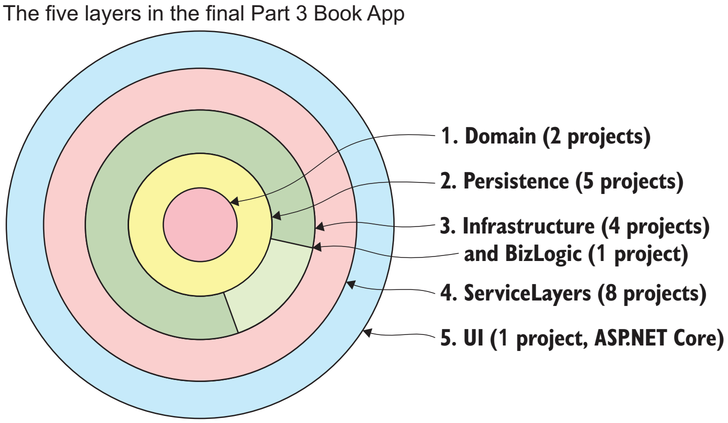


Figure 13.9 The five layers of the Book App in part 3, with the number of projects in each layer after chapter 16 was finished. See the `Part3` branch of the associated GiHub repo for each project.

The main problem was fitting the EF Core DbContext into the clean architecture. Clean architecture says that the database should be on the outer ring, with interfaces for the access. The problem is that there is no simple interface you can use for the application's DbContext. Even if I were using a repository pattern (which I wasn't), I would still have a problem because the application's DbContext has to be defined deep in the onion.

One rule of the clean architecture approach that I didn't like, but stuck to, is that the Domain layer shouldn't have any significant external package (such as a NuGet library) added to it. This clean-architecture rule required me to do more work in a few places. In chapter 15, for example, I had code that marked every `Book` entity when it was added or updated. It would have been easier to pass EF Core's `EntityEntry` class to the `LogAddUpdate` method in the Domain level. Also, in chapter 16, I wanted to use an owned type with Cosmos DB, and I had to use Fluent API configuration commands to set that up. I would rather have added the `[Owned]` attribute to the class, which would save me from adding a `OnModelCreating` method to the `CosmosDbContext` class to add extra Fluent API configuration commands. Next time, I might add the EF Core base NuGet package to handle these features.

Summary

- The architecture you use to build an application should help you focus on the feature you are adding while keeping the code nicely segregated so that it's easier to refactor.
- DDD provides lots of good recommendations on how to build an application, but this chapter focuses on EF Core entity classes and application DbContexts.
- DDD-styled entities control how they are created and updated; it's the job of an entity to ensure that the data inside it is valid.
- DDD has lots of rules to make sure that developers can put all their effort into the domain (business) needs that they have been asked to implement.
- DDD groups entities into aggregates and says that one entity in the group, known as the root aggregate, manages the data and relationships across the aggregate.
- Bounded context is a major DDD concept. This chapter looks only at how bounded context might be applied to the application's DbContext.
- To update a DDD entity, you call a method within the entity class. In this book, these methods are referred to as access methods.
- To create a new instance of an DDD entity, you use a constructor with specific parameters or a static create factory method that returns validation feedback.
- To update a DDD entity, first load the entity so that you can call the access method. You can do this via normal EF Core code, a repository, or the `EFCore.GenericServices` library.
- The `EFCore.GenericServices` library saves you development time. It removes the need to write a repository, and it can find and call access methods by using the name and properties in the DTO.
- Updating collection relationships can be slow if there are lots of existing entries in the collection. You have three ways to improve performance in these cases.

- A review of applying the three architectural approaches through chapter 16 shows that all three made enhancing and refactoring the Book App easier. All approaches worked, but the standouts were modular monolith and DDD.

For readers who are familiar with EF6.x:

- In EF6.x, you can't fully create DDD entities in EF6.x because you can't make navigational collection properties read-only. EF Core has solved that problem with its backing fields feature.

14

EF Core performance tuning

This chapter covers

- Deciding which performance issues to fix
- Employing techniques that find performance issues
- Using patterns that promote good performance
- Finding patterns that cause performance issues

This chapter is the first of three addressing performance-tuning your database accesses. Covering what to improve, as well as where and how to improve your EF Core database code, this chapter is divided into three parts:

- *Part 1*—Understanding performance, the difference between speed and scalability, deciding what to performance-tune, and determining the costs of performance tuning
- *Part 2*—Techniques you can use to find performance issues and the use of EF Core's logging to help you spot problems
- *Part 3*—A whole range of database access patterns, both good and bad, to help you diagnose and fix many EF Core performance issues

In chapter 15, you'll apply the approaches shown in this chapter to the Book App's book list query. You'll start by tuning EF Core code and then progress to more complex techniques, such as adding SQL commands to squeeze the best performance out of the database accesses.

14.1 Part 1: Deciding which performance issues to fix

Before describing how to find and fix performance issues, I want to provide an overview of the subject of performance. Although you can ignore performance at the start of a project, some concepts might help you later, when someone says, "The application is too slow; fix it."

When people talk about an application's *performance*, they're normally talking about how fast an application deals with requests—how long it takes an API to return a specific request, for example, or how long a human user has to wait when searching for a specific book. I call this part of the application's performance *speed* and use terms such as *fast* and *slow* to describe it.

The other aspect is what happens to the speed of your application when it has lots of simultaneous requests. A fast website with a few users might become slow when it has many simultaneous users, a situation that is referred to as the *scalability* of the application—the ability of the application to feel fast even when it has a high load of users. Scalability is often measured via *throughput*—the number of requests an application can handle per second.

14.1.1 "Don't performance-tune too early" doesn't mean you stop thinking

Pretty much everyone says you shouldn't performance-tune early; the number-one goal is to get your application working properly first. A saying attributed to Kent Beck is "Make it Work. Make it Right. Make it Fast," which gets across the progressive steps in building an application, with performance tuning coming last. I totally agree, but with three caveats:

- Make sure that any software patterns you use don't contain inherent performance problems. Otherwise, you'll be building in inefficiencies from day one. (See section 14.4.)
- Don't write code that makes it hard to find and fix performance problems. If you mix your database access code with other code, such as frontend code, for example, performance changes can get messy and difficult to test. (See section 14.4.6.)
- Don't pick the wrong architecture. Nowadays, the scalability of web applications is easier to improve by running multiple instances of the web application. But if you have an application that needs high scalability, a Command and Query Responsibility Segregation (CQRS) architecture might help. I cover this topic in chapter 16.

It's often hard to predict what performance problems you're going to hit, so waiting until your application is starting to take shape is sensible. But a bit of up-front thought can save you a lot of pain later if you find that your application is too slow.

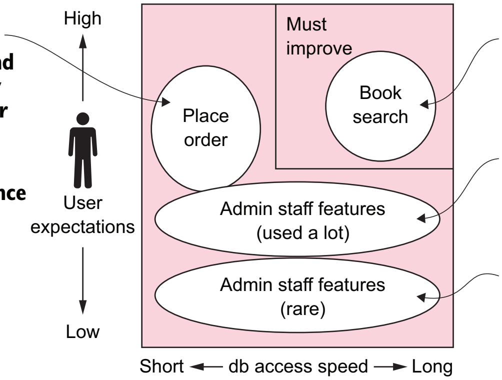
14.1.2 How do you decide what's slow and needs performance tuning?

The problem with terms such as *fast*, *slow*, and *high load* is that they can be subjective. You might think your application is fast, but your users may think it's slow. Sticking with subjective views of an application's performance isn't going to help, so the key questions are these: Does the speed matter in this case, and how fast should it be?

You should remember that in human-facing applications, the raw speed matters, but so do the *user's expectations* of how fast a certain feature should be. Google search has shown how blindingly fast a search can be, for example; therefore, we expect all searches to be fast. Conversely, paying for an online purchase—with the need to fill in your address, credit card number, and so on—isn't something that we expect to be fast (although if it's too slow or too difficult, we'll give up!).

When you think about what needs to be performance-tuned, you need to be selective; otherwise, you're in for a lot of work for little gain. I once developed a small e-commerce site that had a little more than 100 different queries and updates to 20 database tables. More than 60% of the database accesses were on the admin side, and some were rarely used. Maybe 10% of the database accesses affected paying users. That analysis helped me decide where to put my effort.

Figure 14.1 shows what happens when you apply the same analysis of the user's expectations against the speed of the database access for the Book App. This analysis covers the book listing/search; the placing of an order; and the few admin commands, ranging from updating the publication date of a book (fast) to wiping and inputting all the books again (quite slow).



Strategic decision:
The order-processing code is already fast, and users don't necessarily expect placing an order to be quick.

But you might want to improve the performance to make the user experience better (so that they buy more).

Must improve:
Users expect the search to be quick, because they're used to Google's search speed.

Might improve:
It may be worth speeding up the everyday admin commands.

Don't improve:
Rare admin commands aren't worth the effort to improve.

Figure 14.1 Various features from the Book App graded with the user's expectations of speed on the vertical access and the actual complexity/speed of the database access part of the feature. The type of user and user expectations have a big impact on what needs performance-tuning.

After you've done some analysis of your application, you should get a list of features that are worthy of performance tuning. But before you start, you need clear metrics:

- *Define the feature.* What's the exact query/command that needs improving, and under what circumstances is it slow (number of concurrent users, for example)?
- *Get timings.* How long does the feature take now, and how fast does it need to be?
- *Estimate the cost of the fix.* How much is the improvement worth? When should you stop?
- *Prove that it still works.* Do you have a way to confirm that the feature is working properly before you start the performance tuning and that it still works after the performance change?

TIP You can find an old but still-useful article on general performance tuning at http://mng.bz/G62D.

14.1.3 The cost of finding and fixing performance issues

Before diving into finding and fixing performance issues, I want to point out that there's a cost to performance-tuning your application. It takes development time and effort to find, improve, and retest an application's performance. As figure 14.1 illustrates, you need to be picky about what you plan to improve.

Many years ago, I wrote an article in which I measured the gain in performance in an EF6.x database access against the time it took me to achieve that improvement. Figure 14.2 shows the results of that work. I started with an existing EF6.x query (1 on the

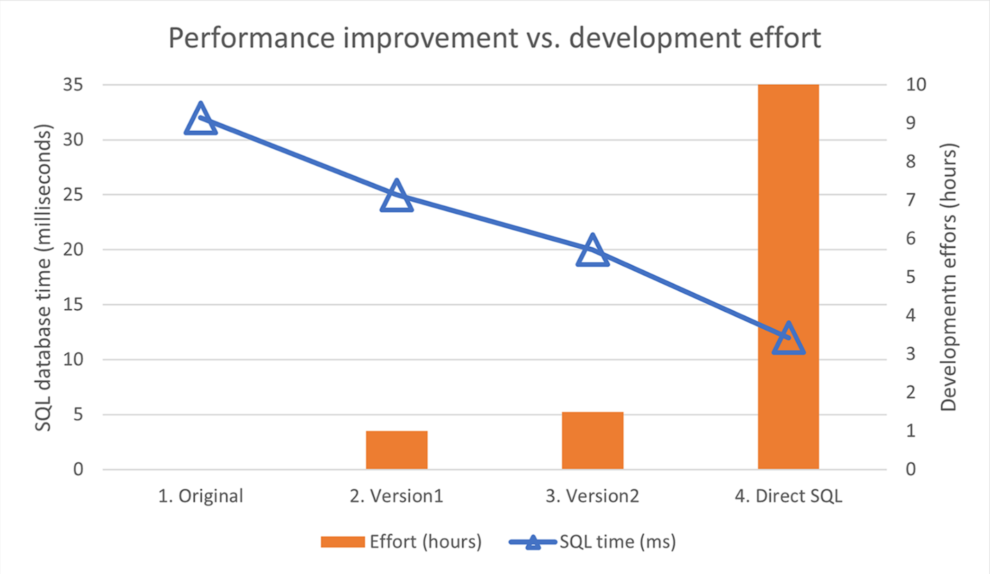


Figure 14.2 The trade-off between database performance and development effort for three stages of improvement of an EF database access. Development time is shown as a bar chart (hours: left scale), and the speed of the database access is shown as a line (milliseconds: right scale). An almost-exponential increase occurs in development effort against an almost-linear reduction in database access time.

horizontal scale) and then applied two steps (2 and 3) of improvement, still using EF6.x. Finally, I estimated the time it would take to write a raw SQL version (4 on the horizontal scale).

The point of figure 14.2 is to show that extreme performance improvements aren't easy. I had an exponential increase in development effort against an almost-linear reduction in database access time. Therefore, it's worth thinking about the problem holistically. Although it might be that the database access is slow, the solution might come from changing other parts of the application. For web/mobile applications, you have a few other possibilities:

- *HTTP caching*—Caching allows you to remember a request in memory and return a copy if the same URL is presented, thus saving any need to access the database. Caching takes work to get right, but it can have a big effect on perceived performance.
- *Scaling up/out*—Cloud hosting allows you to pay for more-powerful host computers (known as *scaling up* in Azure) and/or running more instances of the web application (known as *scaling out* in Azure). This approach might solve a lot of small performance problems quickly, especially if scalability is the problem.

I'm not suggesting sloppy programming. I certainly try to show good practices in this book. But by choosing EF Core over writing direct SQL commands, you've already opted for quicker development time with (possibly) slower database access times. In the end, it's always about effort against reward, so you should performance-tune only the parts of your application that need the extra speed or scalability.

14.2 Part 2: Techniques for diagnosing a performance issue

In part 1, you decided which parts of your application need improving and how much improvement you want. The next steps are finding the code involved in the slow feature and diagnosing the problem.

This book is about EF Core, so you'll concentrate on the database code, but those database accesses rarely exist on their own. You need to drill down through your application to find the database code that's hitting the application's performance. Figure 14.3 shows a three-step approach that I use to pinpoint performance bottlenecks. You'll explore these stages in detail in the next three subsections.

> **WARNING** Measuring the time it takes for ASP.NET Core to execute a command in debug mode can give misleading figures, because some slow logging methods may be enabled. These methods can add significant extra time to each HTTP request. I recommend testing your software in Release mode to get more-representative figures.

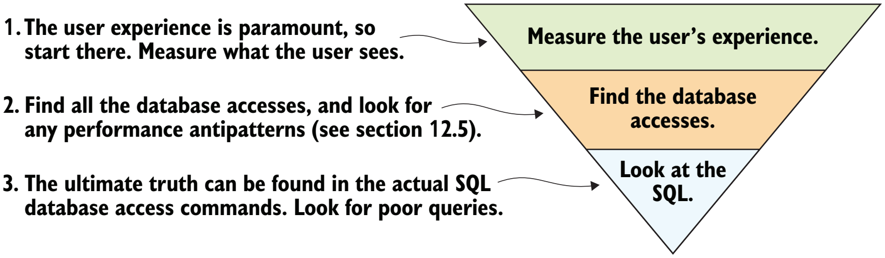


Figure 14.3 Finding database performance issues requires you to start with what the user sees and then drill down to the database code. After finding the database code, you check whether it uses the optimal strategies outlined in this chapter. If this step doesn't improve the situation, you need to look at the actual SQL commands sent to the database and consider ways to improve them.

14.2.1 Stage 1: Get a good overview, measuring the user's experience

Before you go digging to find a performance problem, you need to think about the user's experience, because user experience is what matters. You might improve the speed of a database access by 500%, but if speed is a small part of the whole picture, that improvement won't help much.

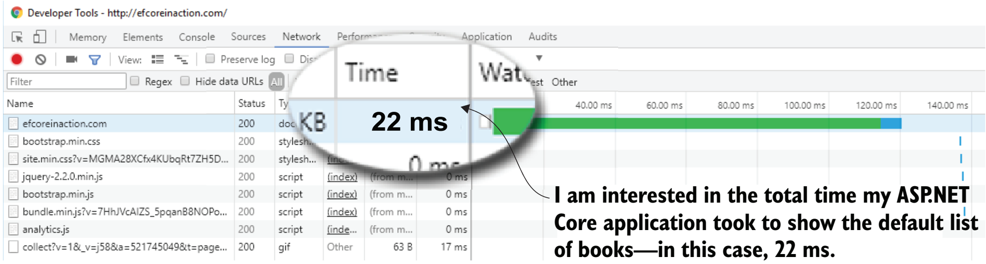
First, you need to find a tool that measures how long a specific request/feature takes. What you use will depend on the type of application you're using. Here's a list of free tools that are available for looking at the overall time a request takes:

- For Windows applications, you can use the Performance Profiler in Visual Studio.
- For websites, you can use your browser in developer mode to obtain timings (I use Google Chrome).
- For the ASP.NET Core Web API, you can use Azure Application Insights locally in debug mode.
- And don't forget logging output. ASP.NET Core and EF Core's logging output include timings.

NOTE All the tools I listed are free, but plenty of commercial (paid) tools are available for testing and profiling all manner of systems.

Figure 14.4 shows the timeline for the Book App before any performance tuning, as measured by the Google browser, Chrome, in developer mode (F12), but most web browsers contain the same features. The figure shows only one timing, but you should take multiple timings for a query, as timings will differ. Also, to get an overview of where the performance issues exist, you should try different sort/filter combinations in the book list feature. See chapter 15 for an example of timings for multiple sort/filter combinations.

NOTE The Book App in branch `Part3` captures ASP.NET Core's `RequestFinished` log, which contains the total time for the HTTP request. If you



Name	Status	Ty			Time
efcoreinaction.com	200	doc		KB	22 ms
bootstrap.min.css	200	stylesh...			
site.min.css?v=MGMA28XCfx4KUbqRt7ZH5D...	200	stylesh...	(inc		0 ms
jquery-2.2.0.min.js	200	script	(index)	(from m...	0 ms
bootstrap.min.js	200	script	(index)	(from m...	0 ms
bundle.min.js?v=7HhJVcAIZS_5pqanB8NOPo...	200	script	(index)	(from m...	0 ms
analytics.js	200	script	(inde...	(from m...	0 ms
collect?v=1&_v=j58&a=521745049&t=page...	200	gif	Other	63 B	17 ms

I am interested in the total time my ASP.NET Core application took to show the default list of books—in this case, 22 ms.

Figure 14.4 Using the Google Chrome browser in development mode to find out how long the Book App takes to display 700 books when using EF Core 5, before you start any performance tuning. This feature is already quick, but in chapter 15, when we take it up to 100,000 books, it starts to have problems.

repeat the same query, this feature will provide max, min, and average timings. You can access this timing feature via the Admin > Timings last URL menu command.

14.2.2 Stage 2: Find all the database code involved in the feature you're tuning

Having identified the part of the application you want to performance-tune, you need to locate all the database access code involved in that feature. After you've found the database code, run your eye over the code, looking for performance antipatterns (see sections 14.5 and 14.6), which is a quick way to find and fix issues. It's not foolproof, but after a while, you'll get a feel for what might be causing a problem.

When you look at the listing of books in your Book App, for example, the most obvious performance bottleneck is calculating the average review votes. Average review votes are used not only for displaying to the user, but also for sorting and filtering the books to be displayed. Running various timing tests showed that sorting or filtering on average votes was slow, but it wasn't until I looked at the EF Core logging output (section 14.2.3) that I saw the problems.

The Book App doesn't have a lot of writes, only adding a review or adding or removing a promotion, and they are fast, but in many applications, writes can be a bit of a bottleneck. Write-performance issues can be more complex to diagnose, as there are two parts to consider: the time EF Core takes to detect and link changes to the data and the time it takes to write to the database. For writes, the overall timing is important, as it contains both parts (see section 14.6).

14.2.3 Stage 3: Inspect the SQL code to find poor performance

The ultimate source of database query performance is the SQL code, and the EF Core logs will list the SQL sent to the database, along with the time that query took. I'll cover how you can use this information to look for performance issues, but first, let me describe how to access the logging information that EF Core produces. The steps in obtaining the EF Core logging output are

1. Understanding the logging output produced by EF Core
2. Capturing the logging output
3. Extracting the SQL commands sent to the database

UNDERSTANDING THE LOGGING OUTPUT PRODUCED BY EF CORE

.NET Core defines a standard logging interface that any piece of code can use. EF Core produces a substantial amount of logging output, which is normally collected by the application it's running in. Logging information is categorized by a `LogLevel`, which ranges from the most detailed information at the `Trace` (0) level, right up to `Critical` (5). In production, you'd limit the output to `Warning` (3) and above, but when running in debug mode, you want `Information` level, as EF Core (and ASP.NET Core) have useful information and timings at this level.

CAPTURING THE LOGGING OUTPUT

One way to access the logs is to use what is known as a *logging provider*. Logging is so useful that most applications include code to set up the logging providers. In an ASP.NET Core application, for example, a logging provider(s) is configured during startup (see http://mng.bz/KH6W), so you can obtain the logs as your application is running either in debug mode or from your live application.

Another way to capture logging information is to use EF Core 5's new `LogTo` feature inside your unit tests. This feature provides a simple way to capture the log output by EF Core. Listing 14.1 shows you one way to use this feature, but I recommend that you also read chapter 17, which is all about unit-testing your EF Core code.

> **NOTE** Because you're using the xUnit unit-tests library (see https://xunit.net), you can't output by using the `Console.Writeline` method, as xUnit runs tests in parallel. Therefore, you'll log to a list. Chapter 17 covers this topic in detail in section 17.11.1, including how to output to a console from xUnit.

Listing 14.1 Capturing EF Core's logging output in a unit test

```
var logs = new List<string>();
var builder =
    new DbContextOptionsBuilder<BookDbContext>()
    .UseSqlServer(connectionString)
    .EnableSensitiveDataLogging()
    .LogTo(log => logs.Add(log),
        LogLevel.Information);
using var context = new BookDbContext(builder.Options);
 //... your query goes here
```

Holds all the logs that EF Core outputs

The DbContextOptionsBuilder<T> is the way to build the options needed to create a context.

Says you are using a SQL Server database and takes in a connection string

The log string is captured and added to the log.

By default, exceptions don't contain sensitive data. This code includes sensitive data.

Sets the log level. Information level contains the executed SQL.

Creates the application's DbContext—in this case, the context holding the books data

> **WARNING** The `EnableSensitiveDataLogging` method in listing 14.1 will include any parameters in the logging. This method is helpful for debugging but should *not* be used in your live application, as the parameters may contain private data that should not be logged for security and/or privacy reasons.

We've covered how to capture EF Core's logging; next, you'll see how to use this information to find performance issues.

EXTRACTING THE SQL COMMANDS SENT TO THE DATABASE

EF Core logs what it is doing, and these logs can be useful. If you set the log level to `Information` in your application, you'll get a complete list of the SQL commands generated by EF Core and sent to the database. The following listing shows an example of an `Information` message containing the SQL code from the part 1 or 2 Book App context.

Listing 14.2 An `Information` log showing the SQL command sent to the database

```
Executed DbCommand (4ms)
    [Parameters=[],
     CommandType='Text',
     CommandTimeout='30']
SELECT [p].[BookId], [p].[Description],
    [p].[ImageUrl], [p].[Price],
    [p].[PublishedOn], [p].[Publisher],
    [p].[Title],
    [p.Promotion].[PriceOfferId],
    [p.Promotion].[BookId],
    [p.Promotion].[NewPrice],
    [p.Promotion].[PromotionalText]
FROM [Books] AS [p]
LEFT JOIN [PriceOffers] AS [p.Promotion]
ON [p].[BookId] = [p.Promotion].[BookId]
ORDER BY [p].[BookId] DESC
```

Tells you how long the database took to return this command

If any external parameters are used in the command, their names will be listed here.

The timeout for the command. If the command takes more than that time, it's deemed to have failed.

SQL command that was sent to the database

For those of you who are happy working with SQL, you can copy the SQL code from the logging output and run it in some form of query analyzer. Microsoft SQL Server Management Studio (SSMS) allows you to run a query and look at its execution plan, which tells you what each part of the query is made up of and the relative cost of each part. Other databases have a query analyzer, such as MySQL Query Analyzer and the PostgreSQL plprofiler.

14.3 Part 3: Techniques for fixing performance issues

The rest of this chapter provides a list of good and bad EF Core patterns for database access. These patterns are here both to teach you what can help or hurt performance and to act as a reference on database performance issues. This section consists of four parts:

- *Good EF Core patterns*—“Apply always” patterns that you might like to adopt. They aren’t foolproof but give your application a good start.
- *Poor database query patterns*—EF Core code *antipatterns*, or patterns you shouldn’t adopt, because they tend to produce poor-performing SQL queries.
- *Poor software patterns*—EF Core code antipatterns that make your database write code run more slowly.
- *Scalability patterns*—Techniques that help your database handle lots of database accesses.

Chapter 15 walks you through an example of the performance-tuning approaches shown in this chapter. Chapter 15 starts with tuning the EF Core commands in your Book App, but then goes into deeper techniques, such as replacing EF Core code with direct SQL and changing the database structure to provide better performance. Chapter 16 takes the discussion to the next level with a CQRS approach using the Cosmos DB database, which has excellent performance and scalability.

14.4 Using good patterns makes your application perform well

Although I’m not a fan of early performance tuning, I do look at the performance aspects of any patterns I adopt. It’s silly to create a pattern that’s going to bake in poor performance right from the start. Many of the patterns and practices described in this book do have some effect on performance or make performance tuning easier. Here’s a list of the patterns that help with performance issues that I always apply right from the start of a project:

- Using `Select` loading to load only the columns you need
- Using paging and/or filtering of searches to reduce the rows you load
- A warning that using lazy loading will affect database performance
- Always adding the `AsNoTracking` method to read-only queries
- Using the async version of EF Core commands to improve scalability
- Ensuring that your database access code is isolated/decoupled, so it’s ready for performance tuning

14.4.1 Using Select loading to load only the columns you need

In section 2.4, you learned about the four ways of loading related data, one of which was to use the LINQ `Select` command. For database queries that require information from multiple tables, the `Select` method often provides the most efficient database access code for queries. (See section 14.5.1 for more on minimizing database accesses.) Figure 14.5 illustrates this process.

Creating a `Select` query with a DTO does take more effort than using eager loading with the `Include` method (see section 2.4.1), but benefits exist beyond higher database access performance, such as reducing coupling between layers.

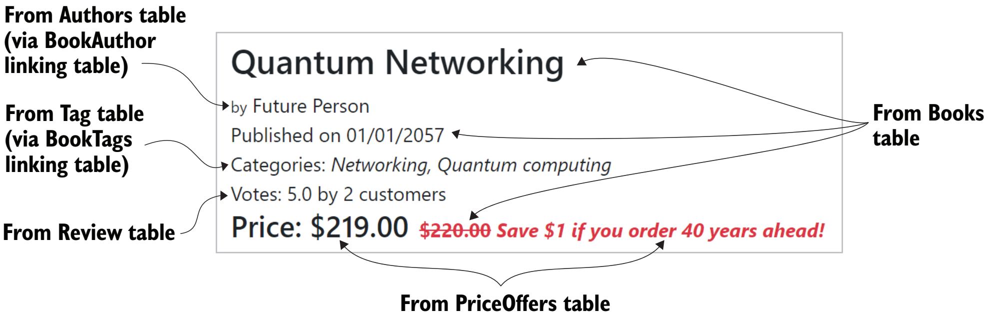


Figure 14.5 `Select` queries provide the best-performing database access, in which the final result consists of a mixture of columns from multiple tables.

TIP Section 6.1.9 describes how you can use AutoMapper to automate the building of a `Select` query and thus speed your development.

14.4.2 Using paging and/or filtering of searches to reduce the rows you load

Because EF Core's queries use LINQ commands, you can sometimes forget that one query can pull in thousands or millions of rows. A query that works fine on your development system, which might have only a few rows in a table, may perform terribly on your production system, which has a much larger set of data. You need to apply commands that will limit the amount of data returned to the user. Typical approaches are as follows:

- *Paging*—You return a limited set of data to the user (say, 100 rows) and provide the user commands to step through the "pages" of data (see section 2.7.3).
- *Filtering*—If you have a lot of data, a user will normally appreciate a search feature, which will return a subset of the data (see section 2.7.2).

Remember not to write open-ended queries, such as `context.Books.ToList()`, because you might be shocked when it runs on your production system, especially if you're writing code for Amazon's book site.

14.4.3 Warning: Lazy loading will affect database performance

Lazy loading (see section 2.4.4) is a technique that allows relationships to be loaded when read. This feature is in EF6.x and was added to EF Core in version 2.1. The problem is that lazy loading has a detrimental effect on the performance of your database accesses, and after you've used lazy loading in your application, replacing it can require quite a bit of work.

This instance is one in which where you bake in poor performance, and you might regret doing that. When I understood the effects of lazy loading in EF6.x, I didn't use it anymore. Sure, it can make development easier in some cases, but each lazy load is going to add another database access. Considering that the first performance antipattern

I list is "Not minimizing the number of calls to the database" (section 14.5.1), if you have too many lazy loads, your query is going to be slow.

14.4.4 Always adding the AsNoTracking method to read-only queries

If you're reading in entity classes directly and aren't going to update them, including the `AsNoTracking` method (see section 6.1.2) in your query is worthwhile. It tells EF Core not to create a tracking snapshot of the entities loaded, which saves a bit of time and memory use. It also helps when saving data, as it reduces the work that `DetectChanges` method has to do (see section 14.6.2).

The query in listing 14.3 is an example of one for which the `AsNoTracking` method, in bold, will improve performance. The simple performance test of loading 100 `Books` with `Reviews` and `Authors` in chapter 6 said that using `AsNoTracking` was 50% quicker—an extreme case, because the query had 5,000 `Reviews` in it; fewer relationships will provide less performance savings. See table 6.1 for detailed timings.

Listing 14.3 Using the `AsNoTracking` method to improve the performance of a query

```
var result = context.Books
    .Include(r => r.Reviews)
    .AsNoTracking()
    .ToList();
```

Returns a Book entity class and a collection of Review entity classes

Adding the AsNoTracking method tells EF Core not to create a tracking snapshot, which saves time and memory use.

If you use a `Select` query in which the result maps to a DTO, and that DTO doesn't contain any entity classes, you don't need to add the `AsNoTracking` method. But if your DTO contains an entity class, adding the `AsNoTracking` method will help.

14.4.5 Using the async version of EF Core commands to improve scalability

Microsoft's recommended practice for ASP.NET applications is to use async commands wherever possible. (Section 5.10 explains async/await.) This practice improves the scalability of your website by releasing a thread while the command is waiting for the database to respond; this freed-up thread can run another user's request.

Nowadays, using async/await has a small performance cost, so for applications that handle multiple simultaneous requests, such as a website, you should async/await. Section 14.7.2 covers this topic in more detail.

14.4.6 Ensuring that your database access code is isolated/decoupled

As I said earlier, I recommend that you get your EF Core code working first, without any performance tuning—but you should be ready to make that code faster if you need to later. To achieve isolation/decoupling, make sure that your code

- *Is in a clearly defined place (isolated).* Isolating each database access into its own method allows you to find the database code that's affecting performance.

- *Contains only the database access code (decoupled).* My advice is to not mix your database access code with other parts of the application, such as the UI or API. That way, you can change your database access code without worrying about other, nondatabase issues.

Throughout this book, you've seen lots of examples of this approach. Chapter 2 introduced the Query Object pattern (see section 2.6), and chapter 4 showed the use of a separate project to hold the database access code for the business logic (see section 4.4.4). These patterns make performance-tuning your database access code easier, as you have a clearly defined section of code to work on.

14.5 Performance antipatterns: Database queries

The previous patterns are worth using all the time, but you'll still bump into issues that require you to tune up your LINQ. EF doesn't always produce the best-performing SQL commands, sometimes because EF didn't come up with a good SQL translation, and sometimes because the LINQ code you wrote isn't as efficient as you thought it was.

This section presents some of the performance antipatterns that affect the time it takes to get data to and from the database. I use the negative antipattern terms, as that's what you're looking for—places where the code can be improved. Here's a list of potential problems, followed by how to fix them, with the ones you're most likely to hit listed first:

- Not minimizing the number of calls to the database
- Missing indexes from a property that you want to search on
- Not using the fastest way to load a single entity
- Allowing too much of a data query to be moved into the software side
- Not moving calculations into the database
- Not replacing suboptimal SQL in a LINQ query
- Not precompiling frequently used queries

14.5.1 Antipattern: Not minimizing the number of calls to the database

If you're reading an entity from the database with its related data, you have four ways of loading that data: select loading, eager loading, explicit loading, and lazy loading. Although all three techniques achieve the same result, their performance differs quite a lot. The main difference comes down to the number of separate database accesses they make; the more separate database accesses you do, the longer your database access will take.

Since EF Core 3.0, the default way to handle any collections found in a query has been to load the collection with the base entity. `context.Books.Include(b => b.Reviews)`, for example, would load the `Book` entity and the related `Review` entities in one database access. `Select` and eager loading queries will load the collections to the database in one call. The example queries in the following code snippets take only one database access:

```
var bookInclude = context.Books.Include(b => b.Reviews).First();

var bookSelect = context.Books.Select(b => new
{
    b.Title,
    Reviews = b.Reviews.ToList()
}).First();
```

On the other hand, explicit or lazy loading would take two database accesses. To see the effect of the different approaches on performance, load the `Book` entity with its `Reviews`, `BookAuthor`, and `Authors` (two authors) by using select/eager loading, eager loading with `AsSplitQuery` (see section 6.1.4), and explicit/lazy loading. Table 14.1 shows the results.

Table 14.1 Comparing the four ways to load data, which tells you that the more trips to the database the query makes, the longer the query will take

Type of query	#Database accesses	EF 5 time (ms) / %
`Select` and eager loading	1	1.95 / 100%
Eager loading with `AsSplitQuery`	4	2.10 / 108%
Explicit and lazy loading	6	4.40 / 225%

> **WARNING** Queries that include multiple collections with large amounts of entries will not perform well when you use the default query approach. Loading an entity with three collections, each containing 100 entries, would return 100*100*100 = 1,000,000 rows. In these cases you should add the `AsSpiltQuery` method to your query. See section 6.1.4 for details.

> **NOTE** The figures in table 14.1 were so different from the first edition of the book that I ran the old code to check my results, and EF Core 2.1 was much slower. EF Core 3.0 improved loading of collections, and NET 5 improved the time taken to access the SQL Server database.

With the improvements in EF Core, the differences between `Select`/eager, eager with `AsSplitQuery`, and explicit/lazy loading are smaller, but multiple accesses to the database still have a cost. So the rule is to try to create one LINQ query that gets all the data you need in one database access. `Select` queries are the best-performing if you need only specific properties; otherwise, eager loading, with its `Include` method, is better if you want the entity with its relationships to apply an update.

14.5.2 Antipattern: Missing indexes from a property that you want to search on

If you plan to search on a property that isn't a key (EF Core adds an index automatically to primary, foreign, or alternate keys), adding an index to that property will improve search and sort performance. It's easy to add an index to a property; see section 6.9.

There's a small performance cost to updating an index when the value of a property (column) is changed, but often, update performance cost is far smaller than the performance gain when sorting or filtering on that property. Even so, adding indexes works best if you have lots of entries to sort/filter by a property, and reads are more important than update times.

14.5.3 Antipattern: Not using the fastest way to load a single entity

When I learned EF Core, I thought that the best way to load a single entity was to use the EF Core's `Find` method. I used that method until I saw Rick Anderson, who works for Microsoft, using `FirstOrDefault`. I asked why, and he said it was quicker. At that point, I measured performance, and he was right.

Table 14.2 gives you the timings for each of the methods you could use to load a single entity, in these cases via the entity's primary key, with the timing.

Table 14.2 Time taken to read in a single book using different methods. The timing was taken by averaging the time taken to load 1,000 books. Note that there are two versions of loading via the `Find` method.

Method	Time	Ratio to single
`context.Books.Single(x => x.BookId == id)`	175 us.	100%
`context.Books.First(x => x.BookId == id)`	190 us.	109%
`context.Find<Book>(id)` (entity not tracked)	610 us.	350%
`context.Find<Book>(id)` (entity already tracked)	0.5 us.	0.3%

NOTE I couldn't find any significant performance difference between the sync and async versions or `First` or `FirstOrDefault` methods I show.

The table shows that `Single` (and `SingleOrDefault`) was fastest for a database access, and also better than using `First`, as `Single` will throw an exception if your `Where` clause returns more than one result. `Single` and `First` also allow you to use `Includes` in your query.

You should use the `Find` method if the entity is being tracked in the context, in which case `Find` will be super-fast; see the last row of table 14.2. `Find` is fast because it scans the tracked entities first, and if it finds the required entity, it returns that entity without any access to the database. The downside of this scan is that `Find` is slower if the entity isn't found in the context.

NOTE The `Find` method will return a tracked entity that hasn't yet been added or updated in the database. I use this capability in a concurrency handler (see listing 15.11) to recalculate a cached value, using the new author name that hasn't been written out to the database yet.

14.5.4 Antipattern: Allowing too much of a data query to be moved into the software side

It’s all too easy to write LINQ code that moves part of the database evaluation out of the database and into the software, often with a big impact on performance. Let’s start with a simple example.

Listing 14.4 Two LINQ commands that would have different performance times

This query would perform well, as the Where part would be executed in the database.

```
context.Books.Where(p => p.Price > 40).ToList();
context.Books.ToList().Where(p => p.Price > 40);
```

This query would perform badly, as all the books would be returned (which takes time), and then the Where part would be executed in software.

Although most people would immediately spot the mistake in listing 14.4, it’s possible for code like this listing to be hidden in some way. So if you find a query that’s taking a long time, check the parts of the query.

One big change in EF Core 3 was to use only client vs. server evaluation (see section 2.3) at the last `Select` level of a query. This situation caused problems when people updated to EF Core 3, but it exposed only LINQ queries that were running slowly. Since that change, if EF can’t translate your query to database commands, you get a `could not be translated` exception, so many bad LINQ queries are caught. The exception goes on to say

```
… or switch to client evaluation explicitly by inserting a call
to 'AsEnumerable', 'AsAsyncEnumerable', 'ToList', or 'ToListAsync'
```

This exception message is helpful, but sometimes EF Core throws a `could not be translated` exception because you didn’t get the LINQ query quite right. Aggregate LINQ methods (that is, `Sum`, `Max`, `Min`, and `Average`; see section 14.5.5) require a nullable version of the type to work, and if you don’t provide it, you will get the `could not be translated` exception. See “Aggregates need a `null` (apart from count)” in section 6.1.8. So before you add `'AsEnumerable'`, `'AsAsyncEnumerable'`, and so on, you should check for a way to make the query translate to database commands.

14.5.5 Antipattern: Not moving calculations into the database

One of the reasons why the Book App is fast is that you moved part of the calculations into the database—specifically, the count of `Reviews` and the average of the votes from the `Reviews`. If you hadn’t moved these calculations into the database, the Book App might work, but it would be slow, especially on sorting or filtering of average votes.

Typically, you won’t be able to move many calculations into the database, but the ones you do get can make a big difference, especially if you want to sort or filter on the calculated value. Here are a couple of examples of what you can do:

- `Count` a collection navigational property, such as `Book.Reviews`. This approach is useful if you need the count but don’t need the content of the collection type.

- `Sum` a value in a collection, such as summing the price of all the `LineItems` in an `Order`. This approach is useful if you want to sort the `Orders` by price.

NOTE See section 6.1.8 for LINQ commands that require special attention to make the LINQ queries translate to database commands.

14.5.6 Antipattern: Not replacing suboptimal SQL in a LINQ query

Sometimes, you know something about your data that allows you to come up with a piece of SQL code that's better than EF Core. But at the same time, you don't want to lose the ease of creating queries with EF Core and LINQ. You have several ways to add SQL calculations to the normal LINQ queries:

- *Add user-defined functions to your LINQ queries.* A scalar-valued user-defined function (UDF; see section 10.1) returns a single value that you can assign to a property in a query, whereas a table-valued UDF returns data as though it came from a table. In section 15.3, I use a scalar-valued UDF 3 to build the list of author names for a book.
- *Create an SQL View in your database that has the SQL commands to compute values.* Map an entity class to that View (see section 7.9.3) and then apply LINQ queries to that mapped entity class. This approach gives you room to add some sophisticated SQL inside the View while using LINQ to access that data.
- *Use EF Core's raw SQL methods* `FromSqlRaw` *and* `FromSqlInterpolated`. These methods allow you to use SQL to handle the first part of the query. You can follow with other LINQ commands, such as `sort` and `filter`, but read section 11.5 for the limitations of the `FromSqlRaw` and `FromSqlInterpolated` methods.
- *Configure a property as a computed column.* Use this approach if that property calculation can be done with other properties/columns in the entity class and/or SQL commands. (See listing 10.7 for some examples, and see section 10.2 for more on computed columns.)

Clearly, you need to understand and write SQL, but if you can, these techniques provide a simpler experience than using a library that works with SQL, such as ADO.NET or Dapper (see section 11.5.4).

14.5.7 Antipattern: Not precompiling frequently used queries

When you first use an EF Core query, it's compiled and cached, so if you use it again, the compiled query can be found in the cache, which saves compiling the query again. But there's a (small) cost to this cache lookup, which the EF Core method `EF.CompiledQuery` can bypass. If you have a query that you use a lot, it's worth trying, but I don't think that precompiled queries improve performance much. The other issue is that precompiled queries have some limitations that can make them hard to use:

- You can use a compiled query only if the LINQ command isn't built dynamically, with parts of the LINQ being added or removed. The `BookListFilter` method, for example, builds the LINQ command dynamically by using a switch statement, so you couldn't turn that LINQ into a compiled query.

- The query returns a single entity class—an `IEnumerable<T>` or an `IAsyncEnumerable<T>`—so you can't chain query objects as you did in chapter 2.

The `EF.CompiledQuery` method allows you to hold the compiled query in a static variable, which removes the cache lookup part. The LINQ queries can have variables in the LINQ methods, and you pass the values for these variables with the application's DbContext, as shown in the following listing.

Listing 14.5 Creating a compiled query and holding it in a static variable

```
private static Func<EfCoreContext, int, Book>
    _compiledQueryComplex =          f
    EF.CompileQuery(
        (EfCoreContext context, int i) =>
        context.Books
            .Skip(i)
            .First()
        );
```

You define a static function to hold your compiled query—in this case, the function with two inputs and the type of the returned query.

Expects a DbContext, one or two parameters to use in your query, and the returned result (an entity class or IEnumerable<TEntity>)

Defines the query to hold as compiled

The `EF.CompiledQuery` method is for taking a specific query and compiling it. In the case of the book query, you'd need to build a separate compiled query for each filter and sort option to allow each one to be compiled, as follows:

- Query books, no filter, no sort
- Query books, filter on votes, no sort
- Query books, filter on votes, sort on votes
- Query books, filter on votes, soft on publication date

The `EF.CompiledQuery` method is useful, but it's best to apply it when the query you want to performance-tune is stable because it may take some work to reformat your query in the correct form to fit the `EF.CompiledQuery` method.

14.6 Performance antipatterns: Writes

Now that you've learned about performance antipatterns that apply to queries, let's look at performance antipatterns that apply to writes. These performance issues are a mixture of patterns that produce poor performance from either the database or compute time in your application. I've listed the problems with the most likely ones first:

- Calling `SaveChanges` multiple times
- Making `DetectChanges` work too hard
- Not using `HashSet<T>` for navigational collection properties
- Using the `Update` method when you want to change only part of the entity
- Startup issue: Using one large DbContext

14.6.1 Antipattern: Calling SaveChanges multiple times

If you have lots of information to add to the database, you have two options:

- *Add one entity and call* `SaveChanges`. If you're saving 10 entities, call the `Add` method followed by a call to the `SaveChanges` method 10 times.
- *Add all the entity instances, and call* `SaveChanges` *at the end*. To save 10 entities, call `Add` 10 times (or, better, one call to `AddRange`) followed by one call to `SaveChanges` at the end.

Option 2—calling `SaveChanges` only once—is a *lot* faster, as you can see in table 14.3, because EF Core will batch multiple data writes on database servers that allow this approach, such as SQL Server. As a result, this approach generates SQL code that's more efficient at writing multiple items to the database. Table 14.3 shows the difference in time for the two ways of writing out 100 new entities to an SQL Server database on my development system.

Table 14.3 A comparison of calling `SaveChanges` after adding each entity, and adding all the entities and then calling `SaveChanges` at the end. Calling `SaveChanges` at the end is about 15 times faster than calling `SaveChanges` after every `Add`.

<table>
<tr><th>One at a time</th><th>All at once (batched in SQL Server)</th></tr>
<tr><td><pre>
for (int i = 0; i < 100; i++)
{
    context.Add(new MyEntity());
    context.SaveChanges();
}
</pre></td><td><pre>
for (int i = 0; i < 100; i++)
{
    context.Add(new MyEntity());
}
context.SaveChanges();
</pre></td></tr>
<tr><td>Total time = 160 ms</td><td>Total time = 9 ms</td></tr>
</table>

The difference between the two ways of saving multiple entities can be large. In the extreme example in table 14.3 where `SaveChanges` is called 100 times (left side), the time taken is more than 15 times slower than calling `SaveChanges` once (right side).

Some of the performance loss of the "one at a time" approach is due to extra database accesses. The "all at once" approach taps EF Core's batching capability, which produces SQL that performs well when adding lots of data to a database. For a detailed look at this topic, see http://mng.bz/ksHg.

> **NOTE** It's also not a good practice to call `SaveChanges` after each change, because what happens if something goes wrong halfway through? The recommendation is to do all your additions, updates, and removals and then call `SaveChanges` at the end. That way, you know that all your changes were applied to the database or that if there was an error, none of the changes were applied.

14.6.2 Antipattern: Making DetectChanges work too hard

Every time you call `SaveChanges`, by default it runs a method inside your application's DbContext called `ChangeTracker.DetectChanges` to see whether any of the tracked entities has been updated. (See section 9.3.3 for details.) The time `DetectChanges` takes to run depends on how many tracked entities are loaded—that is, the number of entities you read in without the `AsNoTracking` or `AsNoTrackingWithIdentityResolution` method (see section 6.1.2) that don't implement the `INotifyPropertyChanged` interface (see section 11.4.2).

Table 14.4 shows the time taken for different levels of tracked entities. In this case, the entities are small, with a few properties; if the tracked entities were more complex, the time would be larger.

Table 14.4 Time taken by the `SaveChanges` method, which contains the call to the `DetectChanges.Detect` method, to save one entity for different levels of tracked entities. Note that the tracked entities used in this table are small.

Number of tracked entities	How long `SaveChanges` took	How much slower?
0	0.2 ms.	n/a
100	0.6 ms.	2 times slower
1,000	2.2 ms.	11 times slower
10,000	20.0 ms.	100 times slower

This sort of problem has various solutions, depending on the design of your application. Here are ways to solve this sort of performance issue:

- Do you need all these tracked entities loaded? If `SaveChanges` is taking a long time, did you forget to use the `AsNoTracking`/`AsNoTrackingWithIdentityResolution` method when you made read-only queries?
- Can you break a big insert into smaller batches? I do this in chapter 15, where I build a class to create large test data sets for performance tests. In that class, I write out in batches of ~700 Books and use a new instance of the application's DbContext so that there aren't any tracked entities.
- When you need a lot of entities loaded that are ready to be modified, consider changing your entity classes to use the `INotifyPropertyChanged` change tracking strategy. This change requires extra coding of your entity classes to add the `INotifyPropertyChanged` and configure the entity class's change tracking strategy (see section 11.4.2). The result is that your entities will report any changes to EF Core, and `DetectChanges` doesn't have to scan your loaded entities for changes.

14.6.3 Antipattern: Not using HashSet<T> for navigational collection properties

In section 6.2.2, you learned that when you call the `Add` method to add a new entity to the database, EF Core runs a series of steps to ensure that all the relationships are set up correctly. One of the steps EF Core runs, called *relational fixup*, checks whether any of the entities in the added entity are already tracked. (See section 6.2.2 for details.)

From a performance point of view, the relational fixup stage can cost you compute time because it must compare all the tracked entities that are used in the added entity and its relationships. It's hard to get reliable timings because the first few uses of the application's DbContext are slow, but here are some observations:

- When you're loading collection navigational properties in a query—say, by using the `Include` method—`HashSet<T>` for collections is quicker than collection navigational properties using `ICollection<T>` / `IList<T>`. Adding an entity with 1,000 entities in a collection navigational property, for example, took 30% longer with `ICollection<T>` than using `HashSet<T>` because it is easier to detect/find instances in a `HashSet<T>`.
- The more tracked entities of the same type found in the entity (and its relationships) that was added, the more time it takes to check them all. The performance hit is hard to measure but seems to be small. But if you have issues with an `Add` taking a long time, it's worthwhile to check for a lot of tracked entities, which may be part of the slowness of your `Add` method call.
- As I said in section 2.1.3, the downside of using `HashSet<T>` is that it does not guarantee the order of the entries in the collection. So if you are using EF Core 5's ability sort entries in an `Include` method, you can't use `HashSet<T>`.

14.6.4 Antipattern: Using the Update method when you want to change only part of the entity

EF Core is great at detecting changes to individual properties in an entity class using the `DetectChanges.Detect` method. If you change one property, such as the publication date of a book, and then call `SaveChanges`, the `DetectChanges.Detect` method will find that property change, and EF Core will create some SQL to update that single column in the correct row of the Books table.

On the other hand, if you use the `Update` method on the `Book` entity, all the properties are marked as changes, and the SQL becomes bigger and takes (a bit) longer to execute. The `Update` method should be used only when the whole entity has changed; see section 11.3.4 for an example.

14.6.5 Antipattern: Startup issue—Using one large DbContext

The first time you create your application's DbContext, it'll take some time, perhaps several seconds. There are many reasons for this slowness, but one of them is that EF Core needs to scan all the entity classes in the application's DbContext to configure itself and build a model of the database you want to access. Normally, this problem isn't a big one, because after your application is running, the configuration and database model information is cached by EF Core. But if your application is constantly being started and stopped—say, in a serverless architecture (see https://martinfowler.com/articles/serverless.html)—this startup time could matter.

You can help speed the building of the first application's DbContext by reducing the number of entity classes it includes. The only reasonable way to do that is to produce multiple application DbContexts, with each one covering a subset of the tables in the database. Section 13.4.8 covers splitting a database across multiple DbContexts based on the DDD approach bounded contexts. Figure 14.6 illustrates how a large database could be split across multiple applications' DbContexts.

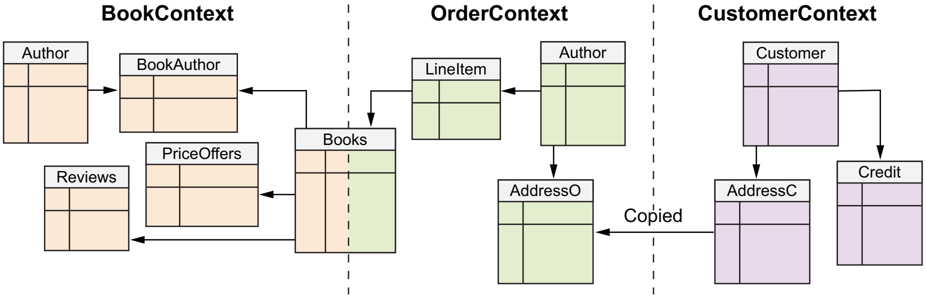


Figure 14.6 **A large database can be split into multiple applications' DbContexts. In this case, the database is split along business lines. If you need to minimize application startup costs, you could create specific DbContexts for each application containing only the entities that the application needs to access.**

Figure 14.6 splits the database across different applications' DbContexts based on the business domains, which might be an appropriate split for some applications. If you're building small, self-contained applications, such as in a serverless architecture or a microservices architecture (see https://martinfowler.com/articles/microservices .html), you could build an application's DbContext, including only the entities/tables specific to each application.

14.7 Performance patterns: Scalability of database accesses

Scalability of an application (the number of simultaneous accesses that the application can handle) is a big topic. Even when limiting the scope to database access scalability, you still have a lot of things to think about. Scalability issues typically can't be tracked to a poorly written piece of code, because scalability is more about design. This section covers

- Using pooling to reduce the cost of creating a new application's DbContext
- Adding scalability with little effect on overall speed

- Helping your database scalability by making your queries simple
- Scaling up the database server
- Picking the right architecture for applications that need high scalability

14.7.1 Using pooling to reduce the cost of a new application's DbContext

If you're building an ASP.NET Core application, EF Core provides a method called `AddDbContextPool<T>` that replaces the normal `AddDbContext<T>` method. The `AddDbContextPool<T>` method uses an internal pool of an application's DbContext instances, which it can reuse. This method speeds your application's response time when you have lots of short requests.

But be aware that you shouldn't use it in some situations. When you're passing in data based on the HTTP request, such as the logged-in user's ID, you shouldn't use DbContext pooling because it would use the wrong user ID in some instances of the application's DbContext. DbContext pooling is simple to use, and this listing shows an updated registration of the `EfCoreContext` context in the Book App.

Listing 14.6 Using `AddDbContextPool` to register the application's DbContext

```
services.AddDbContextPool<EfCoreContext>(
    options => options.UseSqlServer(connection,
    b => b.MigrationsAssembly("DataLayer")));
```

You're using an SQL Server database, but pooling works with any database provider.

You register your application DbContext by using the AddDbContextPool<T>.

Because you're using migrations in a layered architecture, you need to tell the database provider which assembly the migration code is in.

Whether DbContext pooling makes a significant difference to the scalability of your application depends on the type of concurrent traffic you have. But you should get at least a small improvement in speed, as the `AddDbContextPool<T>` method will be quicker at returning a fresh application's DbContext instances.

14.7.2 Adding scalability with little effect on overall speed

In section 14.4.5, I said that you should use the async versions of the database access methods in an application that must handle multiple simultaneous requests because async/await releases a thread to allow other requests to be handled while the async part is waiting for the database to respond (see figure 5.8). But using an async method instead of the normal, synchronous method does add a small amount of overhead to each call. Table 14.5 lists performance figures for a few types of database accesses.

The differences between sync and async in table 14.5 are small, but there is a difference: the slow queries need async, as it releases a thread for a long time. But the fact that the fastest queries have the smallest sync/async difference says that using async won't penalize the small queries. Overall, you have plenty to gain and little downside from using async/await.

Table 14.5 Performance for a mixture of types of database access returning books, using sync and async versions. The database contains 1,000 books.

Type of database access	#DB trips	Sync	Async	Difference
Read book only, simple load	1	0.7 ms.	0.8 ms.	112%
Read book, eager-load relationships	1	9.7 ms.	13.7 ms.	140%
Read book, eager-load relationships+sort and filter	1	10.5 ms.	14.5 ms.	140%

14.7.3 Helping your database scalability by making your queries simple

Creating SQL commands that have a low cost on the database server (they're easy to execute and return a minimal amount of data) minimizes the load on the database. Performance-tuning your key queries to be simple and return only the data needed not only improves the speed of your application, but also helps with the scalability of your database.

14.7.4 Scaling up the database server

With the move to using cloud databases, you can increase the performance of your database with the click of a button (and a credit card!). You have are so many options (Azure has more than 50 options for SQL Server) that it's not hard to balance performance and cost.

14.7.5 Picking the right architecture for applications that need high scalability

Section 5.2 details how a web application can have multiple instances to provide more scalability. Running multiple instances of your web application is helpful for software/compute performance, but if all the web application instances are accessing only one database, it doesn't necessarily help the database scalability.

Although software/compute performance is normally the bottleneck in scalability, for applications that make high demands on the database, extra instances of the web application won't help much. At this point, you need to think about other architectures. One approach, called *sharding*, spreads your data across multiple databases, which can work for certain types of multitenant applications. In chapters 15 and 16, you will explore two architectural approaches—caching and the CQRS pattern—that improve performance and scalability.

Because most applications read the database more than they write to the database, the CQRS architecture can help with database performance. In addition, by splitting out the read-only queries to a NoSQL database called Cosmos DB, you can make the replication of the read-only databases easier, which gives you more database bandwidth. I implement such an architecture by using a CQRS approach in chapter 16, with impressive performance gains.

Summary

- Don't performance-tune too early; get your application to work properly first. But try to design your application so that if you need to performance-tune later, it's easier to find and fix your database code.
- Performance tuning isn't free, so you need to decide what performance issues are worth the development effort to fix.
- EF Core's log output can help you identify database access code that has performance issues.
- Make sure that any standard patterns or techniques you use in writing your application perform well. Otherwise, you'll bake in performance issues from day one.
- Avoid or fix any database performance antipatterns (database accesses that don't perform well).
- If scalability is an issue, try simple improvements, but high scalability may need a fundamental rethinking of the application's architecture.
- Chapter 15 provides an example of applying the recommendations in this chapter to improve the performance of the Book App.

For readers who are familiar with EF6:

- Some EF6.x performance issues, such as using the `AddRange` method over repeated `Add` method calls, have been fixed in EF Core.

15

Master class on performance-tuning database queries

This chapter covers

- Understanding four different approaches to performance-tuning EF Core queries
- Comparing the different performance gains each approach provides
- Extracting the good practices from each approach to use in your applications
- Evaluating the skills and development effort needed to implement each approach
- Understanding what database scalability is and how to improve it

Chapter 14 provided lots of information on how to performance-tune an application. In this chapter and part of chapter 16, you are going to see how quickly you can make the part 3 Book App display books. This information will expose you to various ways to performance-tune an EF Core application; each approach involves a balance between better performance and extra development time. By learning a range of approaches, you'll be ready to decide what you need in your own applications.

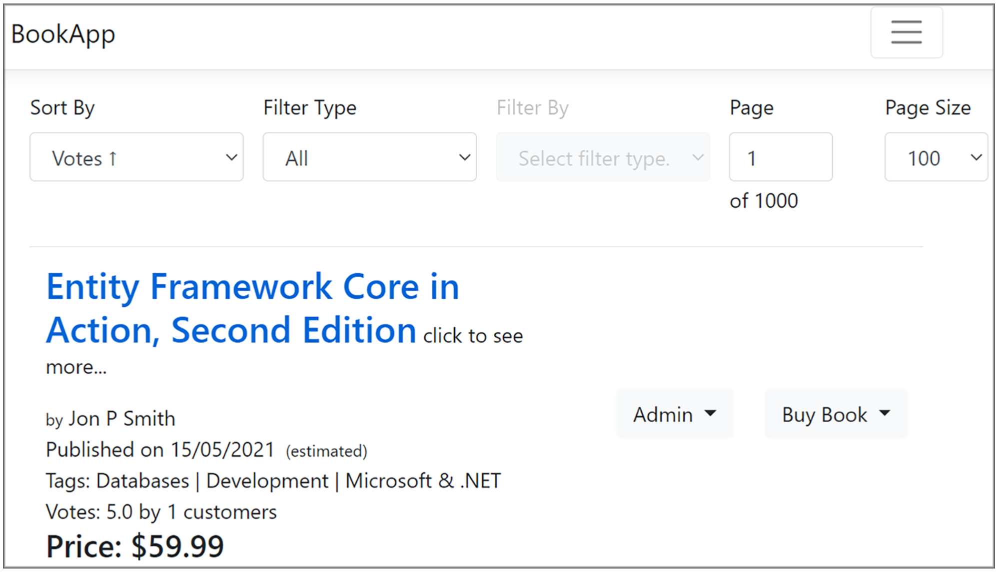
You will apply different performance-tuning approaches that progressively increase the speed of the Book App while taking more and more development effort to achieve these performance gains. Although the specific performance code in the Book App may not apply to your application, each of the performance changes uses a different methodology, so you can adapt the approach that works for you.

This chapter focuses on read-only queries, which are often the main performance problem areas in applications. For database writes, see section 14.6.

15.1 The test setup and a summary of the four performance approaches

Before we can performance tune an application, we need some example data to test against. Sometimes, the data comes from an existing application that is showing performance problems, or maybe your design/management team has set some performance targets. But to improve performance, you need test data that's representative of the real data that you would encounter in the real world.

For part 3 of this book, I reached out to Manning Publications (this book's publisher), which provided a set of real data containing about 700 real books. Figure 15.1 shows this book's information in the Book App. (Clicking the book title takes you to a details page with further data and an image of the cover.)



Sort By: Votes ↑
Filter Type: All
Filter By: Select filter type.
Page: 1 of 1000
Page Size: 100

Entity Framework Core in Action, Second Edition click to see more...

by Jon P Smith
Published on 15/05/2021 (estimated)
Tags: Databases | Development | Microsoft & .NET
Votes: 5.0 by 1 customers
Price: $59.99

Figure 15.1 The upgraded BookApp.UI, using real book data from Manning Publications. This figure shows an example of the Book App using real book data provided by Manning, which has been duplicated so that there are 100,000 Books in the database, as well as more than a half-million Reviews.

NOTE You can try this example yourself by downloading the GitHub repo associated with this book (http://mng.bz/XdlG) and then selecting the `Part3` branch. The BookApp.UI project contains the ASP.NET Core application. When you run this application, see the Things to Do section of the home page for a link to information on configuring the app to show the four approaches used in this chapter.

The four approaches used in this chapter to performance-tune are

- *Good LINQ*—Uses the same approach shown in section 2.6 and follows the suggestions in chapter 14. This approach is our base performance.
- *LINQ+UDFs*—Combines LINQ with SQL UDFs (user-defined functions; see section 10.1) to move concatenations of `Author's Names` and `Tags` into the database.
- *SQL+Dapper*—Creates the required SQL commands and then uses Dapper to execute that SQL to read the data.
- *LINQ+caching*—Precalculates some of the costly query parts, such as the averages of a `Review`'s `NumStars` (referred to as *votes*).

To provide a more demanding set of data to test these approaches, use the Book App's `BookGenerator` to duplicate the initial 700 books to get as many as you like. For the tests in this chapter, I used 100,000 books. Table 15.1 shows the full list of the data in the database.

Table 15.1 The test data used in this chapter to test the four performance approaches

Table	Books	Review	BookAuthor	Authors	BookTags	Tags
Number of rows	100,000	546,023	156,958	868	174,405	35

In this chapter, you are going to compare the performance of these four query approaches with that of three different book-display queries. The three queries range from the simple sort-by-date query to the complex sort-by-votes query. Figure 15.2 shows the time taken for each query approach for each of the three queries, using test data detailed in table 15.1.

Here is a detailed explanation of the three types of queries shown in figure 15.2:

- *Sort by votes*—Sorts by average votes, which is calculated by averaging the `NumStars` property in the `Reviews` linked to a book. This query shows that sorting on the average votes, which is a sort that users would use a lot, has a large amount of variation across all four approaches, for reasons explained in the sections on each approach.
- *Filter votes+sort price*—Filters out all books with fewer than 4 for its average vote (which leaves about 3,000 books) and then sorts on price. This query shows that the first three approaches take a similar amount of time. The cached version is fast because the average vote is precalculated and has an SQL index.

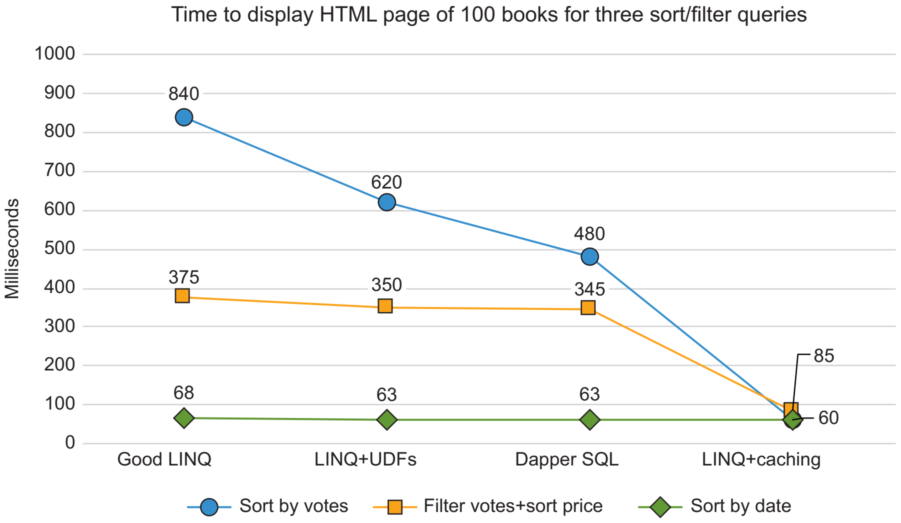


Figure 15.2 The chart shows the time it took to display a page containing 100 books for three different sorts/filters. The database contains 100,000 books and a half-million reviews (see table 15.1 for full details). The timings were done on my local PC, using a localdb SQL Server running on the same PC; all the queries are async.

- *Sort by date*—Sorts by the date of publication of the book, which is a sort on a known property that has an SQL index. All the approaches provide good performance, with some subtle differences between the Good LINQ approach and the SQL+UDFs and Dapper SQL approaches.

Although these four approaches are applied to the Book App, they define four general approaches to performance-tuning EF Core database queries. The explanation of each approach details the application of the performance improvements in the Book App, but then pulls out the learning from each approach so that you can decide whether it would work in your EF Core application.

15.2 Good LINQ approach: Using an EF Core Select query

This approach is close to the book query you created in chapter 2. The great thing about this approach is that it's simple: the query uses only LINQ to build this version, whereas the LINQ+SQL and Dapper versions require you to use raw SQL, and the cached SQL requires some serious code to make it work.

First, I should say that the current LINQ query is fast enough with only 700 books; it takes about 70 ms to sort on votes and displays 100 books. The reason is that the query from chapter 2 already uses some good practices. I didn't call out these good practices in chapter 2 because that chapter was early in the book, but now we can explore this query in detail.

NOTE If you download and run the `Part3` branch Book App, you can see the SQL generated by each approach by selecting the approach and type of filter/sort and then clicking the Logs menu item, which will show you the SQL as used in the query you executed.

The following listing shows the part of the query that gathers all the data needed, with comments on various parts that make this query a good LINQ query.

Listing 15.1 `MapBookToDto` method that selects what to show in the book display query

```
public static IQueryable<BookListDto>
public static IQueryable<BookListDto>
    MapBookToDto(this IQueryable<Book> books)
{
    return books.Select(p      => new BookListDto
    {
        BookId                 = p.BookId,
        Title                  = p.Title,
        PublishedOn            = p.PublishedOn,
        EstimatedDate          = p.EstimatedDate,
        OrgPrice               = p.OrgPrice,
        ActualPrice            = p.ActualPrice,
        PromotionText          = p.PromotionalText,
        AuthorsOrdered         = string.Join(", ",
            p.AuthorsLink
                .OrderBy(q     => q.Order)
                .Select(q      => q.Author.Name)),
        TagStrings             = p.Tags
            .Select(x => x.TagId).ToArray(),
        ReviewsCount           = p.Reviews.Count(),
        ReviewsAverageVotes    =
            p.Reviews.Select(y =>
                (double?)y.NumStars).Average(),
        ManningBookUrl         = p.ManningBookUrl
    });
}
```

Good practice: Load only the properties you need.

Good practice: Use indexed properties to sort/filter on (in this case, the ActualPrice).

Good practice: Don't load the whole entity of each relationships, only the parts you need.

Good practice: The ReviewsCount and ReviewsAverageVotes are calculated in the database.

Next, let's look at what is going in the `MapBookToDto` extension method so that you can understand and apply these good practices to your own applications.

LOADING ONLY THE PROPERTIES YOU NEED FOR THE QUERY

You could have loaded the whole `Book` entity, but that would mean loading data you didn't need. The Manning Publications book data contains large strings summarizing the book's content, what technology it covers, and so on. The book display doesn't need that data, however, and loading it would make the query slower, so you don't load it.

In line with the recommendation that you don't performance-tune too early, you might start with a simple query that reads in the entity classes, and performance-tune later. In the Book App, it was obvious that the book display query was a key query, especially with respect to sorting by votes, so I started with a `Select` query. But for you, if a query is slow and you are loading the whole entity class, consider changing to the LINQ `Select` method and loading only the properties you need.

DON'T LOAD WHOLE RELATIONSHIPS—ONLY THE PARTS YOU NEED

There are many ways to load relationships, including eager loading, explicit loading, and lazy loading. The problem is that these three approaches to reading relationships load the whole entity class of each relationship. Typically, you don't need to load the relationship's whole entity classes.

In listing 15.1, you see that the `AuthorLink` collection is used to select only the `Author's Name`, which minimizes the data returned from the database. Similarly, the `Tags` are stripped to return only an array of the `TagIds`. So to improve the performance of a query, if you need data from relationships, try to extract the specific parts from any relationships. An even better idea is to move calculations into the database if you can, which I cover next.

IF POSSIBLE, MOVE CALCULATIONS INTO THE DATABASE

If you want good performance, especially for sorting or filtering on values that need calculating, it's much better for the calculation to be done inside the database. Calculating data inside the database has two benefits:

- The data used in the calculation never leaves the database, so less data needs to be sent back to the application.
- The calculated value can be used in a sort or filter, so you can execute the query in one command to the database.

If you didn't calculate the `ReviewsAverageVotes` value in the database, for example, you would need to read in *all* the `Reviews` `NumStars` and `BookId` properties, and work out the `ReviewsAverageVotes` value for every book. Only then could you work out which `Books` you should read in. That process is going to be slow and take up a lot of memory because it would have to read in all the `Reviews` from the database and then work out the average votes in software before it could read in the `Books` to display.

I have to say that getting these types of calculations right wasn't obvious! When I wrote the first edition of this book, I couldn't get the `ReviewsAverageVotes` value query correct, and it took raising an issue on the EF Core GitHub issues page to get the right answer. In section 6.1.8, I cover some of the LINQ commands that must be written in a specific way to work.

IF POSSIBLE, USE INDEXED PROPERTIES TO SORT/FILTER ON

In part 1, I applied a promotion to a `Book` by adding a `PriceOffer` entity class. I did that not only because I wanted to show how one-to-one relationships worked, but also because using a `PriceOffer` entity class made it obvious what I was doing. The downside of this approach is that the query had to include code to look for the `PriceOffer` entity class. The following code snippet is from the part 1 version of the `MapBookToDto` method:

```
ActualPrice = book.Promotion == null
    ? book.Price
    : book.Promotion.NewPrice,
```

```
PromotionPromotionalText =
    book.Promotion == null
    ? null
    : book.Promotion.PromotionalText,
```

That code has two negative effects on sorting on price: the LINQ is converted to an SQL `JOIN` to find the optional PriceOffers row, which takes time, and you can't add a SQL index to this calculation. In part 3, the Book App moved to using DDD, so you could add or remove a price promotion by using access methods in the `Book` entity (see section 13.4.2). The access methods hide the business logic of the promotion, which means that the `ActualPrice` property always contains the price that the book is sold for. Changing the code to not use the `PriceOffer` entity removes the SQL `JOIN`, and you can add an SQL `INDEX` to the ActualPrice column in the database, significantly improving the sort-on-price feature.

So if you need to query some data, especially if you're sorting or filtering on that data, try to precompute the data in your code. Or use a persisted computed column (see section 10.2) if the property is calculated based on other properties/columns in the same entity class, such as `[TotalPrice] AS (NumBook * BookPrice)`. That way, you will get a significant improvement in any sort or filter because of the SQL index on that column.

15.3 LINQ+UDFs approach: Adding some SQL to your LINQ code

In the Good LINQ approach, both the LINQ that forms the book display reads in `Authors Names` and the `Tag`'s `TagId` return collections, because there can be many `Authors` and `Tags`. Before EF Core 3.0, these collections were read in by using an extra query per collection, so reading in 100 books with `Author's Name` alone would create 101 accesses to the database (one for the main query and then one per book for `Author's Name`) and take about 230 ms.

Since EF Core 3.0, this query has been reduced to one access to the database by returning multiple rows per book and extra columns to make sure that the rows are in the right order. With lots of `Books`, `Author's Names`, and `TagIds`, the end of the SQL produced by the Good LINQ book display with the default ordering (order on `BookId` descending) looks like this:

```
SELECT [t].[BookId],…
-- other parts of the SQL
ORDER BY [t].[BookId] DESC
   ,[t0].[Order]
   ,[t0].[BookId], [t0].[AuthorId], [t0].[AuthorId0]
   ,[t2].[BookId], [t2].[TagId0], [t2].[TagId]
```

I'm not going to explain the various tables and columns in `ORDER BY` (you can see the whole SQL query by running the Book App and clicking the Logs menu item), but you can see that there are a lot of `ORDER BY` parameters. It turns out that if you add the sort-on-average-votes LINQ query at the top of the existing `ORDER BY`s, performance starts to drop, which is one reason why the Good LINQ book display is so bad (840 ms, as shown in figure 15.2).

> **NOTE** Before you say that having all those `ORDER BY` parameters is bad SQL, I can tell you that without that code, the query would take about twice the time and would go from one database access to five separate database accesses. The EF Core 3.0 change has improved most, but not all (see section 6.1.4) queries containing collections.

Some time ago, I found some SQL code on Stack Overflow that concatenated a series of strings into a single string inside the database. In section 14.5.6, I described four ways to enhance a LINQ query by providing SQL that is custom-made for your specific situation. In this case, I used a scalar UDF to access this code, as shown in the following code snippet:

```
CREATE FUNCTION AuthorsStringUdf (@bookId int)
RETURNS NVARCHAR(4000)
AS
BEGIN
-- Thanks to https://stackoverflow.com/a/194887/1434764
DECLARE @Names AS NVARCHAR(4000)
SELECT @Names = COALESCE(@Names + ', ', '') + a.Name
FROM Authors AS a, Books AS b, BookAuthor AS ba
WHERE ba.BookId = @bookId
      AND ba.AuthorId = a.AuthorId
      AND ba.BookId = b.BookId
ORDER BY ba.[Order]
RETURN @Names
END
```

> **NOTE** You should add raw SQL only if you have code that does something better than EF Core. Merely adding SQL that is the same as what EF Core would have created won't improve performance.

To use the UDF code `AuthorsStringUdf` and `TagsStringUdf` to concatenate the `TagIds`, I had to define it (see section 10.1) and add the UDFs to a database by editing a migration (see section 9.5.2). Then I needed to create a new mapping from the `Book` entity to a book-display DTO, as shown in the next listing. See the lines with comments for the calls to the two UDFs.

Listing 15.2 `MapBookUdfsToDto` using UDFs to concatenate `Name`/`Tag` names

```
public static IQueryable<UdfsBookListDto>
    MapBookUdfsToDto(this IQueryable<Book> books)
{
    return books.Select(p      => new UdfsBookListDto
    {
        BookId         = p.BookId,
        Title          = p.Title,
```

Updated MapBookToDto method, now called MapBookUdfsToDto

```
        PublishedOn     = p.PublishedOn,
        EstimatedDate   = p.EstimatedDate,
        OrgPrice        = p.OrgPrice,
        ActualPrice     = p.ActualPrice,
        PromotionText   = p.PromotionalText,
        AuthorsOrdered = UdfDefinitions
            .AuthorsStringUdf(p.BookId),
        TagsString      = UdfDefinitions
            .TagsStringUdf(p.BookId),
        ReviewsCount = p.Reviews.Count(),
        ReviewsAverageVotes =
            p.Reviews.Select(y =>
                (double?)y.NumStars).Average(),
        ManningBookUrl = p.ManningBookUrl
    });
}
```

The AuthorsOrdered and TagsString are set to the strings from the UDFs.

When you change the `MapBookToDto` extension method to use the `AuthorsStringUdf` and the `TagsStringUdf` UDFs, each book returns only one row, and there is no `ORDER BY` other than the default ordering on `BookId`, descending. This change has a small effect on a nonsorted display of 100 books (improving it by a few milliseconds), but the big effect is on the sort by average votes, which comes down from 840 ms in the Good LINQ approach to 620 ms in the LINQ+SQL approach—an improvement of about 25%.

15.4 SQL+Dapper: Creating your own SQL

The ultimate SQL approach is to stop using EF Core and write your own SQL query. If you want to do this, you need a library that can execute your SQL code for you. The best one I have found is Dapper (covered in section 11.5.4). The issue is coming up with better SQL than EF Core.

I studied the SQL that EF Core produced and did some digging, and found one place where I could improve the SQL over EF Core. It turns out that you can sort on a parameter in a SQL `SELECT` command; see https://stackoverflow.com/a/38750143/1434764. According to this Stack Overflow page, "`ORDER BY` is solved after the `SELECT` (which means you can use a calculated column from the `SELECT`), unlike `WHERE` or `FROM`, which are solved before the `SELECT` and therefore can't refer to column aliases in SQL Server."

EF Core doesn't take advantage of this feature, so its SQL computes average votes twice: once in the `SELECT` and again in the `ORDER BY`. My tests showed that computing average votes only once significantly improved the performance in the sort-by-votes query, so I set about rewriting the various sort, filter, and paging features used by the Book App, which involved selecting and concatenating SQL strings to form the correct SQL query. Converting the LINQ features to SQL was quite complicated. Figure 15.3 shows a flow chart depicting how the SQL query was built.

NOTE The SQL that I created uses the two UDFs used in the LINQ+UDFs approach; otherwise, it would be slower than LINQ+UDFs. If you want to see

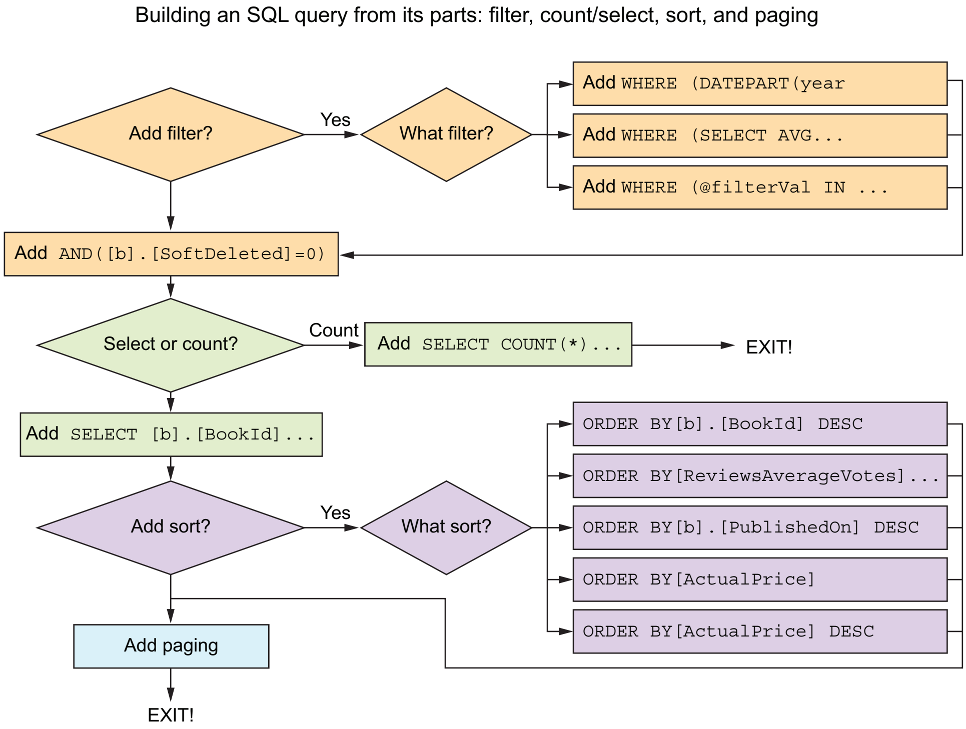


Figure 15.3 The Dapper code consists of a series of string concatenations that produce the final SQL query. This code isn't as elegant as the EF Core version, with its four Query Objects, but when you're performance-tuning, you often must accept some loss of cleanness from your original code to achieve the performance you need.

the code that builds and runs the SQL, you can find it at http://mng.bz/n2Q2.

The performance improvement for the sort-by-votes query is impressive: the Dapper version is nearly twice as fast as the Good LINQ version (Dapper: 480 ms, Good LINQ: 840 ms). But on every other query that didn't include a sort on votes, the Dapper version wasn't much faster than LINQ version, especially against LINQ+UDFs. To understand this result, I looked at the simplest query—sort by the date of publication—to see where the time was going. Figure 15.4 breaks down the time into three parts:

- (Bottom) *Database time* (important)—Time taken for the SQL to run
- (Middle) *HTML time*—Time it took to send the HTML page to the browser
- (Top) *Software time*—Rest of the time, mostly ASP.NET Core

NOTE The SQL timing came from EF Core's logging, which includes the time the execution took. For Dapper, I used a stopwatch, starting it before the call to Dapper and stopping it when the data was returned.

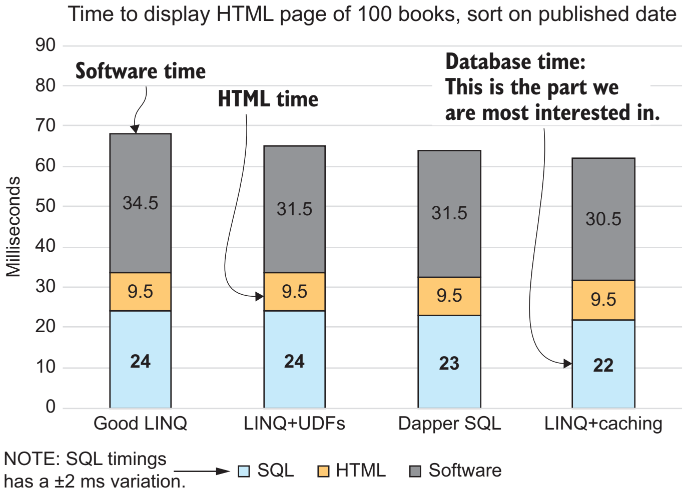


Figure 15.4 Breakdown of the sort by the date of publication of the book with a page of 100 books. The important part to look at is the bottom timings, which cover the time taken to execute the SQL: ±2 ms on the SQL part, with some outliers that I left out. The other parts have larger variations. The overall variation is 10 ms for the Good LINQ version and smaller (say, 5 ms).

As you can see from figure 15.4, the differences in the SQL are small, and because of the ±2 ms variations in the timings, they are essentially the same. The quick performance of the Dapper library becomes less and less a factor when the SQL used in the query takes many milliseconds to execute. And because the only queries you need to performance-tune typically take many milliseconds to run, ½ or 1 ms saved by Dapper doesn't make much of a difference. (It helps that EF Core is getting quicker.)

The takeaway from figure 15.4 is that it's worth converting your slow queries to Dapper only if you can find some SQL that is better than what EF Core produces. It took me quite a bit of time to build and debug the complex book display, and if I hadn't had another way to improve the performance, the effort would have been worthwhile. The cached SQL approach (section 15.5) provides a much bigger performance improvement, but it's a lot more work.

NOTE To be clear, other than the sort-by-votes issue, I didn't find any other part of the EF Core that would be improved by using Dapper, and EF Core already had issue #16038 to solve this problem.

15.5 LINQ+caching approach: Precalculating costly query parts

The final approach in this chapter is precalculating the parts of the query that take a long time to calculate and storing them in extra properties/columns in the `Book` entity class. This technique is known as *caching* or *denormalization*. Caching works best with data that is expensive to generate, such as the average votes for a `Book`. As you saw in figure 15.2, caching has the biggest effect on the sort-by-votes query, making it about 14 times faster than the Good LINQ approach and 8 times faster than the Dapper approach.

But when you're thinking about using caching, you also need to think about how often the cached value is updated and how long it takes to update the cache. If the data that is cached is updated a lot, the cost of updating the cache may move the performance problem from running the query to updating entities. As you will see in section 15.5.2, the design of the caching algorithms used in the Book App is quick when it comes to handling updates.

But the main problem with caching is that it's really hard to make sure your cached values are up to date. Under the caching SQL approach, for example, you must update the cached `ReviewsAverageVotes` property every time a `Review` is added, updated, or deleted. And what happens if two `Reviews` are applied to a `Book` entity simultaneously, or when the database update of the cached `ReviewsAverageVotes` property fails? Here's a quote from the 1990s stating that cache updates have always been a problem:

> *There are only two hard things in computer science: cache invalidation and naming things.*
>
> —Phil Karlton (while at Netscape)

I can attest that building a caching system is hard. I built a caching system for the first edition of the book, and it was good, but now I know about one rare situation in which it would fail to update a cached properly. (I fixed this problem in the new version for this book.)

Studying the SQL query shows that caching the average votes (the average of the `NumStars` in all `Reviews` linked to a specific `Book` entity) would improve performance on sort/filter on average votes. You could stop there, but caching the number of `Reviews`, the `Book`, and the concatenation of `Author's Names` would provide a small boost for all displays of books (about a 5 ms performance gain for displaying 100 books).

Adding a caching system isn't trivial to implement. Here are the steps:

1. Add a way to detect changes that affect the cached values.
2. Add code to update the cached values.
3. Add the cache properties to the `Book` entity and provide concurrency code to handle simultaneous updates of the cached values.
4. Build the book display query to use the cached values.

At the end of the description of this caching system, section 15.5.4 describes a checking/healing system that checks whether the cached values are set properly.

15.5.1 Adding a way to detect changes that affect the cached values

I have had good results from a domain events approach (see chapter 12) to implement caching, so this design uses that approach. One positive feature of the domain events approach is that the change that triggers an update of a cached value is saved in the same transaction that saves the cached value (see figure 12.3). As a result, both changes are applied to the database, or if anything fails, none of the updates are applied to the database. That approach prevents the real data and cached data from getting out of step (known as a dirty cache).

As for detecting a change of properties or relationships, we can take advantage of the fact that the part 3 Book App uses the DDD design approach. So, to update the two cached values related to the `Reviews`, you can add code to the `Book`'s `AddReview` and `RemoveReview` access methods.

For the cached property called `AuthorsOrdered`, we are going to use a non-DDD approach to trigger a domain event in which an `Author's Name` is changed. This example shows how you would handle domain events and caching when you're not using DDD.

To speed the development, you are going to use my `EfCore.GenericEventRunner` library. This library is well tested and contains other features that will speed development. So let's see what the code would look like, starting with the event-enhanced `BookDbContext`, as shown in the following listing.

Listing 15.3 `BookDbContext` updated to use `GenericEventRunner`

```
public class BookDbContext
    : DbContextWithEvents<BookDbContext>
{

    public BookDbContext(
        DbContextOptions<BookDbContext> options,
        IEventsRunner eventRunner = null)
        : base(options, eventRunner)
    { }

    //… rest of BookDbContext is normal, so left out
}
```

The BookDbContext handles the Books side of the data.

Instead of inheriting EF Core's DbContext, you inherit the class from GenericEventRunner.

The constructor of the DbContext-WithEvents class needs the Event-Runner.

DI will provide GenericEventRunner's EventRunner. If null, no events are used (useful for unit tests).

The next stage is adding the events to the `Book`'s `AddReview` and `RemoveReview` access methods. The following listing shows how these methods create an event.

Listing 15.4 The `Book` entity with the `AddReview` and `RemoveReview` methods

```
public class Book : EntityEventsBase,
    ISoftDelete
{

     //… other code left out for clarity
```

Adding the EntityEventsBase will provide the methods to send an event.

```
    public void AddReview(int numStars,
        string comment, string voterName)
    {
        if (_reviews == null)
            throw new InvalidOperationException(
                "The Reviews collection must be loaded");

        _reviews.Add(new Review(
            numStars, comment, voterName));

        AddEvent(new BookReviewAddedEvent(numStars,
            UpdateReviewCachedValues));
    }

    public void RemoveReview(int reviewId)
    {
        if (_reviews == null)
            throw new InvalidOperationException(
                "The Reviews collection must be loaded");

        var localReview = _reviews.SingleOrDefault(
            x => x.ReviewId == reviewId);
        if (localReview == null)
            throw new InvalidOperationException(
                "The review was not found.");

        _reviews.Remove(localReview);

        AddEvent(new BookReviewRemovedEvent(localReview,
            UpdateReviewCachedValues));
    }

    private void UpdateReviewCachedValues
         (int reviewsCount, double reviewsAverageVotes)
    {
        ReviewsCount = reviewsCount;
        ReviewsAverageVotes = reviewsAverageVotes;
    }
}
```

The AddReview is the only way to add a Review to this Book.

Adds a BookReview-AddedEvent domain event with the NumStars of the new Review

Provides the event handler a secure way to update the Review cached values

The RemoveReview method is the only way to remove a Review from this Book.

Adds a BookReview-AddedEvent domain event with the review that has been deleted

This private method can be used by the event handlers to update the cached values.

To catch a change of an `Author's Name`, we will use a non-DDD approach and intercept the setting of a property. This approach uses EF Core's backing-field feature so that we can detect a change in the `Author's Name`. The modified `Author` entity class is shown in the following listing.

Listing 15.5 `Author` entity sending an event when the `Name` property is changed

```
public class Author : EntityEventsBase
{
    private string _name;

    public string Name
    {
```

Adding the EntityEventsBase will provide the methods to send an event.

The backing field for the Name property, which EF Core will read/write

```
        get => _name;
        set
        {
            if (value != _name &&
                AuthorId != default)
                AddEvent(
                    new AuthorNameUpdatedEvent());
            _name = value;
        }
    }

    //… other code left out for clarity
}
```

You make the setting public and override the setter to add the event test/send.

If the Name has changed, and it's not a new Author, sends a domain event

Note that the test of whether the event should be sent includes a test of whether the `Author`'s primary key, `AuthorId`, is set. Because the `Author` entity class doesn't follow the DDD style, you can't be sure how the developer might create a new instance of the `Author` entity, so you add the extra primary-key test to ensure that events are sent only when a `Author's Name` is updated.

15.5.2 Adding code to update the cached values

Now you will create some event handlers to update the cached values when the appropriate domain event comes in. These event handlers will be called before `SaveChanges/SaveChangesAsync`, so the changes that triggered the events and the subsequent changes applied by the event handlers will be saved in the same transaction. I am going to show two styles of updating the cached values within the event handlers:

- The fast delta updates, which work with numeric changes to cached values. When the `AddReview` event is received, for example, the event handler will increment the `ReviewsCount` cache property. This option is fast, but it needs careful coding to make sure that it produces the correct result in every situation.
- The more-normal recalculate updates, in which you run a query to recalculate the cached value. This option is used to update the `AuthorsOrdered` cache property.

UPDATING THE REVIEWS CACHED VALUES USING THE DELTA UPDATE STYLE

Adding, updating, or removing `Reviews` causes specific events, which in turn cause an event handle linked to each event type to run. In this example, you are going to build the event handler code that will update the two cached values, `ReviewsCount` and `ReviewsAverageVotes`, in the `Book` entity. Figure 15.5 shows the stages in the process of adding a new `Review` to a `Book` that already has one `Review`.

The main part of the process is in the event handler, which uses a delta style to update the two `Review` cached values. Listing 15.6 shows the `ReviewAddedHandler` class, which the `GenericEventRunner` library will run before calling `SaveChanges/SaveChangesAsync`.

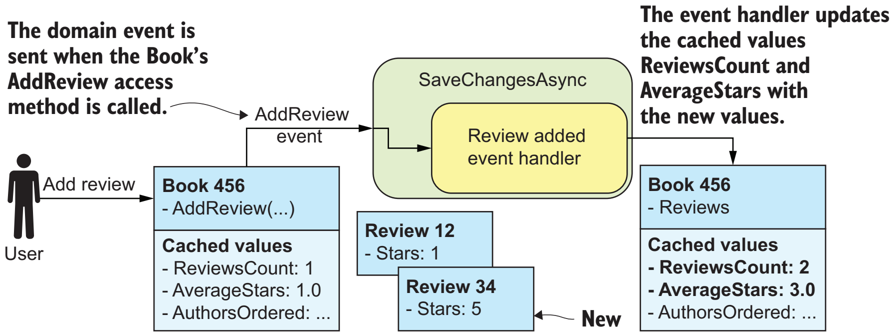


Figure 15.5 When a user adds a new `Review`, the `AddReview` access method creates a domain event, which is picked up by the `GenericEventRunner` when `SaveChanges`/ `SaveChangesAsync` is called. The `GenericEventRunner` runs the `ReviewAddedHandler`, which updates the `Review` cached values using a delta update-style approach.

Listing 15.6 Linking `ReviewAddedHandler` class to the `BookReviewAddedEvent`

```
public class ReviewAddedHandler :
    IBeforeSaveEventHandler<BookReviewAddedEvent>
{
    public IStatusGeneric Handle(object callingEntity,
        BookReviewAddedEvent domainEvent)
    {
        var book = (Domain.Books.Book) callingEntity;

        var totalStars = Math.Round(
            book.ReviewsAverageVotes
                * book.ReviewsCount) +
            domainEvent.NumStars;
        var numReviews = book.ReviewsCount + 1;

        domainEvent.UpdateReviewCachedValues(
            numReviews,
            totalStars / numReviews);

        return null;
    }
}
```

Tells the Event Runner that this event should be called when it finds a BookReviewAddedEvent

The Event Runner provides the instance of the calling entity and the event.

Casts the object back to its actual type of Book to make access easier

The first part of this calculation works out how many stars before adding the new stars.

Adds the star rating from the new Review

A simple add of 1 gets the new number of Reviews.

The entity class provides a method to update the cached values.

The first parameter is the number of reviews.

Returning null is a quick way to say that the event handler is always successful.

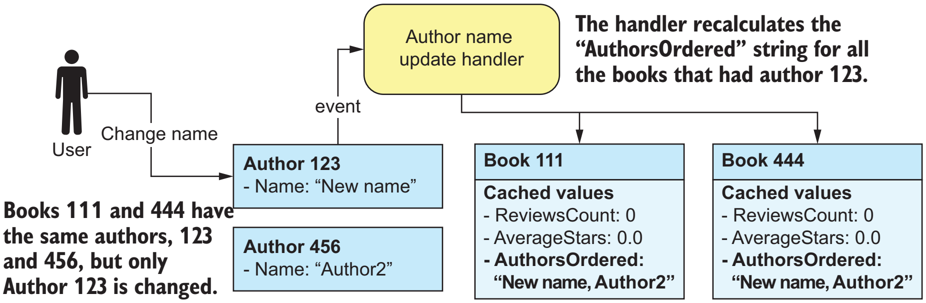
The second parameter provides the new average of the NumStars.

This event handler doesn't access the database and therefore is quick, so the overhead of updating the `ReviewsCount` and `ReviewsAverageVotes` cached values is small.

NOTE The `RemoveReview` event handler isn't shown here but works the same way as the `AddReview` event handler.

UPDATING THE BOOK'S AUTHORS' NAME CACHED VALUE BY RECALCULATION

There are many ways that an `Author` or `Author`'s `Name` could be changed in a `Book`. At `Book` level, someone might have left an `Author` out. At `Author` entity level, someone might have misspelled the author's name (as *John P Smith* instead of *Jon P Smith*, for example). For any of these changes, the affected `Book` entity or entities should update the `Book`'s `AuthorsOrdered` cache value. This string isn't used in a filter or sort, but it saves some time for the display of author names. For this example, you are going to implement the update of the `Author's Name` property, which requires looping through all the `Books` that contain that `Author` entity, as shown in figure 15.6.



The handler recalculates the "AuthorsOrdered" string for all the books that had author 123.

Books 111 and 444 have the same authors, 123 and 456, but only Author 123 is changed.

Figure 15.6 An admin user changes the `Name` of an `Author` that is used in two `Books`. In this example, `Books` 111 and 444 have two `Authors`—123 and 456—linked to them. Changing the 123 `Author's Name` requires the event handler to loop through all the `Books` that the 123 `Author` is used in and recalculate the correct `AuthorsOrdered` string.

The following listing shows the `AuthorNameUpdatedHandler` that the `GenericEventRunner` calls when it finds the domain event that was created when an `Author's Name` property was changed. This event handler loops through all the `Books` that have that `Author` and recalculates each `Book`'s `AuthorsOrdered` cache value.

Listing 15.7 The event handler that manages a change of an `Author's Name` property

Tells the Event Runner that this event should be called when it finds a AuthorNameUpdatedEvent

```
public class AuthorNameUpdatedHandler :
    IBeforeSaveEventHandler<AuthorNameUpdatedEvent>
{
    private readonly BookDbContext _context;

    public AuthorNameUpdatedHandler
        (BookDbContext context)
    {
        _context = context;
    }
```

The event handler needs to access the database.

```
    public IStatusGeneric Handle(object callingEntity,
        AuthorNameUpdatedEvent domainEvent)
    {
        var changedAuthor = (Author) callingEntity;

        foreach (var book in _context.Set<BookAuthor>()
            .Where(x => x.AuthorId == changedAuthor.AuthorId)
            .Select(x => x.Book))
        {
            var allAuthorsInOrder = _context.Books
                .Single(x => x.BookId == book.BookId)
                .AuthorsLink.OrderBy(y => y.Order)
                .Select(y => y.Author).ToList();

            var newAuthorsOrdered =
                string.Join(", ",
                    allAuthorsInOrder.Select(x =>
                x.AuthorId == changedAuthor.AuthorId
                    ? changedAuthor.Name
                    : x.Name));

            book.ResetAuthorsOrdered(newAuthorsOrdered);
        }

        return null;
    }
}
```

The Event Runner provides the instance of the calling entity and the event.

Casts the object back to its actual type of Author to make access easier

Loops through all the books that contain the Author that has changed

Gets the Authors, in the correct order, linked to this Book

Returns the list of author names, but replaces the changed Author's Name with the name provided in the callingEntity parameter

Creates a comma-delimited string with the names from the Authors in the Boo

Updates each Book's AuthorsOrdered property

Returning null is a quick way to say that the event handler is always successful.

As you can see, the `Author's Name` event handler is much more complex and accesses the database multiple times, which is much slower than the `AddReview/RemoveReview` event handler. Therefore, you need to decide whether caching this value will provide an overall performance gain. In this case, the likelihood of updating an `Author's Name` is small, so on balance, it is worthwhile to cache the list of author names for a book.

15.5.3 Adding cache properties to the Book entity with concurrency handling

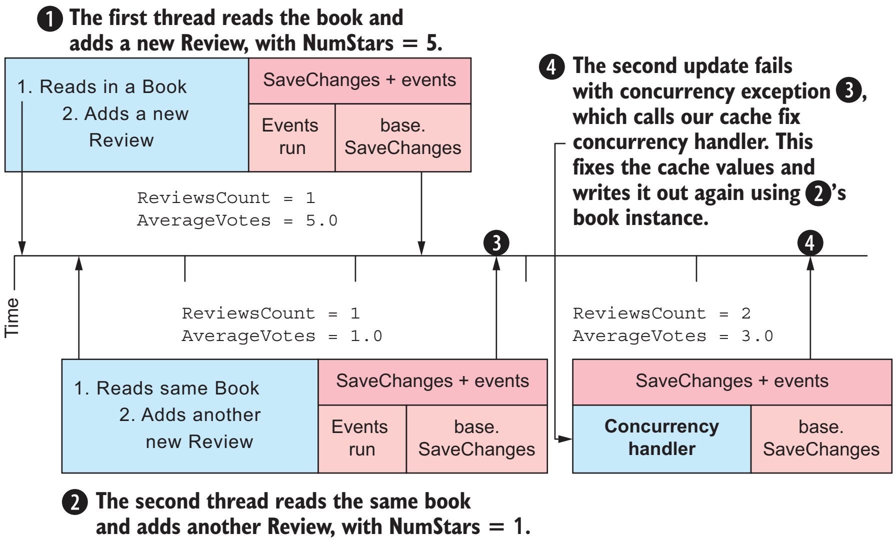
Adding the three cached value properties—`ReviewsCount`, `ReviewsAverageVotes`, and `AuthorsOrdered`—is easy to do. But an issue may occur if two `Reviews` are added simultaneously (or nearly simultaneously) to the same `Book`, which could cause the `Review`-related cached values to be out of date.

Working out the best way to handle simultaneous updates took the most time to think through and design. I spent days thinking about all the concurrency issues that could cause a problem and then even more days coming up with the best way to handle those issues. This part of the caching design is the most complex and needs careful thought.

First, I considered updating the cache values inside a transaction, but the isolation level needed for totally accurate cache updating required locking a lot of data. Even using direct SQL commands to calculate and update the cache wasn't safe. (See the fascinating Stack Overflow question/answer "Is a single SQL Server statement atomic and consistent?" at https://stackoverflow.com/q/21468742/1434764.)

I found that the best way to handle the simultaneous-updates problem was to configure the three cache values as concurrency tokens (see section 10.6.2). Two simultaneous updates of a cache value will throw a `DbUpdateConcurrencyException`, which then calls a concurrency handler written to correct the cache values to the right values.

Figure 15.7 shows what happens if two `Reviews` are added simultaneously, which causes a `DbUpdateConcurrencyException` to be thrown. Then the concurrency handler comes in to fix the `ReviewsCount` and `ReviewsAverageVotes` cache values.



① The first thread reads the book and adds a new Review, with NumStars = 5.

④ The second update fails with concurrency exception ③, which calls our cache fix concurrency handler. This fixes the cache values and writes it out again using ②'s book instance.

② The second thread reads the same book and adds another Review, with NumStars = 1.

Figure 15.7 This figure shows how two simultaneous updates could cause an incorrect cached value, which is detected by making the `ReviewsCount` and `ReviewsAverageVotes` cache properties configured as concurrency tokens. This example would throw a `DbUpdateConcurrencyException`, which would be caught and directed to the concurrency handler. The concurrency handler is designed to handle this type of concurrency issue and correct the cache values.

This section shows the following parts of the concurrency handler:

- Code to capture any exception thrown by `SaveChanges`/`SaveChangesAsync`
- The top-level concurrency handler that finds the `Book(s)` that caused the `DbUpdateConcurrencyException`
- The concurrency handler for a problem with the `Review`'s cached values
- The concurrency handler for a problem with the `AuthorsString` cached value

Code to capture any exception thrown by SaveChanges/SaveChangesAsync

To capture `DbUpdateConcurrencyException`, you need to add a C# `try/catch` around the call to the `SaveChanges/SaveChangesAsync` methods. This addition allows you to call an exception handler to try to fix the problem that caused the exception or rethrow the exception if it can't fix the problem. If your exception handler managed to fix the exception, you call `SaveChanges/SaveChangesAsync` again to update the database with your fix.

In this specific case, you need to consider another issue: while you were fixing the first concurrency update, another concurrency update could have happened. Sure, this scenario is rather unlikely, but you must handle it; otherwise, the second call to `SaveChanges/SaveChangesAsync` would fail. For this reason, you need a C# `do/while` outer loop to keep retrying the call to the `SaveChanges/SaveChangesAsync` method until it is successful or an exception that can't be fixed occurs.

Also, the `GenericEventRunner` library allows you to register an exception handler to be called if the `SaveChanges/SaveChangesAsync` method throws an exception. Your exception handler must return an `IStatusGeneric`, and there are three possible options:

- *Status has no errors.* Your exception handler has fixed the problem, and the `SaveChanges/SaveChangesAsync` method should be called again to update the database
- *Status returns errors.* The exception handler has converted the exception to error message(s). This approach is useful for turning database exceptions into user-friendly error messages.
- *Status returns* `null`. The exception handler can't handle the exception, and the exception should be rethrown

The following listing shows the code inside the `GenericEventRunner` library that calls `SaveChanges`, showing the outer `do/while` and the inner `try/catch` of the exception.

Listing 15.8 A simplified version of the `GenericEventRunner`'s `SaveChanges` call

```
private IStatusGeneric<int>
    CallSaveChangesWithExceptionHandler
    (DbContext context,
        Func<int> callBaseSaveChanges)
{
    var status = new StatusGenericHandler<int>();

    do
    {
        try
        {
            int numUpdated = callBaseSaveChanges();
            status.SetResult(numUpdated);
            break;
        }
```

The returned value is a status, with int returned from SaveChanges.

The base SaveChanges is provided to be called.

The status that will be returned

The call to the SaveChanges is done within a do/while.

Calls the base SaveChanges

If no exception occurs, sets the status result and breaks out of the do/while

The catch caches any exceptions that SaveChanges throws.

```
        catch (Exception e)
        {
            IStatusGeneric handlerStatus
                 = … YOUR EXCEPTION HANDLER GOES HERE;
            if (handlerStatus == null)
                throw;
            status.CombineStatuses(handlerStatus);
        }

    } while (status.IsValid);

    return status;
}
```

Your exception handler is called here, and it returns null or a status.

If the exception handler returns null, it rethrows the original exception…

…otherwise, any errors from your exception handler are added to the main status.

If the exception handler was successful, it loops back to try calling SaveChanges again.

Returns the status

TOP-LEVEL CONCURRENCY HANDLER THAT FINDS THE BOOK(S) THAT CAUSED THE EXCEPTION

Handling a concurrency issue involves several common parts, so you build a top-level concurrency handler to manage those parts. The following listing shows the top-level concurrency handler method `HandleCacheValuesConcurrency`.

Listing 15.9 The top-level concurrency handler containing the common exception code

If the exception isn't a Db.UpdateConcurrencyException, we return null to say that we can't handle that exception

This extension method handles the Reviews and Author cached values concurrency issues.

```
public static IStatusGeneric HandleCacheValuesConcurrency
    (this Exception ex, DbContext context)
{
    var dbUpdateEx = ex as DbUpdateConcurrencyException;
    if (dbUpdateEx == null)
        return null;
```

Casts the exception to a DbUpdateConcurrency-Exception

Should be only one entity, but we handle many entities in case of bulk loading

```
    foreach (var entry in dbUpdateEx.Entries)
    {
        if (!(entry.Entity is Book bookBeingWrittenOut))
            return null;
```

Casts the entity to a Book. If it isn't a Book, we return null to say the method can't handle it.

```
        var bookThatCausedConcurrency = context.Set<Book>()
            .IgnoreQueryFilters()
            .AsNoTracking()
            .SingleOrDefault(p => p.BookId
                == bookBeingWrittenOut.BookId);
```

Reads a nontracked version of the Book from the database. (Note the Ignore-QueryFilters, because it might have been soft-deleted.)

If no book was deleted, marks the current book as detached so it won't be updated

```
        if (bookThatCausedConcurrency == null)
        {
            entry.State = EntityState.Detached;
            continue;
        }

        var handler = new FixConcurrencyMethods(entry, context);
```

Creates the class containing the Reviews and AuthorsOrdered cached values

```
        handler.CheckFixReviewCacheValues(
            bookThatCausedConcurrency, bookBeingWrittenOut);

        handler.CheckFixAuthorOrdered(
            bookThatCausedConcurrency, bookBeingWrittenOut);
    }

    return new StatusGenericHandler();
}
```

Fixes any concurrency issues with the Reviews cached values

Fixes any concurrency issues with the AuthorsOrdered cached value

Returns a valid status to say that the concurrency issue was fixed

CONCURRENCY HANDLER FOR A PROBLEM WITH THE REVIEW'S CACHED VALUES

The `CheckFixReviewCacheValues` concurrency handler method deals only with the Review cached values. Its job is to combine the `Review` cached values in the entity that is being written out and the `Review` cached values that have been added to the database. This method uses the same delta update style used in the `Review` cached values event handler. The following listing shows the `CheckFixReviewCacheValues` concurrency handler.

> **NOTE** If you aren't familiar with EF Core concurrency handling, I recommend that you look at section 10.6.3, which describes the different types of data that are involved in handling a concurrency exception.

Listing 15.10 The code to fix a concurrent update of the `Review` cached values

```
public void CheckFixReviewCacheValues(
    Book bookThatCausedConcurrency,
    Book bookBeingWrittenOut)
{
    var previousCount = (int)_entry
        .Property(nameof(Book.ReviewsCount))
        .OriginalValue;
    var previousAverageVotes = (double)_entry
        .Property(nameof(Book.ReviewsAverageVotes))
        .OriginalValue;

    if (previousCount ==
        bookThatCausedConcurrency.ReviewsCount
        && previousAverageVotes ==
           bookThatCausedConcurrency.ReviewsAverageVotes)
        return;

    var previousTotalStars = Math.Round(
        previousAverageVotes * previousCount);

    var countChange =
        bookBeingWrittenOut.ReviewsCount
        - previousCount;
    var starsChange = Math.Round(
          bookBeingWrittenOut.ReviewsAverageVotes
          * bookBeingWrittenOut.ReviewsCount)
      - previousTotalStars;
```

This method handles concurrency errors in the Reviews cached values.

This parameter is the Book from the database that caused the concurrency issue.

This parameter is the Book you were trying to update.

Holds the count and votes in the database before the events changed them

If the previous count and votes match the current database, there is no Review concurrency issue, so the method returns.

Works out the stars before the new update is applied

Gets the change that the event was trying to make to the cached values

```
    var newCount =
        bookThatCausedConcurrency.ReviewsCount
        + countChange;
    var newTotalStars = Math.Round(
            bookThatCausedConcurrency.ReviewsAverageVotes
            * bookThatCausedConcurrency.ReviewsCount)
        + starsChange;
```

Works out the combined change from the current book and the other updates done to the database

```
    _entry.Property(nameof(Book.ReviewsCount))
        .CurrentValue = newCount;
    _entry.Property(nameof(Book.ReviewsAverageVotes))
        .CurrentValue = newCount == 0
        ? 0 : newTotalStars / newCount;
```

Sets the Reviews cached values with the recalculated values

```
    _entry.Property(nameof(Book.ReviewsCount))
        .OriginalValue = bookThatCausedConcurrency
        .ReviewsCount;
    _entry.Property(nameof(Book.ReviewsAverageVotes))
            .OriginalValue =
        bookThatCausedConcurrency
            .ReviewsAverageVotes;
}
```

Sets the OriginalValues for the Review cached values to the current database

Yes, this code is quite complicated, which is why I give the variables good names. Even I can get lost in this code if I come back to it months later.

CONCURRENCY HANDLER FOR A PROBLEM WITH THE AUTHORSSTRING CACHED VALUE

The `CheckFixAuthorsOrdered` concurrency handler method has the same format as the `CheckFixReviewCacheValues` method, but it deals with the `AuthorsOrdered` cached value. Its job is to combine the `AuthorsOrdered` cached value in the entity that is being written out and the `AuthorsOrdered` cached value that has been added to the database. As a result, the `CheckFixAuthorsOrdered` concurrency handler, shown in the next listing, must use the recalculate update style, because you can't use the delta update approach.

Listing 15.11 The code to fix a concurrent update of the `AuthorsOrdered` cached value

This method handles concurrency errors in the AuthorsOrdered cached value.

This parameter is the Book from the database that caused the concurrency issue.

This parameter is the Book you were trying to update.

```
public void CheckFixAuthorsOrdered(
    Book bookThatCausedConcurrency,
    Book bookBeingWrittenOut)
{
    var previousAuthorsOrdered = (string)_entry
        .Property(nameof(Book.AuthorsOrdered))
        .OriginalValue;
```

Gets the previous AuthorsOrdered string before the event updated it

```
    if (previousAuthorsOrdered ==
        bookThatCausedConcurrency.AuthorsOrdered)
        return;
```

If the previous AuthorsOrdered match the current database AuthorsOrdered, there is no AuthorsOrdered concurrency issue, so the method returns.

```
    var allAuthorsIdsInOrder = _context.Set<Book>()
        .IgnoreQueryFilters()
        .Where(x => x.BookId ==
                    bookBeingWrittenOut.BookId)
        .Select(x => x.AuthorsLink
            .OrderBy(y => y.Order)
            .Select(y => y.AuthorId)).ToList()
        .Single();

    var namesInOrder = allAuthorsIdsInOrder
        .Select(x => _context.Find<Author>(x).Name);

    var newAuthorsOrdered =
        string.Join(", ", namesInOrder);

    _entry.Property(nameof(Book.AuthorsOrdered))
        .CurrentValue = newAuthorsOrdered;

    _entry.Property(nameof(Book.AuthorsOrdered))
        .OriginalValue =
        bookThatCausedConcurrency.AuthorsOrdered;
}
```

Gets the AuthorIds for each Author linked to this Book in the correct order

Gets the Name of each Author, using the Find method

Creates a comma-delimited list of authors

From this, you can set the AuthorsOrdered cached value with the combined values.

Sets the OriginalValues for the AuthorsOrdered cached value to the current database

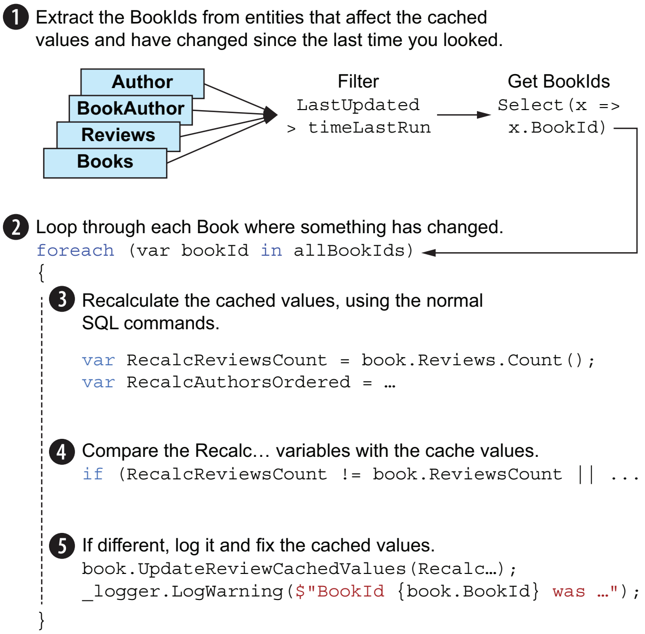
The important part to point out is that you must read in the `Author` entity classes by using the `Find` method because the `Author` that created the update to the `AuthorsOrdered` cached value hasn't yet been written to the database. `Find` is the only query method that will first inspect the current application's DbContext for tracked entities to find the entity you want. The `Find` will load the tracked entity with that `AuthorId` instead of loading the version in the database that hasn't been updated yet.

15.5.4 Adding a checking/healing system to your event system

Since the first edition, I have performance-tuned several client systems and created a caching system that covers all eventualities I can think of. But I may have missed something, so I added a separate checking/healing system to run alongside my caching system to tell me if there is a problem. That system lets me sleep at night, and my clients like the fact that they can be certain their data is up to date.

You may think this approach is overkill, but if you are adding a caching system to an existing system, you need some way to fill in the cached values of existing data anyway. Typically, I build some code to add the cached values to the current application's production database before releasing a new version of the application that uses the cached values in a query. It takes only a bit more effort to make that update-cache code into a useful service that can be used to check and fix cached values.

As an example, I have built a checking/healing system into the Book App. This service, called `CheckFixCacheValuesService`, is available in the ASP.NET Core app. This service can be used in checking/healing as required. Rather than detailing the code, I provided figure 15.8, which shows an overview of what the `CheckFixCacheValuesService` class does.



❶ Extract the BookIds from entities that affect the cached values and have changed since the last time you looked.

```
LastUpdated
> timeLastRun
```

```
Select(x =>
 x.BookId)
```

❷ Loop through each Book where something has changed.

```
foreach (var bookId in allBookIds)
{
```

❸ Recalculate the cached values, using the normal SQL commands.

```
    var RecalcReviewsCount = book.Reviews.Count();
    var RecalcAuthorsOrdered = …
```

❹ Compare the Recalc… variables with the cache values.

```
    if (RecalcReviewsCount != book.ReviewsCount || ...
```

❺ If different, log it and fix the cached values.

```
    book.UpdateReviewCachedValues(Recalc…);
    _logger.LogWarning($"BookId {book.BookId} was …");
}
```

Figure 15.8 The five stages of the `CheckFix` service in the `Part3` Book App. This code is run from a background service, which periodically checks the database for entities that have changed and could potentially change the cached values. Because this code uses a different way to find and calculate the cached values, it will find any cached values that are out of date and correct them for you.

> **NOTE** The `CheckFixCacheValuesService` class and its related classes are in the GitHub repo associated with this book, inside the folder called CheckFixCode in the project called BookApp.Infrastructure.Books.EventHandlers. You can also find a background service in the project called BookApp.BackgroundTasks.

The downside of the checking/healing code shown in figure 15.8 is that it adds more database accesses, which could affect the performance of your system. In the Book App, for example, an update to an entity class causes a `LastUpdatedUtc` property to be updated (see section 11.4.3). The checking/healing code can find all the entities that were changed in, say, the past 24 hours quite quickly (the test database has 700,000 entities and takes only about 10 ms to scan), but each check of a changed entity takes 5 ms. So if your application has lots of changes per day, the checking/healing code is going to take some time.

For that reason, this sort of checking/healing system is run at a time when there aren't many users on the system—at night or on the weekend, or manually by an admin person when they suspect a problem. The system isn't likely to find anything, but if it does find a bad cache value setting, you know that there is a bug in your cache code.

The `Part3` Book App has an example of the overnight and manual triggering of its checking/healing system. An ASP.NET Core background service runs `CheckFixCacheValuesService` at 1:00 every morning (was GMT time zone, but failed on Linux, so now uses UTC), and you can run the checking/healing service manually by choosing the Admin > Check Cached Vals menu item.

> **WARNING** The design of the Book App's `CheckFixCacheValuesService` service assumes that no database updates are happening when it is fixing incorrect cache values. If concurrency exceptions arise, the `CheckFixCacheValuesService` service would need its own concurrency exception handler.

15.6 Comparing the four performance approaches with development effort

At the start of this chapter, I compared the performance improvements of the four approaches. Although the improvements in performance are undeniable, there are other factors to consider when considering each performance-tuning approach, such as how much development effort each approach took, whether any of them needed specific skills, and how complex the solutions were.

In this section, I look at these considerations and provide some extra information to try to answer some development questions. To start, figure 15.9 provides a quick summary of the four ways to improve your application in terms of performance, skills, and development time.

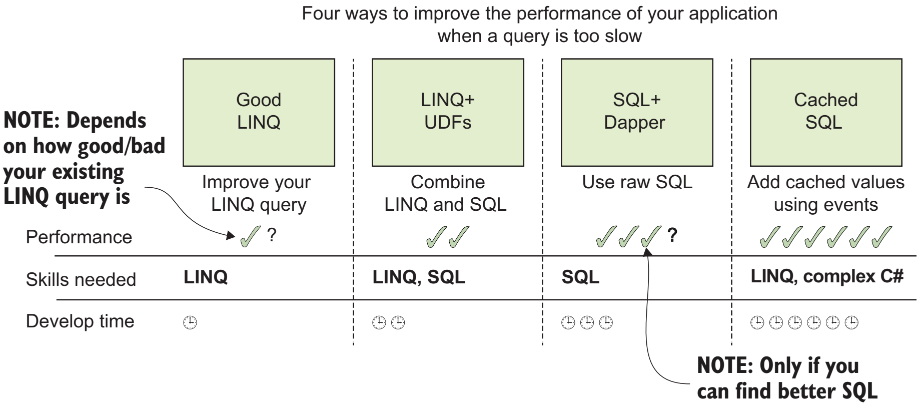


Figure 15.9 Four approaches to improving the performance of a query. Each approach is scored for performance improvement (more check marks mean better performance), the skills you need to apply that approach, and the amount of development time needed to implement the code associated with the approach.

Table 15.2 provides a textual summary of the four approaches in terms of effort and skills required.

Table 15.2 The amount of effort needed to apply the four approaches to the Book App

Approach	Effort+skills	Comments
Good LINQ	Time: Low (built in chapter 2) Skills: LINQ, DDD	The `Select` query is the same one I used in chapter 2, and it works well. The key part was working out how to average the `Review`'s `NumStars` properties inside the database (see section 6.1.8). Also, the change to a DDD-styled entity class meant that the price was available as a single property that could have an SQL index added to it.
LINQ+UDFs	Half a day LINQ + SQL	I have found that UDFs (see section 10.1) are good ways to keep a LINQ approach but replace part of a LINQ query that isn't working as well as I would like. But UDFs are useful only if you can find some better SQL to put into a UDF.
SQL+Dapper	Half-day of study, half-day to write SQL	This approach required studying the SQL generated by EF Core and working out whether I could do anything to improve it. I found only one thing to improve (sort on votes), but that feature is a key one. Rewriting the SQL to have all the filters, sorts, and paging was a bit tedious—much harder than using LINQ.
LINQ+caching	LINQ+caching: about a week, but quicker next time. Check/heal: 1.5 days Complex C#, concurrency	This approach is definitely hard work but also provides a fantastic result. It took a lot of time to work out the best way to handle concurrent updates and testing, but having implemented this approach once, I'd be quicker next time. The checking/healing code took a bit more time, but as I said, I'd normally have to write it anyway if I was performance-tuning an existing application that already had user data. Another source of time taken to implement a caching system was work I did for a client. I took 11 hours to build a single delta cached values system, but I didn't need to do the concurrency handling, as the client's app stopped all duplicate user updates of data.

Overall, I'm pleased with the process. EF Core produces great SQL code from the get-go, but only because you made sure that your LINQ queries were written in a way that is sympathetic to how EF Core works. The methods and approaches introduced in the first six chapters are a good starting point for writing good LINQ queries.

As I said in chapter 14, make sure that your standard patterns for queries work well; otherwise, you'll be building inefficiencies into your application from the start. But at some point, you'll need to performance-tune an EF Core application; this chapter provides lots of ideas and approaches that can help.

15.7 Improving database scalability

The four performance-tuning approaches are all about speed: how fast you can return the result to the user. But the other aspect to consider is *scalability*: handling large numbers of concurrent users. To end this chapter, let's look at database scalability.

Section 14.7 talks about database scalability in terms of the ability to buy more-powerful hardware to run your database server on, because this book is about EF Core. But the overall scalability of the application is what matters most. For that reason, I always show the performance of the whole application, as that's what the end user is going to see. Focusing on overall application performance stops you from spending a lot of time shaving a few milliseconds off database access timings when the frontend code is taking more than 100 ms to display the data.

The first thing you should do to improve scalability is use async database accesses. Async commands used in an ASP.NET Core application will release a thread that can be used by another user, thus saving the ASP.NET Core thread pool from being used up (see section 5.10.1). Async commands have a small downside—they take a bit longer to run (see section 14.7.2 for detailed timings)—but overall, async is the way to go in any application that has lots of simultaneous users. The `Part3` Book App uses async commands throughout.

The other helpful whole-application changes you can make with applications such as ASP.NET Core are running more-powerful instances of the application (known as *scaling up*) and running more instances of the application (known as *scaling out*). You might like to pay for more-powerful hardware to run your database server on, too.

> **NOTE** All the approaches used in this chapter will work on an application using multiple instances of ASP.NET Core, including the LINQ+caching approach. The overnight check/heal service, however, would need to be run on a single WebJob instead of as a ASP.NET Core background service.

One basic fact about database scalability is that the quicker you make the database accesses, the more concurrent accesses the database can handle. Reducing the number of round trips to the database also reduces the load on the database (see section 14.5.1). Fortunately, since EF Core 3, the default query type has loaded any collections within one database access. Also, lazy loading might feel like a great time-saver, but it adds all those individual database accesses back in, and both scalability and performance suffer.

But some large applications will have high concurrent database accesses, and you need a way out of this situation. The first, and easiest, approach is to pay for a more powerful database. If that solution isn't going to cut it, here are some ideas to consider:

- *Split your data over multiple databases: Sharding your data*

  If your data is segregated in some way (if you have a financial application that many small businesses use, for example), you could spread each business's data over a different database—that is, one database for each business. This approach is called sharding (see http://mng.bz/veN4). Section 11.7 shows a simple way to implement sharing by using EF Core.
- *Split your database reads from your writes: CQRS architecture*

  Command and Query Responsibility Segregation (CQRS) architecture (see https://martinfowler.com/bliki/CQRS.html) splits the database reads from the database writes. This approach allows you to optimize your reads and possibly use a separate database, or multiple read-only databases, on the CQRS read side.

- *Mix NoSQL and SQL databases: Polyglot persistence*

  The cached SQL approach makes the `Book` entity look like a complete definition of a book that a JSON structure would hold. With a CQRS architecture, you could have used a relational database to handle any writes, but on any write, you could build a JSON version of the book and write it to a read-side NoSQL database or multiple databases. This approach, which might provide higher read performance, is one form of polyglot persistence (see http://mng.bz/K4RX). In section 16.3, you'll implement a mixed SQL/NoSQL application to gain even more performance, especially in terms of scalability.

Summary

- If you build your LINQ queries carefully and take advantage of all its features, EF Core will reward you by producing excellent SQL code.
- You can use EF Core's `DbFunction` feature to inject a piece of SQL code held in an SQL UDF into a LINQ query. This feature allows you to tweak part of an EF Core query that's run on the database server.
- If a database query is slow, check the SQL code that EF Core is producing. You can obtain the SQL code by looking at the `Information` logged messages that EF Core produces.
- If you feel that you can produce better SQL for a query than EF Core is producing, you can use several methods to call SQL from EF Core, or use Dapper to execute your SQL query directly.
- If all other performance-tuning approaches don't provide the performance you need, consider altering the database structure, including adding properties to hold cached values. But be warned: you need to be careful.
- In addition to improving the time that a query takes, consider the scalability of your application—that is, supporting lots of simultaneous users. In many applications, such as ASP.NET Core, using async EF Core commands can improve scalability. Chapter 16 provides another way to improve scalability and performance by adding a Cosmos DB database to the Book App.

For readers who are familiar with EF6:

- EF6.x doesn't have EF Core's `DbFunction` feature, which makes calling a UDF so easy in EF Core.

16

Cosmos DB, CQRS, and other database types

This chapter covers

- Introducing NoSQL databases and how they differ from relational databases
- Exploring the features that the NoSQL database called Cosmos DB
- Performance-tuning the Book App using EF Core Cosmos DB database provider
- Considering the differences between and limitations of using Cosmos DB with EF Core 5
- Knowing what issues you might hit when swapping from one database type to another

The Book App has been a constant theme throughout this book, and up until now, it has used an SQL Server database to store the books data. In this chapter, we are going to performance-tune the Book App by combining the original SQL Server database with another database called Cosmos DB. In chapter 14, we performance-tuned the Book App to handle 100,000 books. In this chapter, we take the number of books to 500,000 with the same or better performance by using Cosmos DB. Cosmos DB is relatively new (it came out in 2017), and some readers won't have used it yet. So in addition to using this database to improve performance and scalability, I point out the differences between Cosmos DB, which is a NoSQL database, and the more traditional relational databases such as SQL Server.

Cosmos DB and relational databases differ a lot, but there are also some small changes between various relational databases that EF Core supports. So at the end of the chapter, there is a list of things to check and change if you're swapping from one relational database type to another.

16.1 The differences between relational and NoSQL databases

TIME-SAVER Skip this section if you already know about NoSQL databases.

Cosmos DB isn't like the databases described so far in the book, such as SQL Server, PostgreSQL, and SQLite. Cosmos DB is what is referred to as a NoSQL database, whereas SQL Server, PostgreSQL, and SQLite (along with many others) are referred to as relational databases.

As you have already read, relational databases use primary keys and foreign keys to form links between tables, which EF Core turns into navigational properties. Relational databases excel at relationships with lots of database rules (called constraints) to make sure that these relationships follow the design you decided on for your database, which is why they are called relational databases.

Relational databases have been around for decades, and nearly all of them use the SQL language, which means that each implementation of a relational database is similar to every other. So swapping from, say, SQL Server to PostgreSQL isn't too hard, especially if you are using EF Core, which hides some of the differences. The long life of relational databases also means that you can find many relational implementations, lots of tools, and expertise on relational databases.

On the other hand, NoSQL databases are designed to be high-performance in terms of speed, scalability, and *availability* (the ability to swap to another database if one fails). There is no common language, such as SQL, so each implementation goes its own way to maximize the features it wants to focus on. To achieve these performance goals, the NoSQL databases give up some of the rules that the relational databases apply.

Many NoSQL databases allow multiple instances of the same database to provide scalability and availability, for example. To do so, NoSQL databases drop the relational rule that the data is always consistent—that is, you will always get the latest data. NoSQL databases are *eventually consistent*, which means that an update to one database instance may take some time (ideally, seconds or less) to be applied to another database instance.

MORE INFORMATION If you want to look into the types of and differences between relational and NoSQL databases, I recommend the Microsoft article "Relational vs. NoSQL data" at http://mng.bz/9Nzj.

16.2 Introduction to Cosmos DB and its EF Core provider

As I've already said, Cosmos DB doesn't follow the way that relational databases work. Sure, it has a database, and it even has some pseudo-SQL commands, but otherwise, it's quite different from relational databases. EF Core's support of Cosmos DB, however, provides a common frontend that makes it easier for someone who already knows EF Core to use Cosmos DB.

In this chapter, you are going to look at features of both Cosmos DB itself and EF Core's current Cosmos DB database provider. You should note that I say *current* Cosmos DB database provider because the EF Core 5 Cosmos DB database provider is far from finished, as I cover in detail in this chapter.

To understand why EF Core's Cosmos DB database provider hasn't been improved, you need only look at this statistic: the number of Cosmos DB downloads is only 1% of all SQL Server downloads. The EF Core team is driven by what the developers need, and being a small team, it can't do everything. Therefore, the Cosmos DB database provider hasn't been improved in EF Core 5. But as you will see, I successfully used the EF Core 5 Cosmos DB database provider to improve the Book App's performance.

So why am I dedicating this chapter to Cosmos DB if EF Core's database provider has limitations, and why should you read it? Fundamentally, for some applications, using a NoSQL database is going to provide better performance and scalability than a similarly priced relational database. Also, the plan for EF Core 6 (see http://mng.bz/Wreg) has a section on improving EF Core's support of Cosmos DB, so I am hopeful that some (if not many) of the limitations in this chapter will be removed.

Because EF Core's current Cosmos DB database provider is likely to improve, I am careful to separate the differences between the Cosmos DB and a relational database and the limitations of the EF Core Cosmos DB database provider. This convention ensures that this chapter will still be useful when improved versions of the Cosmos DB database provider are released.

> **NOTE** This SQL/NoSQL comparison *doesn't* say that one is better than the other; each has its own strengths and weaknesses. Also, Cosmos DB is one implementation of a NoSQL database, so its limitations are going to be different from other NoSQL implementations. The comparison is here to point out the parts of the Cosmos DB that work differently from the relational databases that have been around for years.

The other reason for looking at the differences between a Cosmos DB database and relational databases is to give you some pointers about when you could use Cosmos DB instead of an SQL database. Section 16.6.1 covers many of the differences between Cosmos DB and relational databases, with a few other differences identified by notes starting with COSMOS DB DIFFERENCE; see the following example.

> **COSMOS DB DIFFERENCE** This feature of the Cosmos DB database doesn't work the same way as relational databases.

The other area I want to highlight is the limitations of the EF Core 5 Cosmos DB database provider. These are areas where EF Core 5 doesn't implement code to take advantage of all the features of Cosmos DB (but be aware that future releases of EF Core may well remove some of these limitations). Section 16.6.3 covers many of the limitations of the EF Core 5 Cosmos DB database provider, with a few other limitations identified by notes starting with EF CORE 5 LIMITATION; see the following example.

> **EF CORE 5 LIMITATION** This limitation applies to the current EF Core 5 Cosmos DB database provider.

16.3 Building a Command and Query Responsibility Segregation (CQRS) system using Cosmos DB

To get a good feel for Cosmos DB, we need to build something real; that's the way I learn. I suggested in section 15.7 that a CQRS architecture could provide better scalability performance. Adding a CQRS system that uses Cosmos DB isn't trivial, so this example will reveal many differences between the NoSQL Cosmos DB and relational databases. I hope that it will also provide another technique you can use to performance-tune your own applications.

In this section, you are going to build a CQRS architecture by using a polyglot database structure that will provide better performance and scalability.

> **DEFINITION** A CQRS architecture segregates query operations from operations that update data by using separate interfaces. This architecture can maximize performance, scalability, and security, and supports the evolution of the system over time through higher flexibility. See http://mng.bz/Ix8D.

> **DEFINITION** A polyglot database structure uses a combination of storage types: relational databases, NoSQL databases, flat files, and so on. The idea is that each database type has strengths and weaknesses, and by using two or more, you can obtain a better overall system. See http://mng.bz/6r1W.

The CQRS architecture acknowledges that the read side of an application is different from the write side. Reads are often complicated, drawing in data from multiple places, whereas in many applications (but not all), the write side can be simpler and less onerous. You can see in the current Book App that listing the books is complex but adding a review is fairly trivial. Separating the code for each part can help you focus on the specific features of each part—another application of the SoC software principle.

In chapter 15, you produced the performance version, in which you cached values (see section 15.5). It struck me then that the final query didn't access any relationships and could be stored in a simpler database, such as a NoSQL database. In this example, you'll use a polyglot database structure, with a mixture of SQL and NoSQL databases, for the following reasons:

- Using an SQL write-side database makes sense because business applications often use relational data. Think about a real book-selling site, which would have a lot of complex, linked data to handle business aspects such as suppliers, inventory, pricing, orders, payment, delivery, tracking, and audits. I think that a well-known relational/SQL database, with its superior level of data integrity, would be a good choice for many business problems.

- But those relationships and some aspects of an SQL database, such as the need to dynamically calculate some values, can make it slow in retrieving data. So a NoSQL database with precalculated values such as average review votes can improve performance considerably over an SQL database. The CQRS read-side projection is what Mateusz Stasch calls "a legitimate cache" in his article at http://mng.bz/A7eC.

As a result of these design inputs, you'll develop what I refer to as a two-database CQRS architecture, as shown in figure 16.1.

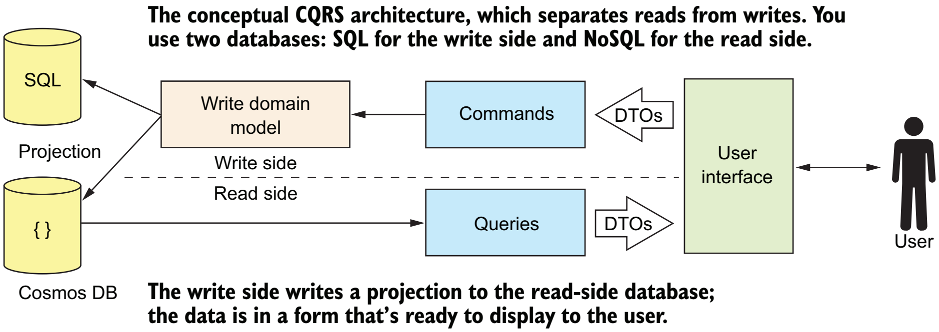


Figure 16.1 A conceptual view of a CQRS architecture: an SQL database for the write side and a NoSQL database for the read side. A write takes a bit more work because it writes to two databases: the normal SQL database and the new NoSQL read-side database. In this arrangement, the read-side database is writing in the exact format needed by the user, so reads are fast.

Because the CQRS architecture separates read and write operations, using one database for read operations and another for write operations is a logical step. The write side holds the data in a relational form, with no duplication of data—a process known as *normalization*—and the read side holds the data in a form that is appropriate for the user interface.

In the Book App, the read side would contain the data already converted to match what the book display needs; these prebuild entities are known as *projections*. These projections are built with the same code as the `MapBookToDto` method in section 2.6. What you are doing is prebuilding the views you need and writing them to the read-side database.

This design creates good performance gains for reads but a performance cost on writes, making the two-database CQRS architecture appropriate when your business application has more reads of the data than writes. Many business applications have more reads than writes (e-commerce applications are good examples), so this architecture fits our Book App well.

16.4 The design of a two-database CQRS architecture application

The fundamental issue in building any CQRS system is making sure that any changes to the data change the associated projection in the read-side CQRS database. If you get that part wrong, you will show the wrong data to the user. This issue is the same cache invalidation issue I worked so hard to get right in the cached SQL approach described in section 15.5. The trick is to capture every change to the SQL `Book` entity and its associated entities and to make sure that the read-side CQRS database is updated.

In the first edition of this book, I detected changes to `Book` and associated entities by looking at the `State` of tracked entities within the call to the `SaveChanges/SaveChangesAsync` methods. These `States` and entities were decoded to define whether a projection to the NoSQL database should be added, updated, or deleted. That approach is valid (I show an example in section 12.5), but using the `State` of multiple entities can be quite complex.

Another approach is using integration events (section 12.1.2) triggered by the DDD access methods (see section 13.4.2). Here are some benefits of this approach:

- *More robust*—Using integration events ensures that the SQL database is updated only when the Cosmos DB database has successfully updated its database. Applying both database updates within a transaction reduces the possibility that the Cosmos DB database will get out of step with the SQL write side. (The design in the first edition of this book could get out of step if a RavenDb update failed.)
- *More obvious*—You trigger integration events inside the DDD methods that change the data. Each event tells the event handler whether it's an `Add`, `Update`, or `Delete` (soft delete, in this case) of a `Book`. Then it's easy to write the event handler to `Add`, `Update`, or `Delete` a `Book` projection in the Cosmos DB.
- *Simpler*—As already stated, sending integration events is much simpler than making detected changes via the `State` of the tracked entities. (See section 12.5 for a description of that approach.)

Figure 16.2 shows what happens when an admin person adds a new `Book` and how that new `Book` gets added to the Cosmos DB database so that the user can see it.

To implement the CQRS system shown in figure 16.2, you must take the following steps:

1. Create an event to trigger when the SQL Book entity changes.
2. Add events to the `Book` entity to send `Add`, `Update`, or `Delete` integration events.
3. Use the `EfCore.GenericEventRunner` to override your `BookDbContext`.
4. Create the Cosmos entity classes and DbContext.
5. Create the Cosmos `Add`, `Update`, and `Delete` event handlers.

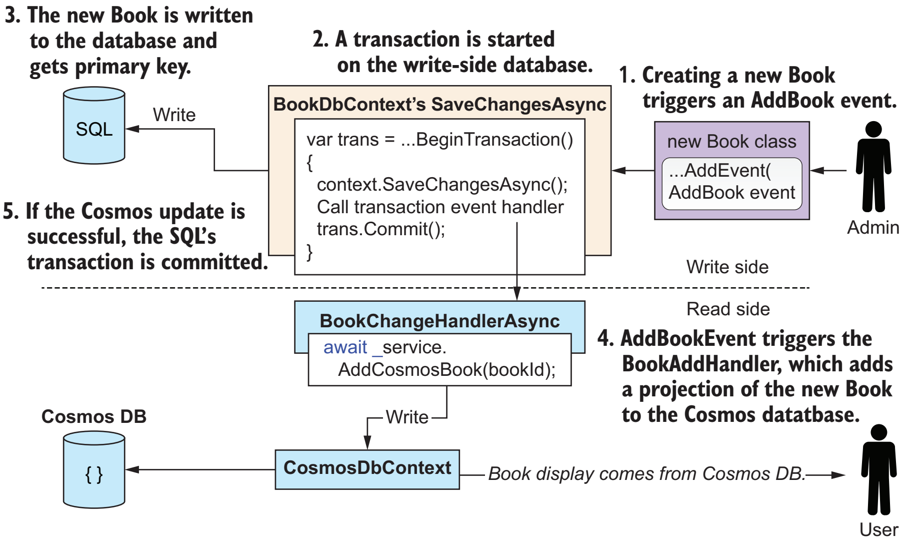


Figure 16.2 **Adding a new `Book` entity. The `Book`'s static factory adds an `Add Book` integration event; this event is picked up by the `BookDbContext`, which handles access to the SQL database. `SaveChanges/SaveChangesAsync` has been overridden by the `EfCore.GenericEventRunner`. Because the event is an integration event, the library starts a transaction and writes out the new `Book`, which obtains the SQL primary key. Then the `Add Book` integration event calls the `BookChange` event handler, which creates a projection of the new `Book` and adds it to the Cosmos DB database. If the write to the Cosmos DB database is successful, the transaction is committed, and both databases are in step. If the Cosmos DB fails, the SQL transaction is rolled back, and the admin person is alerted that the add of the new `Book` failed.**

16.4.1 Creating an event to trigger when the SQL Book entity changes

In this design, you want to update the Cosmos DB database when an `Add`, `Update`, or `Delete` integration event is found. But it's possible that when you add a `Book`, which creates an `Add` event, you may trigger an `Update` event too (that happens when seeding the database). Also, some complex updates, such as changing multiple parts of the entity, might trigger multiple `Update` events. At minimum, multiple events are inefficient, as you would update the Cosmos DB database multiple times, and, in certain cases, make your code more complex. The problem is that the event handler has no knowledge of other events, so you can't detect that the update is not needed. How do you reduce multiple events to one?

For this type of problem, the `GenericEventRunner` provides the `RemoveDuplicateEvents` attribute to remove duplicate events that are the same event type, and linked to the same class instance (as determined by the `ReferenceEquals` method). The following listing shows the `BookChangedEvent` with the `RemoveDuplicateEvents` attribute added.

Listing 16.1 The `BookChangedEvent` sending `Add`, `Update`, and `Delete` changes

This attribute causes the GenericEventRunner to remove duplicate events from the same Book instance.

The three types of changes that need mapping to the Cosmos DB database

When an event is created, you must say what type of change the Book has gone through.

```
public enum BookChangeTypes { Added, Updated, Deleted }

[RemoveDuplicateEvents]
public class BookChangedEvent : IEntityEvent
{
    public BookChangedEvent(BookChangeTypes bookChangeType)
    {
        BookChangeType = bookChangeType;
    }

    public BookChangeTypes BookChangeType { get; }
}
```

Used by the event handler to work out whether to add, update, or delete the CosmosBook

Holds the type of change for the event handler to use

As well as being more efficient, this listing makes the code that updates the Cosmos DB simpler, because an `Add` followed by an `Update` would cause problems with updating an entity with the same key that is already being tracked. This problem could be solved in the `Add`/`Update` Cosmos code, but removing duplicate events is easier, especially as that feature is built into the `GenericEventRunner` library.

16.4.2 Adding events to the Book entity send integration events

Because you are using DDD-styled entity classes, it is reasonably easy to spot all the places where a `Book` entity is created or updated. You simply add an `Added` event in the `Book`'s static factory and lots of `Update` events in any DDD access methods. The following listing shows an `Update` event being added via the `AddEvent` method (see section 12.4.2) if the update isn't rejected because of a user input error.

Listing 16.2 Adding a `BookUpdate` to a Book's `AddPromotion` method

```
public IStatusGeneric AddPromotion(
    decimal actualPrice, string promotionalText)
{
    var status = new StatusGenericHandler();
    if (string.IsNullOrWhiteSpace(promotionalText))
    {
        return status.AddError(
            "You must provide text to go with the promotion.",
            nameof(PromotionalText));
    }

    ActualPrice = actualPrice;
    PromotionalText = promotionalText;

    if (status.IsValid)
        AddEvent(new BookChangedEvent(
            BookChangeTypes.Updated),
            EventToSend.DuringSave);
```

You don't want to trigger unnecessary updates, so you trigger only if the change was valid.

Adds a BookChangedEvent event with the Update setting as a During (integration) event

```
    return status;
}
```

For the delete event, you are using a soft delete, so you capture a change to the `SoftDeleted` property via its access method. The options are

- If the `SoftDeleted` value isn't changed, no event is sent.
- If the `SoftDeleted` value is changed to `true`, a `Deleted` event is sent.
- If the `SoftDeleted` value is changed to `false`, an `Added` event is sent.

The following listing shows this example.

Listing 16.3 A change of `SoftDeleted` that triggers an `AddBook` or `DeleteBook` event

```
public void AlterSoftDelete(bool softDeleted)
{
    if (SoftDeleted != softDeleted)
    {
        var eventType = softDeleted
            ? BookChangeTypes.Deleted
            : BookChangeTypes.Added;

        AddEvent(new BookChangedEvent(eventType)
            , EventToSend.DuringSave);
    }
    SoftDeleted = softDeleted;
}
```

You don't trigger unnecessary updates, so you trigger only if there was a change to the SoftDeleted property.

The type of event to send depends on the new SoftDelete setting.

Adds the BookChangedEvent event as a During (integration) event

16.4.3 Using the EfCore.GenericEventRunner to override your BookDbContext

In section 15.5.1, you used the Cached SQL performance-tuning approach. The SQL (+cache) approach uses domain events, but this CQRS approach uses integration events. The Cached SQL approach and this CQRS approach can coexist, with each part having no knowledge of the other—another example of applying the SoC principle.

16.4.4 Creating the Cosmos entity classes and DbContext

The projection of the SQL `Book` needs to contain the normal properties, such as `Title` and `AcualPrice`, plus the values that take a lot of time to calculate, such as the count of the `Reviews` linked to the `Book`. The idea of the CQRS read side is to build a ready-to-display version, known as a *projection*, of the SQL `Book` so that it's quick to display. The following two listings show the `CosmosBook` class and the `CosmosTag` class that are used to hold the projection of the SQL `Book`.

Listing 16.4 The `CosmosBook` that holds the projection of the SQL `Book`

```
public class CosmosBook
{
    public int BookId { get; set; }
```

We use the BookId used in the SQL database to link this entity to the SQL entity.

```
    public string Title { get; set; }
    public DateTime PublishedOn { get; set; }
    public bool EstimatedDate { get;  set; }
    public int YearPublished { get; set; }
    public decimal OrgPrice { get; set; }
    public decimal ActualPrice { get; set; }
    public string PromotionalText { get; set; }
    public string ManningBookUrl { get; set; }

    public string AuthorsOrdered { get; set; }
    public int ReviewsCount { get; set; }
    public double? ReviewsAverageVotes { get; set; }

    public List<CosmosTag> Tags { get; set; }
    public string TagsString { get; set; }
}
```

Normal properties that are needed to display the Book

Precalculated values used for display and filtering

To allow filtering on TagIds we provide a list of CosmosTags, which are configured as Owned Types.

This string is used later to overcome a limitation of EF Core's current Cosmos DB provider.

Listing 16.5 The `CosmosTag` class that holds the `TagId` from the SQL `Book`

```
public class CosmosTag
{
    public string TagId { get; set; }
}
```

As you can see in listing 16.5, the `CosmosTag` class contains one property: `TagId`. This class mimics the `Tag` class used in the SQL database, but it will be added as an owned type (see section 8.9.1). Then the `CosmosBook`'s `Tags` collection holds each `Tag` string for the `Book`, allowing you to filter `Books` by a `Tag`, such as `Books` about `"Databases"`. The `CosmosTag` class is registered as an owned type (see section 8.9.1), so it is embedded in the data sent to Cosmos DB (see listing 16.10).

In fact, a common way to save data to Cosmos DB is to contain collections of other classes within the main class (Cosmos DB calls this approach *nesting*), which is what EF Core's owned types do. Consider using nested owned type classes when building data to be stored in a Cosmos DB.

The EF Core Cosmos DbContext is small and simple, as shown in listing 16.6, because many of the EF Core configuration commands don't work with a Cosmos DB database. You can't set the type of the data stored, as each property is converted to a JSON key/value, and other settings, such as indexing, are handled by Cosmos itself.

Listing 16.6 The `DbContext` class needed to access the Cosmos DB database

```
public class CosmosDbContext : DbContext
{
    public CosmosDbContext(
        DbContextOptions<CosmosDbContext> options)
        : base(options)
    { }

    public DbSet<CosmosBook> Books { get; set; }
```

The Cosmos DB DbContext has the same structure as any other DbContext.

For this use, you need to read/write only the CosmosBooks.

```
    protected override void OnModelCreating(
        ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<CosmosBook>()
            .HasKey(x => x.BookId);

        modelBuilder.Entity<CosmosBook>()
            .OwnsMany(p => p.Tags);
    }
}
```

BookId doesn't match the By Convention rules, so you need to configure it manually.

The collection of CosmosTags is owned by the CosmosBook.

NOTE For a full list of Cosmos DB-specific Fluent API commands, please see the EF Core documentation on the Cosmos database provider at http://mng.bz/8WyK.

16.4.5 Creating the Cosmos event handlers

The `BookChangedEvent` integration event comes into the `BookDbContext`, and you need a matching event handler. A `BookChangeType` property says whether the event is an `Add`, `Update`, or `Delete`, so it uses a C# `switch` to call the correct code. Because adding, updating, and deleting entries in the Cosmos database use similar code, you build a service that contains three methods, one for each type of update. Putting all the update code in a service makes the event handler simple, as the following listing shows.

Listing 16.7 An example Cosmos event handler that handles an `Add` event

```
public class BookChangeHandlerAsync
    : IDuringSaveEventHandlerAsync<BookChangedEvent>
{
    private readonly IBookToCosmosBookService _service;

    public BookChangeHandlerAsync(
        IBookToCosmosBookService service)
    {
        _service = service;
    }

    public async Task<IStatusGeneric> HandleAsync(
        object callingEntity, BookChangedEvent domainEvent,
        Guid uniqueKey)
    {
        var bookId = ((Book)callingEntity).BookId;
        switch (domainEvent.BookChangeType)
        {
            case BookChangeTypes.Added:
                await _service.AddCosmosBookAsync(bookId);
                break;
            case BookChangeTypes.Updated:
                await _service.UpdateCosmosBookAsync(bookId);
                break;
            case BookChangeTypes.Deleted:
                await _service.DeleteCosmosBookAsync(bookId);
                break;
```

Defines the class as a During (integration) event for the BookChanged event

This service provides the code to Add, Update, and Delete a CosmosBook.

The event handler uses async, as Cosmos DB uses async.

Extracts the BookId from the calling entity, which is a Book

The BookChangeType can be added, updated, or deleted.

Calls the Add part of the service with the BookId of the SQL Book

Calls the Update part of the service with the BookId of the SQL Book

Calls the Delete part of the service with the BookId of the SQL Book

```
            default:
                throw new ArgumentOutOfRangeException();
        }

        return null;
    }
}
```

Retuning null tells the GenericEventRunner that this method is always successful.

Remember that if the update to the Cosmos database fails, the SQL update, which was executed in a transaction, is rolled back so the databases are kept in step. But you want to minimize throwing an exception if the service can fix the problem itself, making some extra checks to catch states that it can fix.

The following listing shows the `MapBookToCosmosBookAsync` method that handles updating a `Book`. It's unlikely to happen, but in the time it took the `Update` event handler to trigger that SQL, `Book` might have been (soft-) deleted. Therefore, if the `MapBookToCosmosBookAsync` method returns `null`, it assumes that the `Book` has been deleted and will delete any existing `CosmosBook` with that `BookId`. Note the use of EF Core's `Update` in the code.

Listing 16.8 Creating a projection of the SQL `Book` and adding it to the Cosmos database

This method is called by the BookUpdated event handler with the BookId of the SQL book.

```
public async Task UpdateCosmosBookAsync(int bookId)
{
    if (CosmosNotConfigured)
        return;

    var cosmosBook = await MapBookToCosmosBookAsync(bookId);

    if (cosmosBook != null)
    {
        _cosmosContext.Update(cosmosBook);
        await CosmosSaveChangesWithChecksAsync(
            WhatDoing.Updating, bookId);
    }
    else
    {
        await DeleteCosmosBookAsync(bookId);
    }
}
```

The Book App can be run without access to Cosmos DB, in which case it exits immediately.

This method uses a Select method similar to the one in chapter 2 on a CosmosBook entity class.

If the CosmosBook is successfully filled, the Cosmos update code is executed.

Updates the CosmosBook to the cosmosContext and then calls a method to save it to the database

If the SQL book wasn't found, we ensure that the Cosmos database version was removed.

> **COSMOS DB DIFFERENCE** The Cosmos DB database always updates the whole of the entry for the given key in one go, unlike a relational database, which can change individual columns in a row. The EF Core `Update` method is more efficient because it saves a read of the Cosmos database.

The `CosmosSaveChangesWithChecksAsync` method is also designed to capture and fix any states that it might find. An update that doesn't find a `CosmosBook` to update, for example, will be turned into a new `CosmosBook` instead. These situations are rare but could happen due to concurrent updates to the same `CosmosBook` entity.

Listing 16.9 shows part of the `CosmosSaveChangesWithChecksAsync` method that detects errors, possibly caused by concurrency issues, making sure that the Cosmos database is up to date. The `catch` part of the code in the listing covers the following situations:

- `CosmosException`:
  - An `Update` in which the corresponding entity has been `Deleted`, turning the `Update` into an `Add`.
  - A `Delete` in which the corresponding entity was already `Deleted` (job done).
  - If not fixed, rethrow the exception.
- `DbUpdateException`:
  - An `Add` of a new entity when a corresponding entity already exists there, turning the `Add` into an `Update`.

This code shows another useful difference when using the Cosmos DB provider.

Listing 16.9 Part of the handling of `SaveChanges` exceptions with Cosmos DB

```
private async Task CosmosSaveChangesWithChecksAsync
    (WhatDoing whatDoing, int bookId)
{
    try
    {
        await _cosmosContext.SaveChangesAsync();
    }
    catch (CosmosException e)
    {
        if (e.StatusCode == HttpStatusCode.NotFound
            && whatDoing == WhatDoing.Updating)
        {
            var updateVersion = _cosmosContext
                .Find<CosmosBook>(bookId);
            _cosmosContext.Entry(updateVersion)
                .State = EntityState.Detached;

            await AddCosmosBookAsync(bookId);
        }
        else if (e.StatusCode == HttpStatusCode.NotFound
                 && whatDoing == WhatDoing.Deleting)
        {
            //Do nothing as already deleted
        }
        else
        {
            throw;
        }
    }
    catch (DbUpdateException e)
```

Calls SaveChanges and handles certain states

The whatDoing parameter tells the code whether this is an Add, Update, or Delete.

Catches any CosmosExceptions

Catches an attempt to update a CosmosBook that wasn't there

You need to remove the attempted update; otherwise, EF Core will throw an exception.

Turns the Update into an Add

Catches the state where the CosmosBook was already deleted...

...otherwise, not an exception state you can handle, so rethrow the exception

If you try to add a new CosmosBook that's already there, you get a DbUpdateException.

```
        {
            var cosmosException = e.InnerException as CosmosException;
            if (cosmosException?.StatusCode == HttpStatusCode.Conflict
                && whatDoing == WhatDoing.Adding)
            //… rest of code left out as nothing new there
        }
    }
```

The inner exception contains the CosmosException.

Catches an Add where there is already a CosmosBook with the same key

COSMOS DB DIFFERENCE I found the `CosmosException` to be helpful for diagnosing Cosmos database issues. The `CosmosException` contains a `StatusCode` property that uses HTTTP status codes, such as `NotFound` and `Conflict`, to describe what went wrong.

16.5 Understanding the structure and data of a Cosmos DB account

Before moving on to the query of the `CosmosBook` class, it is worth looking at how Cosmos DB is organized and what the data looks like when EF Core writes to a database. These sections explain how to use a Cosmos DB database account to access a Cosmos database in your application and look at the JSON data stored in the Cosmos database.

NOTE The EF Core Cosmos DB provider uses the Cosmos SQL API. which presents a traditional NoSQL document store using JSON. But Cosmos DB has multiple ways to handle data, such as column store; key-value and graph; and multiple APIs, such as MongoDB, Cassandra, Azure Table, and Gremlin (graph).

16.5.1 The Cosmos DB structure as seen from EF Core

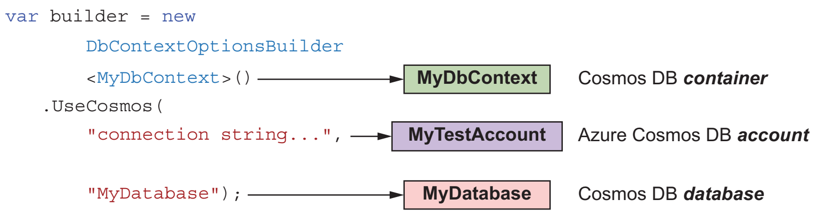
This section provides a quick summary of the various parts of the Cosmos DB structure. It isn't a detailed explanation (the Azure documentation offers one), but it provides the terms you need to use Cosmos DB with EF Core.

Azure provides an Azure Cosmos DB account, which is like a database server, as you can have multiple databases in one Azure Cosmos DB account. This account can be accessed via a connection string, made up of two parts: the URI to access the Cosmos DB account and an account key. This combination allows you to access your Azure Cosmos DB account.

NOTE An Azure Cosmos DB Emulator provides a local (and free) version of the Cosmos DB account. It also contains a feature that allows you to read and manage databases that it has stored locally. I cover Azure Cosmos DB Emulator in section 17.8.

A Cosmos DB account can have many Cosmos DB databases; each database can have many Cosmos DB containers; and containers are where the data is held. Figure 16.3 shows how the EF Core code maps onto the Cosmos DB structure.

Having configured the `DbContextOptionsBuilder<T>` class (or registered the Cosmos DbContext via the `AddDbContext` method), you can obtain an instance of the application's DbContext and are ready to access the Cosmos DB database.



```
var builder = new
        DbContextOptionsBuilder
        <MyDbContext>()
    .UseCosmos(
        "connection string...",

        "MyDatabase");
```

Figure 16.3 Mapping the EF Core setup of a Cosmos DB database provider to the three levels in the Cosmos DB system. The Azure Cosmos DB account can have many Cosmos databases, but this figure shows only one. The database's name is defined in the `UseCosmos` method. A Cosmos DB database can have many containers, but when used by EF Core, it allows only one container per EF Core application DbContext. By default, the container is given the name of the application DbContext's class.

16.5.2 How the CosmosClass is stored in Cosmos DB

When you have a correctly configured application DbContext for a Cosmos DB database, you can read and write to its database—strictly, the Cosmos DB's container, but to keep the discussion EF Core-centric I'm going to use the word *database*. For normal read/writes, you don't need to know how the data is stored in the Cosmos database, but sometimes that information is useful, as it gives you an idea of what is being stored.

The next listing shows the data stored when you write a `CosmosBook` to the database. As you will see, extra properties at the end aren't in the `CosmosBook` class but are critical for making Cosmos DB work.

Listing 16.10 The `CosmosBook` data stored as JSON in Cosmos DB

```
{
  "BookId": 214,
  "ActualPrice": 59.99,
  "AuthorsOrdered": "Jon P Smith",
  "EstimatedDate": true,
  "ManningBookUrl": "
  "OrgPrice": 59.99,
  "PromotionalText": null,
  "PublishedOn": "2021-05-15T05:00:00+01:00",
  "ReviewsAverageVotes": 5,
  "ReviewsCount": 1,
  "Title": "Entity Framework Core in Action, Second Edition",

  "Tags": [
    {
      "TagId": "Databases"
    },
    {
      "TagId": "Microsoft & .NET"
    }
  ],
```

The standard properties from the CosmosBook class

Holds the collection of Tags, which are configured as an owned type

```
    "YearPublished": 2021,
    "TagsString": "| Databases | Microsoft & .NET |",

    "Discriminator": "CosmosBook",

    "id": "CosmosBook|214",

    "_rid": "QmRlAMizcQmwAg…",
    "_self": "dbs/QmRlAA==/colls/QmRlAMizcQk=…",
    "_etag": "\"1e01b788-0000-1100-0000-5facfa2f0000\"",
    "_ts": 1605171759,
    "_attachments": "attachments/"
  }
```

These two properties are added to overcome some limitations in the EF Core 5 Cosmos provider.

Cosmos-specific properties; see the following notes

The id is the database's primary key and must be unique. This id is set by EF Core, using the EF Core designated primary key and the discriminator.

EF Core adds the discriminator to differentiate this class from other classes saved In the same Cosmos container.

The first set of JSON key/values comes from the properties and relationships in the `CosmosBook` class, including the `Tags` collection:

- The `id` key/value is the unique key used to define this data. EF Core fills the unique key with a value—by default, a combination of the `Discriminator` value and the value from the property(s) that you told EF Core is the primary key of this entity class.
- The `_etag` key/value can be used with the `UseETagConcurrency` Fluent API method to provide a concurrency token covering any change in the data.
- The `_ts` key/value contains the time of the last `Add`/`Update` in Unix format and is useful for finding when an entry last changed. The `_ts` value can be converted to C# `DateTime` format by using the `UnixDateTimeConverter` class.
- The `_rid` and `_self` key/value are unique identifiers used internally for navigation and resources.
- The `_attachments` key/value is depreciated and is there only for old systems.

16.6 Displaying books via Cosmos DB

Having built a system that copies changes in the SQL `Book` entity class to a Cosmos database, we are ready to implement the book-display features of the original Book App by getting data from the Cosmos DB database. Implementing all the book-display features exposes several interesting Cosmos DB differences from the way a relational database works.

In the end, I could match the original book display, but it is interesting to understand the differences, which tell me what I can achieve by using a Cosmos DB database. I also built the display with direct Cosmos DB commands by using its .NET SDK (software development kit), which I refer to as Cosmos (Direct). The Cosmos (Direct) code allowed me to differentiate between EF Core 5 Cosmos database provider limitations and differences in the way that Cosmos DB natively queries a database.

> **EF CORE 5 LIMITATION** EF Core's raw SQL commands, such as `FromSqlRaw` and `FromSqlInterpolated`, don't work. But you can get a `CosmosClient` instance via `var cosmosClient = context.Database.GetCosmosClient()`. This technique allows you to use the Cosmos DB .NET SDK commands.

Here are the variations from relational databases and the EF Core 5 limitations that I found while implementing the two-database CQRS architecture:

- *Cosmos DB differences from relational databases*
- *Cosmos DB/EF Core difference: Migrating a Cosmos database*
- *EF Core 5 Cosmos DB database provider limitations*

> **NOTE** If you want to try running the Book App with Cosmos DB, download the associated GitHub repo (http://mng.bz/XdlG), run the BookApp.UI project, and look for the Chapter 16 Setup link on the home page for more info.

16.6.1 Cosmos DB differences from relational databases

This section covers the differences between a Cosmos DB (NoSQL) database and a relational (SQL Server) database. This information is useful for developers who haven't worked with NoSQL and, more specifically, a Cosmos DB database before. Here is a summary of the various differences:

- The Cosmos DB provides only async methods.
- There are no database-created primary keys.
- Complex queries may need breaking up.
- Skip is slow and expensive.
- By default, all properties are indexed.

THE COSMOS DB PROVIDES ONLY ASYNC METHODS

Because Cosmos DB uses HTTP to access databases, all the methods in the Cosmos DB .NET SDK use async/await, and there are no sync versions. EF Core does provide access to Cosmos DB via EF Core's sync methods, such as `ToList` and `SaveChanges`, but these methods currently use the `Task`'s `Wait` method, which can have deadlock problems.

I strongly suggest that you use only async EF Core methods when working with the Cosmos database provider. In addition to getting a more robust application, you will get better scalability in multiuser situations, such as ASP.NET Core.

COSMOS DIFFERENCE: THERE ARE NO DATABASE-CREATED PRIMARY KEYS

With a relational database, you are used to having the database provide a unique value for its primary key when a new row is added to a table. But in Cosmos and many other NoSQL databases, by default, the key for an item (item is Cosmos's name for each JSON entry) must be generated by the software before you add an item.

> **NOTE** The Cosmos DB has a way to create a unique key for you, but this key will be stored in the `id` key/value.

The key for an item must be unique, and Cosmos will reject (with the HTTP code `Conflict`) a new item if its key was already used. Also, after you have added an item with a key, you can't change the key.

One easy choice for a Cosmos DB key is a C# `Guid` type, which is designed to be unique. EF Core also makes using a `Guid` type as a key simple, as it has a built-in value generator (see section 10.3.2) that will provide a new `Guid` value if the designated primary key is a `Guid` and its value is `default`. You can configure composite keys with EF Core, which will combine their values into a string that Cosmos DB needs for its `id` key/value. When using Cosmos in the Book App, I used an `int` as the key for the `CosmosBook` entity, but the `int`'s value came from the primary key that the SQL write-side database created.

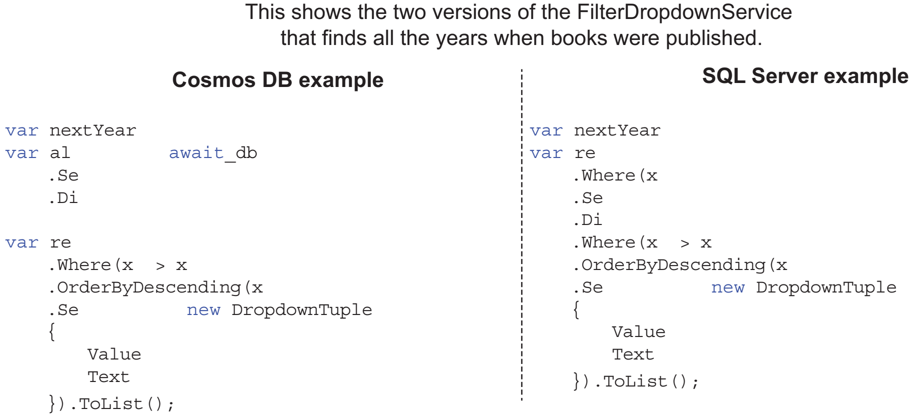
> **NOTE** Cosmos DB talks about a partition key and logical and physical partitions. I'm not covering these topics here, as they're big topics, and I'm not sure I understand them well enough. EF Core 5 defaults to not having a partition key, but you can change that setting.

COMPLEX QUERIES MAY NEED BREAKING UP

In the filter-by-year option in the book display, the `FilterDropdownService` finds all the years when books were published. This task requires a series of steps:

1. Filter out any books that haven't yet been published.
2. Extract the `Year` part of the Book's `PublishedOn DateTime` property.
3. Apply the LINQ `Distinct` command to obtain the years for all the published books.
4. Order the years.

This complex query works in SQL, but Cosmos DB can't handle it. Figure 16.4 shows a side-by-side view of the two queries.



This shows the two versions of the FilterDropdownService that finds all the years when books were published.

Cosmos DB example

```
var nextYear
var al          await_db
    .Se
    .Di

var re
    .Where(x  > x
    .OrderByDescending(x
    .Se            new DropdownTuple
    {
        Value
        Text
    }).ToList();
```

SQL Server example

```
var nextYear
var re
    .Where(x
    .Se
    .Di
    .Where(x  > x
    .OrderByDescending(x
    .Se            new DropdownTuple
    {
        Value
        Text
    }).ToList();
```

Figure 16.4 Two versions of the `FilterDropdownService` that finds all the years when books were published. The Cosmos DB example simplifies the query that is run in the Cosmos DB, with the second part done in the software. This example shows that Cosmos DB doesn't have the wide range of query features that relational databases have.

When I ran the code I was using in SQL Server (see the right side of figure 16.4), I got an exception in Cosmos DB, with a link to EF Core issue #16156, which says that Cosmos DB has some limitations on queries. Cosmos doesn't have the massive depth of query features that relational databases have gained over decades of improvement, so you may have to alter some of your more complex queries when dealing with Cosmos DB. Here is what I did to make the filter drop-down query work in Cosmos DB:

- I added a new property called `YearPublished` that had the year as an integer. (I tried using a Cosmos DB user-defined function to extract the year, but it wouldn't work with the `Distinct` command.) This property is filled in during the projection of the SQL `Book` entity by the `Year` part of the `DateTime PublishedOn` property.
- I ran the `Distinct` query by using the `YearPublished` value in Cosmos and then ordered the returned years in software.

My two changes to the code makes the Cosmos query work, but it's slow (section 16.7.2). But the takeaway from this section is that you shouldn't be applying queries with multiple parts to the Cosmos DB database whether you're using EF Core or not. The strength of a Cosmos DB database is its scalability and availability, not its ability to handle complex queries.

SKIP IS SLOW AND EXPENSIVE

In the Book App, I used paging to allow the user to move through the books display. This type of query uses the LINQ `Skip` and `Take` methods to provide paging. The query `context.Books.Skip(100).Take(10)`, for example, would return the 101st to 111th books in a sequence. Cosmos DB can do this too, but the `Skip` part gets slower as the skip value gets bigger (another difference from relational databases) and is expensive too.

> **COSMOS REQUEST UNITS** Azure's Cosmos DB database uses *request units* (RUs) to manage provisioning the throughput of a container. You have various ways to provision your Cosmos DB container: fixed provisioning (fixedish price), serverless (pay as you use), and autoscale (scales to use). In the end, however, you are going to pay for every access to the Cosmos DB service.

It seems that if you `Skip` 100 items, Cosmos still reads them. But even though Cosmos doesn't send the `Skip`ped items to the application, there is a time and cost in RUs. In the Book App, you can see performance go down as the user goes farther down the list of books (see figure 16.8).

Whether the `Skip` performance is a problem depends on your application. In the Book App, I doubt that people would read much past the first 100 books. But this example suggests that showing 100 books at a time is better than showing 10 books and having the user page, as paging isn't free.

BY DEFAULT, ALL PROPERTIES ARE INDEXED

We know that adding an index to a property in a relational database significantly improves the time it takes to filter or sort on that property, with a (small) performance cost when you update the indexed property. Cosmos DB's default setup is to index all the key/values, included nested key values. (The `CosmosBook` entity has the `Tags.TagId` key/values indexed too, for example.) You can change the Cosmos DB indexing policy, but "index all" is a good starting point.

> **NOTE** EF Core's indexing configuration features, including a unique index, don't work in Cosmos DB. But you can define indexes via the Cosmos DB setup section of the container.

You should also remember that Cosmos DB saves data by using JSON string format, and Cosmos indexes knows about only three index types: numbers, strings, and geography. C#'s `DateTime` and `TimeSpan` types are stored in a string format that can be sorted or filtered by means of a string, so date and time are stored with the more significant time parts first, as in `YYYY-MM-DDTHH:MM:SS`. EF Core handles time-to-string conversions for you, but if you use EF Core's value converters (see section 7.8) or raw SQL queries, you need to understand the various JSON formats that Cosmos DB uses.

16.6.2 Cosmos DB/EF Core difference: Migrating a Cosmos database

Cosmos DB is a *schemaless* database, meaning that each item doesn't have to have the same properties or nested data in each item. Each item is a JSON object, and it's up to you what keys/values you put in the JSON object. This database is different from a relational database, in which the schema is important and requires some effort to change (see chapter 9).

At some point, you are going to change or add properties to an entity class mapped to a Cosmos DB database. You must be careful, though; otherwise, you could break some of your existing Cosmos DB queries. This example shows what can go wrong and how to fix it:

1. You have a `CosmosBook` entity class, and you have written data to a Cosmos DB database.
2. You decide that you need an additional property called `NewProperty` of type `int` (but it could be any non-nullable type).
3. You read back old data that was added before the `NewProperty` property was added to the `CosmosBook` entity class.
4. At this point, you get an exception saying something like `object must have a value`.

Cosmos DB doesn't mind your having different data in every item, but EF Core does. EF Core expects a `NewProperty` of type `int`, and it's not there. The way around this problem is to make sure that any new properties are nullable; then reading the old data will return a `null` value for the new properties. If you want the new property to be non-nullable, start with a nullable version and then update *every* item in the database with a non-null value for the new property. After that, you can change the new property back to a non-nullable type, and because there is a value for that property in every item, all your queries will work.

Another point is that you can't use the `Migrate` command to create a new Cosmos DB database, because EF Core doesn't support migrations for a Cosmos DB database. You need to use the `EnsureCreatedAsync` method instead. The `EnsureCreatedAsync` method is normally used for unit testing, but it's the recommended way to create a database (Cosmos DB container) when working with Cosmos DB.

16.6.3 EF Core 5 Cosmos DB database provider limitations

This section covers the limitations of the EF Core 5 Cosmos DB database provider. This information is useful if you want to use EF Core 5 to access a Cosmos DB database; it'll also be useful when future releases of EF Core remove some of these limitations, making the workarounds I had to apply to the part 3 Book App unnecessary. Here is a summary of the various limitations:

- Counting the number of books in Cosmos DB is *slow*!
- Many database functions are not implemented.
- EF Core 5 cannot do subqueries on a Cosmos DB database.
- There are no relationships or `Includes`.

COUNTING THE NUMBER OF BOOKS IN COSMOS DB IS SLOW

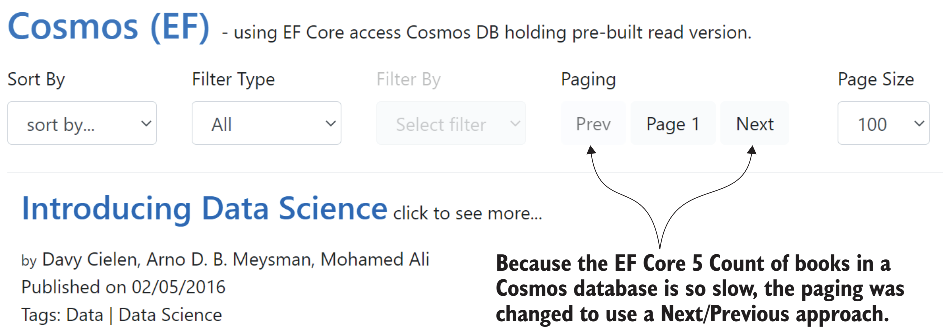
Almost the first thing I noticed when I added a Cosmos version to the Book App was that counting the `CosmosBooks`, which I used for paging, was extremely slow via EF Core. I built a mini version of this Book App in late 2019, and there were two reasons for the poor performance:

- Cosmos DB's aggregates (`Count`, `Sum`, and so on) were slow and took a lot of RUs to run.
- EF Core didn't use Cosmos DB's aggregates, so I needed to read every `CosmosBook` in to count them (a EF Core 5 limitation).

Fortunately, the first issue was fixed in April 2020. Cosmos DB's aggregates are much quicker and uses a lot less resources. (Example: the original Cosmos `Count` took 12,000 RUs, whereas the new `Count` used only 25 RUs.) But EF Core 5 didn't get any quicker, because it was reading all the books in the Cosmos database to count them. Not to be thwarted, I changed the way the Cosmos EF book display worked and moved over to using a `Next/Previous` approach to paging. Figure 16.5 shows this format.

Changing to the `Next/Previous` approach was trivial; the main problem was setting up the ASP.NET Core Razor page. Many e-commerce sites, including Amazon, use this approach, so this change might be a good one to make anyway.

In the Cosmos (Direct) version, I kept the normal paging, with its count of all filtered books. It turns out that the direct Cosmos `Count` command, `SELECT value COUNT(c) FROM c`, is fast (~25 ms to count 500,000 Cosmos books) even compared with the SQL version (90 ms to count 500,000 SQL books).



Cosmos (EF) - using EF Core access Cosmos DB holding pre-built read version.

Sort By: sort by...

Filter Type: All

Filter By: Select filter

Paging: Prev | Page 1 | Next

Page Size: 100

Introducing Data Science click to see more...

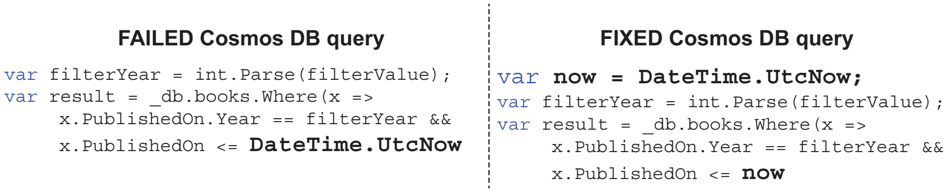
by Davy Cielen, Arno D. B. Meysman, Mohamed Ali
Published on 02/05/2016
Tags: Data | Data Science

Because the EF Core 5 Count of books in a Cosmos database is so slow, the paging was changed to use a Next/Previous approach.

Figure 16.5 The page where the Cosmos DB is accessed via EF Core 5. To overcome the slow speed of counting the number of books, I changed the controls to use a `Next/Previous` approach.

MANY DATABASE FUNCTIONS ARE NOT IMPLEMENTED

EF Core 5 has mapped LINQ to a small set of five Cosmos functions, so you may have to alter your LINQ to work around these issues. One filter I tried failed because EF Core knew that it should convert the method `DateTime.UtcNow` to a UTC date from the database server, but that Cosmos DB function hadn't been mapped in EF Core 5. The problem was easy to fix: I created a variable to hold the value given by `DateTime.UtcNow`. Figure 16.6 shows the failed (left) and the fixed (right) query, with the differences in **`bold`**.



FAILED Cosmos DB query

```
var filterYear = int.Parse(filterValue);
var result = _db.books.Where(x =>
     x.PublishedOn.Year == filterYear &&
     x.PublishedOn <= DateTime.UtcNow
```

FIXED Cosmos DB query

```
var now = DateTime.UtcNow;
var filterYear = int.Parse(filterValue);
var result = _db.books.Where(x =>
     x.PublishedOn.Year == filterYear &&
     x.PublishedOn <= now
```

Figure 16.6 The original query on the right failed, because EF Core 5 knew that it should convert the DateTime.`UtcNow` method (left, in bold) to the UTC time provided by the database server, but that part of the mapping had not been done. The solution was to put the value from the `DateTime`.UtcNow method in a variable (top right, in bold) and feed that variable into the query.

EF CORE 5 CANNOT DO SUBQUERIES ON A COSMOS DB DATABASE

The Book App provides a way to filter books by their `Tags`, such as looking only at books with a `Tag` called `"Databases"`. This solution requires a subquery in the main query, like the Cosmos DB SQL command shown in the following code snippet:

```
SELECT DISTINCT value f.TagId FROM c JOIN f in c.Tags
```

This EF Core 5 limitation precludes querying any nested parts of the Cosmos JSON, such as any owned types that are saved with the main entity class. You can get that data by reading the entity, of course, but you can't filter, sort, or select nested parts on their own via EF Core. In section 16.7.2, I show you a way to get around this problem.

THERE ARE NO RELATIONSHIPS OR INCLUDES

The EF Core 5 Cosmos database provider doesn't support relationships between entity classes (other than via owned types embedded in the main entity class). Although this lack of support seems to be a big missing feature, when it comes to Cosmos entities, owned types are the way to go, so maybe this feature doesn't matter so much.

The design approach of a Cosmos DB item is more about embedding (Cosmos calls it *nesting*), which you can do with owned types, such as the `Tags` collection in `CosmosBook`. In fact, the Cosmos DB documentation (http://mng.bz/EVnq) says

> *Because there is currently no concept of a constraint, foreign-key or otherwise, any inter-document relationships that you have in documents are effectively "weak links" and will not be verified by the database itself.*

Most NoSQL databases are like Cosmos DB in not supporting relationships between items. Personally, I'm not sure that EF Core should add relationships across different items in a Cosmos database, as they aren't going to work in the way we expect with relational databases, but we will see.

16.7 Was using Cosmos DB worth the effort? Yes!

You built a CQRS two-database system to improve the performance and scalability of the Book App. Also, implementing the CQRS system with Cosmos DB taught you a lot about what Cosmos can and can't do, as well as the limitations of the EF Core 5 Cosmos provider. In this section, you are going to look at three topics:

- The performance of the two-database CQRS in the Book App
- The features that the EF Core 5 Cosmos DB database provider can't handle
- How difficult it would be to use this two-database CQRS design in your application

To compare performance and features, you use four types of queries:

- *Cosmos (EF)*—Uses EF Core's Cosmos DB database provider
- *Cosmos (Direct)*—Uses the Cosmos DB .NET SDK
- *SQL (+cache)*—Uses the cached values in the SQL database (see section 15.5)
- *SQL (Dapper)*—Uses the best SQL to access the SQL database (see section 15.4)

NOTE I left out the original book-display code developed in chapter 2 because it was so slow that it wasn't useful. Also, it threw an exception on queries that exceeded the database timeout of 30 seconds.

The aim is to compare the performance, features, and development effort, as I did in section 15.6 for four levels of SQL performance tuning.

16.7.1 Evaluating the performance of the two-database CQRS in the Book App

To compare the performance of the SQL approaches in chapter 15 and the Cosmos CQRS system in this chapter, I had two types of Cosmos DB queries, using EF Core and direct via the Cosmos SQL API, and two SQL queries from chapter 15, using SQL (+cache) and SQL (Dapper). Having these four ways of displaying the books allowed me to compare the performance of the two types of databases.

To make the comparison fair, the two databases need to be

- *Located in the same place* so that the travel time (latency) is the same. I achieved this by creating both databases on the Azure site in London, which is about 50 miles from my location.
- *Similar in price* because price defines the performance of the two databases. The databases are close in price and cheap enough to test without spending lots of money. Table 16.1 shows details on the two databases.

Table 16.1 The two databases used to compare the performance of an SQL database and a Cosmos DB database

Database type	Azure service name	Performance units	Price/month
Azure SQL Server	Standard	20 DTUs	$37
Cosmos DB	Pay as you go	Manual scale, 800 RUs	$47

NOTE Both Azure SQL Server and Cosmos DB have a serverless version, in which the performance of the database can rise and fall based on demand. That version might have been cheaper for me, but I wanted specific performance to compare SQL queries with Cosmos DB queries.

The following list shows the levels of `Books` (both SQL `Book` and `CosmosBook`) in the databases that were used in the performance tests. It also shows the number of `Reviews` in the database, as sorting or filtering by votes is one of the most challenging queries:

- 100,000 `Books`, which has 546,000 `Reviews`
- 250,000 `Books`, which has 1,365,000 `Reviews`
- 500,000 `Books`, which has 2,740,000 `Reviews`

My first attempt at measuring the performance at different sizes of databases included the SQL (+cache) and SQL (Dapper) queries described in chapter 15. But it turns out that performance in counting the number of `Books` in a query is slow. At 500,000 `Books`, a simple display of the first 100 books took 230 ms. I felt that this evaluation between Cosmos (EF) and SQL (EF) wasn't fair, so I created SQL (+cacheNC) and SQL (DapperNC) versions. (*NC* stands for *no count.*) The first performance chart, which looks at performance as the database grows, contains only Cosmos DB (EF) and Cosmos DB (Direct), as shown in figure 16.7.

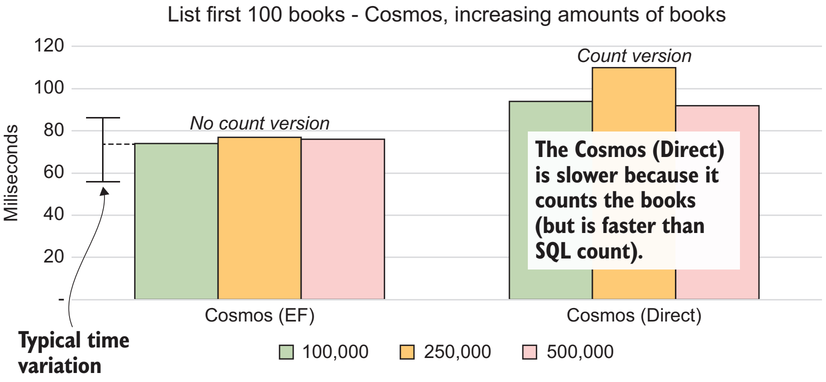


Figure 16.7 Time taken to display an HTML page containing the first 100 books (ordered by primary key, descending) in the Cosmos DB container for three sizes of the database. This figure shows that the size of the database has little effect on the time taken. Note that these timings were done days apart, and the variation is fairly large (~ 35 ms), so this chart might look different if I ran the test again.

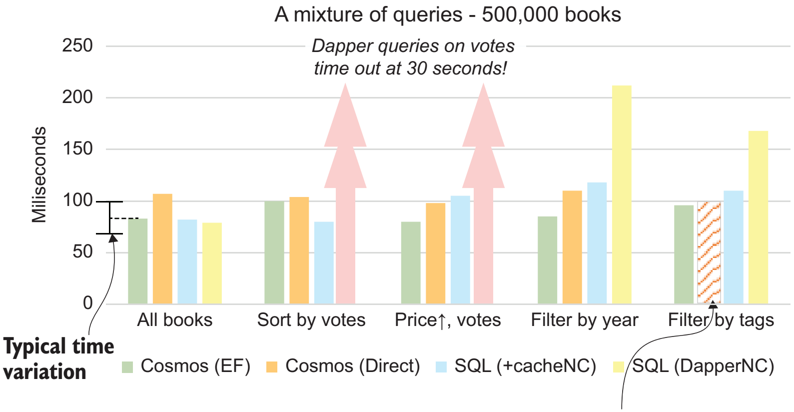
> **NOTE** All timings were taken from the ASP.NET Core's `RequestFinished` log, which contains the total time to the HTTP request. The time for a query is obtained by running the query at least seven times and taking the average of the last five times. To access this data, choose the Admin > Timings command in the Book App.

The main feature difference between Cosmos DB (EF) and Cosmos DB (Direct) is that the Cosmos DB (Direct) uses the original paging approach, which means that it had to count the number of `Books` in the overall query. Figure 16.7 shows that Cosmos DB is fast at counting—in fact, about twice as fast as SQL for 500,000 `Books`. In this case, speed doesn't matter too much, but in some applications, Cosmos's fast counting could make a big difference. The next performance tests were on most of the key sorts and filters across the four types of queries: Cosmos DB (EF), Cosmos DB (Direct), SQL (+cacheNC), and SQL (DapperNC) at 500,000 `Books`, as shown in figure 16.8.

> **NOTE** I discuss the effect of extracting the tags from 500,000 `CosmosBooks` in section 16.7.2, in the sidebar titled "Interesting things happen when you overload a Cosmos DB database."

Figure 16.8 and the SQL count information provide the information to make some conclusions, which are presented in the following list with the important facts first.

- Even the best SQL version, SQL (DapperNC), doesn't work in this application because any sort or filter on the `Reviews` took so long that the connection timed out at 30 seconds.
- The SQL (+cacheNC) version was at parity or better with Cosmos DB (EF) on the first two queries, but as the query got more complex, it fell behind in performance.



The Cosmos (Direct) filter by tags is fast, but extracting the tags from 500,000 CosmosBooks takes 400 ms.

Figure 16.8 Time taken for five key queries on databases containing 500,000 `Books`. The four types of database access are Cosmos DB (EF), Cosmos DB (Direct), SQL (+cacheNC), and SQL (DapperNC).

- The Cosmos DB (Direct), with its `Book` count, was ~25% slower than the Cosmos DB (EF) with no count, but it's still about twice as fast as an SQL count.

Overall, I think this test shows a good win for Cosmos DB, especially when you add the fact that implementing this CQRS was easier and quicker than building the original SQL (+cache) version. Also, Cosmos DB's concurrency handling (see section 16.4.5) is easier than the SQL (+cache) version.

The CQRS/Cosmos DB approach has some downsides, of course. First, adding and updating a book take a bit longer because the CQRS requires four database accesses: two to update the SQL database and two to update the Cosmos database. These updates add up to about 110 ms, which is more than double the time a single SQL database would take. So if your application does lots of writes to the database, this approach on its own might not work for you.

ADVANCED NOTE There are several ways to improve the write performance of the CQRS approach at the expense of more-complex code. I describe some of these approaches in one of my articles; see http://mng.bz/N8dE.

The second downside is a feature of Cosmos DB: using the LINQ `Skip` method is slow and expensive (see section 16.6.4). Figure 16.9 shows that the more books you `Skip`, the more time the process takes. Time shouldn't be a problem with the Book App, as many people would give up after a few pages, but if your application needs deep skipping through data, Cosmos DB is not a good fit.

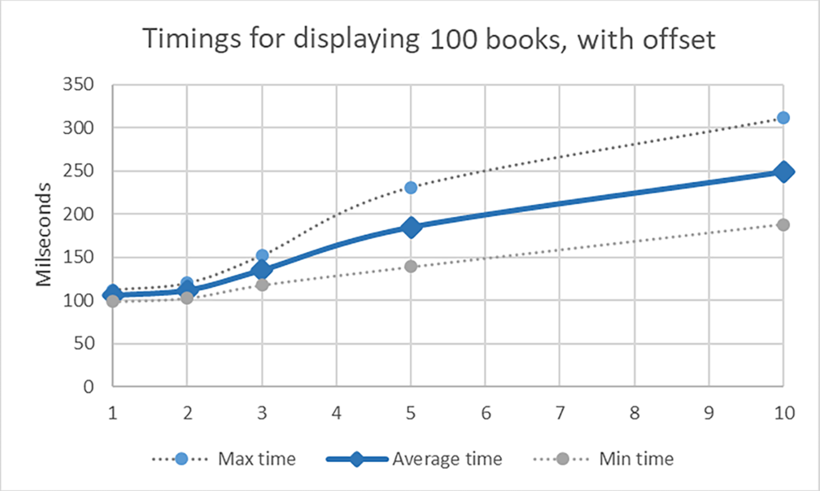


Figure 16.9 The more books you `Skip`, the more execution time increases. The chart shows the maximum and minimum ranges of the five timings that were used. As you can see, the more items `Skip`ped, the more the variation increased.

16.7.2 Fixing the features that EF Core 5 Cosmos DB database provider couldn't handle

In creating the original implementation of querying the Cosmos DB via the EF Core 5 Cosmos DB database provider, I limited myself to using only the features that EF Core 5 provided. But when you are building a real application, you use what you have and then improvise, because making the application work is what you are paid for. In this section, you are going to fix the problems already highlighted in this chapter:

- Couldn't count the number of books in Cosmos DB quickly
- Couldn't create the By Years Published drop-down filter in a reasonable time
- Couldn't create the By Tags drop-down filter
- Couldn't filter by `TagIds` because EF Core 5 Cosmos doesn't support the `IN` command

NOTE This section is only about handling limitations in the EF Core 5 Cosmos DB database provider. It does not cover Cosmos DB limitations, such as the need to break up complex queries (see section 16.6.3).

COULDN'T COUNT THE NUMBER OF BOOKS IN COSMOS DB QUICKLY

This issue is covered in section 16.6.8. Swapping to a `Next/Previous` form of paging is quite acceptable in many places. Amazon uses the `Next/Previous` paging approach, so it should work for selling books.

COULDN'T CREATE THE BY YEARS PUBLISHED DROP-DOWN FILTER IN A REASONABLE TIME

When you select the By Years Published drop-down filter, the code must look through all the books to find the `YearPublished` property and use the `Distinct` method to obtain all the years. The problem here wasn't that the LINQ query couldn't be run, but it was quite slow (25 seconds on 500,000 books). I suspect that it was slow because the `Distinct` method is run in software, but I can't be sure.

But I do know that using direct Cosmos SQL commands can work. In fact, the direct Cosmos SQL is quicker than the SQL version. For 500,000 books, Cosmos SQL took ~400 ms, whereas SQL took ~2.5 seconds. So the solution is to use a direct Cosmos SQL command to get the distinct years, which requires getting the Cosmos DB container via the Cosmos DB context, as shown in the following listing.

Listing 16.11 The `Filter Drop-down` service showing the use of direct Cosmos SQL

```
//… other parts of the switch removed for clarity
case BooksFilterBy.ByPublicationYear:

    var container = _db.GetCosmosContainerFromDbContext(
        _settings.CosmosDatabaseName);

    var now = DateTime.UtcNow;
    var comingSoonResultSet =
        container.GetItemQueryIterator<int>(
        new QueryDefinition(
            "SELECT value Count(c) FROM c WHERE" +
            $" c.YearPublished > {now:yyyy-MM-dd} " +
            "OFFSET 0 LIMIT 1"));
    var comingSoon = (await
            comingSoonResultSet.ReadNextAsync())
        .First() > 0;

    var resultSet = container.GetItemQueryIterator<int>(
        new QueryDefinition(
            "SELECT DISTINCT VALUE c.YearPublished FROM c" +
            $" WHERE c.YearPublished > {now:yyyy-mm-dd}"));

    var years = (await resultSet.ReadNextAsync()).ToList();

    //… the code turns the 'years' into a drop-down tuple
```

This code covers only the section that handles filtering by publication year.

Obtains a Cosmos DB container via the Cosmos DB context plus the name of the database

This query is designed to see whether there are any publications that aren't out yet.

The comingSoonResultSet is executed, and its single value tells us whether there are future publications in the list.

This query gets the distinct years for all books already published.

Executes the query and gets a list of years when books were published

But be warned: the query shown in listing 16.11 is another high-RUs query, coming out about the same as `TagIds` at 2,321 RUs. This chapter may be a place for a static list, as technical books more than five years old normally aren't useful (except for Eric Evan's *Domain-Driven Design*, of course!).

COULDN'T CREATE THE BY TAGS DROP-DOWN FILTER

EF Core 5 couldn't get a distinct set of `TagIds` from the `Tags` collection of each `CosmosBook` because EF Core 5's Cosmos database provider doesn't support subqueries. Again, you can use direct Cosmos SQL commands instead—Cosmos (Direct) takes only ~350 ms—but doing so is costly. Getting the list of `TagIds` from the SQL database is simple because it has a table called Tags, with only 35 rows. So instead of looking through 500,000 `CosmosBooks` and extracting all the `TagIds`, we can simply run the following SQL code, which takes only ~30 ms:

```
var drop-down = _sqlContext.Tags
    .Select(x => new Drop-downTuple
    {
        Value = x.TagId,
        Text = x.TagId
    }).ToList();
```

Interesting things happen when you overload a Cosmos DB database

In building the Cosmos (Direct) to filter by tags, I decided to extract the `TagIds` by using a Cosmos SQL command:

```
SELECT DISTINCT value f.TagId FROM c JOIN f in c.Tags
```

This command works but takes a long time (~400 ms) and costs a lot of RUs—2,445 RUs, to be exact. Because that command exceeds the 800 RUs provisioned for my database (Cosmos container), Cosmos penalizes any queries that come after it.

In this case, Cosmos seemed to go slow for a few seconds, and I was charged more money for going over the 800 RUs I paid for. Try to keep the cost of your queries within the allocated provision if you don't want subsequent queries to be slow.

I should say that asking Cosmos DB to extract all the `TagIds` from all 500,000 `CosmosBooks` and returning the 35 distinct `TagIds` isn't a good design, but it did uncover what happens if you exceed your allotted RUs.

COULDN'T FILTER BY TAGIDS BECAUSE EF CORE 5 COSMOS DOESN'T SUPPORT THE IN COMMAND

The last issue to overcome is filtering the books by their `TagIds` because EF Core 5's Cosmos DB database provider doesn't support the `IN` command. Although you could use a direct Cosmos SQL command, EF Core 5 supports the LINQ `Contains` method for strings.

EF CORE 5 LIMITATION EF Core 5 doesn't support the Cosmos equivalent of the SQL `IN` command to filter on the `Tags` collection. The LINQ query `Books.Where(x => x.Tags.Any(y => y == "some tag name"))` would throw a `could not be translated` exception. I get around this problem by using string `Contains`.

By adding a string called `TagsString` and putting in each `TagId`, plus extra delimiting characters, we can use string `Contains` to filter by `TagIds`. The following code snippet shows the `TagsString` key/value taken from the `CosmosBook` JSON in listing 16.10:

```
"TagsString": "| Databases | Microsoft & .NET |"
```

> **NOTE** The delimiting | character in `TagsString` ensures that the filter-by-tag feature matches the whole `TagId` string; otherwise, the `Tag` `"Data"` would match `"Data"` and `"Databases"`.

This technique makes filtering by `Tag` easy. To select all the `Books` with `Tag` `"C#"`, for example, you would write

```
context.Books
    .Where(x => x.TagsString.Contains("| C# |"))
    .ToListAsync();
```

This approach is quite acceptable in Cosmos DB, which has a page about `Contains` and strings. In fact, the `Contains` string method is faster than the `IN`/subquery method. For 500,000 books, the string `Contains` took ~125 ms, whereas the `JOIN`/`WHERE` version had a large variation in timings, up to 3 seconds.

16.7.3 How difficult would it be to use this two-database CQRS design in your application?

There is no doubt that the Cosmos DB version provides an excellent performance for the Book App when the numbers of `Books` and `Reviews` increase. But how hard would it be to add this approach to an existing application, and would doing that have a negative effect on further development of the application? I added this CQRS design to the existing Book App, so I am in a good position to answer these questions.

On reflection, most of the time was taken up by understanding how the Cosmos DB worked and adjusting things to fit its style. Looking at the GitHub commits, it took me about two weeks to add the two-database CQRS enhancement to the existing Book App, but that time included a lot of research and building the extra Cosmos (Direct) version. As I said earlier, I think that the two-database CQRS design was a bit easier to build and test than the SQL (+cache) version.

> **NOTE** The two-database CQRS enhancement was implemented as an additional query approach while leaving all the original book-display systems; also, I altered the SQL (+cache) and SQL (Dapper) code to have a no-count version. Building all these versions allowed me to compare the performance of the two-database CQRS system with the original SQL book-display systems.

Here is a breakdown of the parts, with my views on how difficult they were:

- *Detecting changes to an SQL* `Book`—This part was made easy by the use of DDD classes, as I could add an event to each access method in the `Book` entity class. If you aren't using DDD classes, you would need to detect changes to entities during `SaveChangesAsync`, but as I say in section 16.4, that approach is harder.

- *Running the event code within a transaction*—My `GenericEventRunner` library made this part significantly quicker to write. You don't need to use this library, but it would take longer to develop.
- *Writing to the Cosmos DB database*—That part was fairly easy, with some straightforward `Add`, `Update`, and `Delete` methods. (See listing 16.8 for an example.) I spent some time making the write more robust by handing possible causes by concurrent updates.
- *Querying the Cosmos DB database*—This part took the most time, mainly because there are limitations in EF Core and in Cosmos DB.

When it came to the effects of adding the CQRS design to the existing Book App, I would say that the Cosmos DB part had little effect on the Book App's structure. Here are the changes I needed to make to the existing code:

- Registering the Cosmos DbContext on startup
- Adding integration events to the `Book` entity class
- Altering the SQL (+cache) and SQL (Dapper) code to have no-count versions

All the existing code still works the same way that it always did. Clearly, changes to the `Book` entity could require changes to the `CosmosBook` entity and its associated `MapBookToCosmosBook` extension method. Except for changes to the `Book` entity, a change to the SQL code should have no effect on the Cosmos DB code, and a change to the Cosmos DB code should have no effect on the SQL code of the application.

16.8 Differences in other database types

Most of this chapter is about Cosmos DB, which is different from the relational databases that this book covers. But at the end of this chapter, we look at relational databases again. Different types of relational databases are similar, mainly because there is an official standard for the SQL language, but many small differences exist. This section is useful if you want to move from one relational database to another, such as SQL Server to PostgreSQL.

EF Core will handle many of the differences between relational database types, such as how table names should be wrapped in the SQL commands, but you have to handle some things yourself, such as the different formats of UDFs (see section 10.1). Here is a list of typical things to check and change if you are moving from one relational database to another:

1. Download the NuGet database provider, and change the registration of your DbContext.

   The first thing you need to do is install the specific EF Core database provider via NuGet, such as `Microsoft.EntityFrameworkCore.SqlServer` or `Npgsql.EntityFrameworkCore.PostgreSQL`. Then you need to change the way you register that database provider to your DbContext. In ASP.NET Core, you would have something like this for a MySQL database provider:

```
services.AddDbContext<MyDbContext>(
    options => options.UseMySql(connection));
```

2. Rerun the `Add-Migration` command for the new database provider.

   EF Core migrations are database-provider-specific and are *not* transferrable between databases. You need to throw away your old migrations and run the `Add-Migration` command, using your new database provider.

   NOTE You can have migrations for multiple database types as long as you keep them in different projects. You must add the `MigrationsAssembly` method to the registration of each DbContext to tell EF Core where the migrations are located.

3. Fix any type mapping between .NET and the database that has changed.

   You need to rerun your LINQ queries and see whether anything has changed. In the first edition of this book, I converted the Book App from SQL Server to MySQL, and the main `Select` book display query (see listing 2.12) threw an exception. It turns out that the returned type of the SQL `AVG` command in MySQL is a nullable `decimal` rather than the nullable `double` in SQL Server. To overcome this problem, you need to change the `BookListDto`'s `AverageReviewVotes` property .NET type to `decimal?` to match the way MySQL works.

   Other, subtler type differences exist between database servers that might go unnoticed. Typical things to look at are

   a. *Concurrency timestamp types*—In SQL Server, it's a `byte[]` type; in PostgreSQL, you use a `uint` type (and you need to configure it when your register your DbContext); and MySQL uses a `DateTime` type, so check that you have the correct type for your database type.
   b. *String queries and collation* (see section 2.8.3)—By default, SQL Server and MySQL use a case-insensitive match between strings, and PostgreSQL is by default case-sensitive. Setting a `Collation` on the database, table, or column has different names and effects.
   c. `DateTime` *precision*—Most databases have moved to `DateTime2`, with its time precision at 100 ns, but it's worth checking. SQLite stores `DateTime` as a string, using ISO8601 format: `"YYYY-MM-DD HH:MM:SS.SSS"`.

4. Check and change any raw SQL that you are using.

   This step is where things get more complex, because EF Core isn't covering any changes in the way that the database type uses SQL. Standard SQL code should work, but the way of referring to tables and columns might change. More-complex SQL such as UDFs and stored procedures seem to have slightly different formats between database types.

Summary

- A NoSQL database is designed to be high-performance in terms of speed, scalability, and availability. It achieves this performance by dropping relational-database features such as strongly linked relationships between tables.
- A CQRS architecture separates the read operations from the write operations, which allows you to improve the read side's performance by storing the data in a form that matches the query, known as a projection.
- The Book App has been augmented by the ability to store a projection of the SQL `Book` on the read side of the CQRS architecture, which uses a Cosmos DB database. This approach improves performance, especially with lots of entries.
- The design used to implement the SQL/Cosmos DB CQRS architecture uses an integration event (see chapter 12).
- The Cosmos DB database works differently from relational databases, and the process of adding this database to the Book App exposes many of these differences.
- The EF Core 5 Cosmos DB database provider has many limitations, which are discussed and overcome in this chapter. But it is still possible to implement a useful app with Cosmos DB.
- The updated Book App shows that the Cosmos DB database can provide superior read performance over a similarly priced SQL Server database.
- The SQL/Cosmos DB CQRS design is suitable for adding to an existing application where read-side performance needs a boost, but it does add a time cost to every addition or update of data.
- Relational databases are more like one another than they are like NoSQL databases, due to the standardization of the SQL language. But you need to make some changes and checks if you change from one type of relational database to another.

17

Unit testing EF Core applications

This chapter covers

- Simulating a database for unit testing
- Using the database type as your production app for unit testing
- Using an SQLite in-memory database for unit testing
- Solving the problem of one database access breaking another part of your test
- Capturing logging information while unit testing

This chapter is about unit testing applications that use EF Core for database access. You'll learn what unit testing approaches are available for working with EF Core and how to choose the correct tools for your specific needs. I also describe numerous methods and techniques to make your unit testing both comprehensive and efficient. Personally, I think unit testing is useful, and I use it a lot. It makes me a better developer because I can catch bugs both when I develop the code and, more important, when I refactor the code.

But although I really like unit testing, I'm also aware that writing unit tests takes development effort, including refactoring unit tests as the application grows. Over the years, I have learned a lot of tips and techniques for unit testing, and I have built a library called EfCore.TestSupport to help me, and you, write unit tests quickly and efficiently.

Unit testing is a big subject, with whole books dedicated to the topic. I focus on the narrow but important area of unit testing applications that use EF Core for database accesses. To make this chapter focused, I don't explain the basics of unit testing, but leap right in. Therefore, I recommend skipping this chapter if you're new to unit testing and coming back to it after you've read up on the subject. This chapter won't make any sense without that background, and I don't want to discourage you from unit testing because I make it look too hard.

> **MORE INFO** For an introduction to unit testing in .NET, try this video: http://mng.bz/K44E. For much more in-depth coverage of unit testing, I recommend Vladimir Khorikov's *Unit Testing Principles, Practices, and Patterns* (Manning, 2020; https://www.manning.com/books/unit-testing).

OK, if you're still with me, I assume that you know what unit testing is and have at least written some unit tests. I'm not going to cover the differences between unit tests and integration tests, acceptance tests, and so on. I'm also not here to persuade you that unit tests are useful; I assume that you're convinced of their usefulness and want to learn tips and techniques for unit testing an EF Core application.

> **NOTE** I call all of my tests *unit tests*, but some people use the term *integration tests* for tests that use a real database.

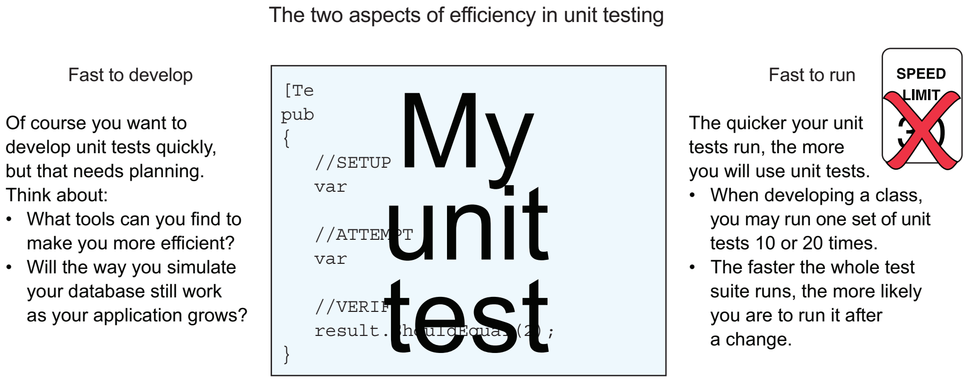
As I said, I use unit tests a lot. I have more than 700 unit tests in this book's GitHub repo, some to check that my Book App works and some to check that what I say in the book is correct. Those tests make me much more confident that what the book says is correct and that the Book App runs properly. Some of the code in the part 3 Book App is pretty complex, which is where unit tests become most useful.

> **NOTE** Arthur Vickers, who is the engineering manager of EF Core, put out a tweet to say that EF Core has more than 70,000 unit tests (using xUnit). See http://mng.bz/D18y for the facts and timings.

One other thing I have learned is that I want my unit tests to run as quickly as possible, because a quick test-debug cycle makes developing and refactoring an application a much nicer experience. Also, I'm much more likely to run all my unit tests if those tests are quick, which might catch bugs in places I didn't think would be affected by my new code. I summarize these two aspects of unit testing in figure 17.1.

The rest of the chapter starts with the basics, looks at ways you could write your unit tests, and finally presents specific tips and problems you might have when testing your EF Core code. The sections in this chapter are

- An introduction to the unit test setup
- Getting your application's DbContext ready for unit testing


The two aspects of efficiency in unit testing

Fast to develop

Of course you want to develop unit tests quickly, but that needs planning. Think about:

- What tools can you find to make you more efficient?
- Will the way you simulate your database still work as your application grows?

Fast to run

The quicker your unit tests run, the more you will use unit tests.

- When developing a class, you may run one set of unit tests 10 or 20 times.
- The faster the whole test suite runs, the more likely you are to run it after a change.

Figure 17.1 I believe wholeheartedly in unit tests, but that doesn't mean I want to spend a lot of time developing or running them. My approach is to try to be efficient at using them, which splits into developing quickly and not having to hang around while the tests run.

- Three ways to simulate the database when testing EF Core applications
  - Using a production-type database in your unit tests
  - Using an SQLite in-memory database for unit testing
  - Stubbing or mocking an EF Core database
- Unit testing a Cosmos DB database
- Seeding a database with test data to test your code correctly
- Solving the problem of one database access breaking another part of your test
- Capturing the database commands sent to a database

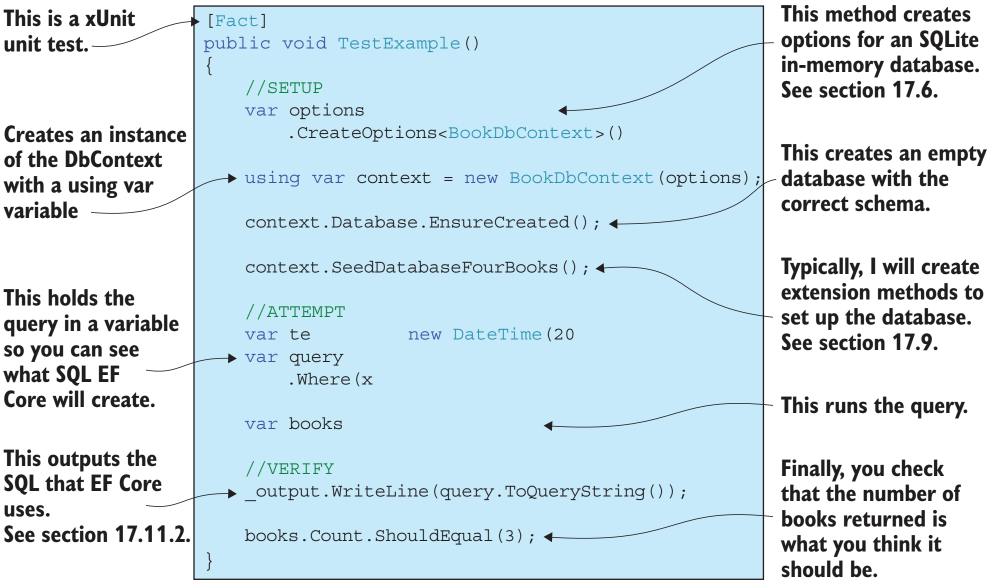
17.1 An introduction to the unit test setup

Before I start explaining the techniques, I need to introduce our unit test setup; otherwise, the examples won't make any sense. I use a fairly standard approach, but as you'll see, I've also created tools to help with the EF Core and database side of unit testing. Figure 17.2 shows a unit test that uses some of the features and methods covered in this chapter.

> **NOTE** All the unit tests in this chapter (apart from the Cosmos DB section, 17.8) use sync methods; they call `SaveChanges`, not `SaveChangesAsync`, for example. I do that partly because the code is a little bit easier to understand without the await code, but in real life, I use sync methods whenever I can because sync code provides better exception `StackTrace` results and is easier to debug when using breakpoints.

This section covers

- The test environment you'll be using: the xUnit unit test library
- A NuGet package I created to help with unit testing EF Core applications



```
[Fact]
public void TestExample()
{
    //SETUP
    var options
        .CreateOptions<BookDbContext>()

    using var context = new BookDbContext(options);

    context.Database.EnsureCreated();

    context.SeedDatabaseFourBooks();

    //ATTEMPT
    var te          new DateTime(20
    var query
        .Where(x

    var books

    //VERIFY
    _output.WriteLine(query.ToQueryString());

    books.Count.ShouldEqual(3);
}
```

This is a xUnit unit test.

This method creates options for an SQLite in-memory database. See section 17.6.

Creates an instance of the DbContext with a using var variable

This creates an empty database with the correct schema.

Typically, I will create extension methods to set up the database. See section 17.9.

This holds the query in a variable so you can see what SQL EF Core will create.

This runs the query.

This outputs the SQL that EF Core uses. See section 17.11.2.

Finally, you check that the number of books returned is what you think it should be.

Figure 17.2 A unit test with three parts: Setup, Attempt, and Verify (also known as Arrange, Act, and Assert). The figure also shows some EF Core techniques that will be explained in this chapter.

17.1.1 The test environment: xUnit unit test library

I'm using the xUnit unit test library (see https://xunit.net) because Microsoft supports it well and because the EF Core team uses it. Also, xUnit is quicker than some other unit test frameworks, such as NUnit (which I used to use), because xUnit can run unit test classes in parallel. Running tests in parallel has a downside, which I show you how to get around, but it does mean that you can run your complete unit test suite a lot quicker.

I also use *fluent validation, which uses a series of extension methods that flow one after another*; see row 1 in table 17.1. I find the fluent validation style to be much easier to work with than the static `Assert` methods approach; it's slightly shorter, and Intellisense can suggest the fluent validation methods that are appropriate.

Table 17.1 Two approaches to checking that two books were loaded by the previous query that was under test. The static `Assert` methods are built into XUnit; the fluent validation style has to be added as an extra step.

Type	Example code
Fluent validation style	`books.Count().ShouldEqual(2);`
Static `Assert` method style	`Assert.Equal(2, books.Count());`

You can find these fluent validation extension methods at http://mng.bz/l2Ej, but you can create your own; they are normal C# extension methods. I've included the sample xUnit fluent validation extension methods, plus a few extra fluent validations, in the NuGet package called EfCore.TestSupport that I've built. See section 17.1.2.

The following listing shows a simple unit test that employs the xUnit unit test package and the fluent validation extensions. This example uses a three-stage pattern of Setup, Attempt, and Verify, shown as `//SETUP`, `//ATTEMPT`, and `//VERIFY` in the unit test code in this chapter. Setup, Attempt, and Verify are also known as Arrange, Act, and Assert, but because I'm dyslexic, I prefer `//SETUP`, `//ATTEMPT`, and `//VERIFY` in my code because they look quite different.

Listing 17.1 A simple example xUnit unit test method

```
[Fact]
public void DemoTest()
{
    //SETUP
    const int someValue = 1;

    //ATTEMPT
    var result = someValue * 2;

    //VERIFY
    result.ShouldEqual(2);
}
```

The [Fact] attribute tells the unit test runner that this method is an xUnit unit test that should be run.

The method must be public. It should return void or, if you're running async methods, a Task.

This line is where you run the code you want to test.

Typically, you put code here that sets up the data and/or environment for the unit test.

Here is where you put the test(s) to check that the result of your test is correct.

You can run your unit tests by using Visual Studio's built-in Test Explorer, which you access from the Test menu. If you're using Visual Studio Code (VS Code), the test runner is also built in, but you need to set up the `build` and `test` tasks in the VS Code tasks.json file, which allows you to run all the tests via the Task > Test command.

17.1.2 A library I created to help with unit testing EF Core applications

I learned a lot about unit testing EF Core applications as I built the software that went with the first edition of this book. As a result, I built an open source library called EfCore.TestSupport (see https://github.com/JonPSmith/EfCore.TestSupport) that contains lots of methods that are useful in the Setup stage of a unit test method.

The EfCore.TestSupport library differentiate between EF Core 2 and EF Core 3, using the netstandard they used, but now that EF Core 5 is out, that differentiation doesn't work anymore. Therefore, I aligned the EfCore.TestSupport library to EF Core via the first part of the version number. For EF Core 5, for example, you need EfCore.TestSupport version 5.

> **NOTE** Readers who are already using my EfCore.TestSupport library should be aware that I also took the opportunity to tidy up the EfCore.TestSupport library, which introduced breaking changes. The SqliteInMemory has changes (see section 17.6), some methods are now obsolete, and I moved the EfSchemaCompare code to another library. See http://mng.bz/BK5v for more details.

This chapter uses many of the methods in the EfCore.TestSupport library, but I don't detail their signatures because the EfCore.TestSupport wiki (see http://mng.bz/dmND) contains documentation for this library. But I'll explain the how and why of unit testing by using some of the methods from my EfCore.TestSupport library and showing some of the code I developed too.

17.2 Getting your application's DbContext ready for unit testing

Before you can unit test your application's DbContext with a database, you need to ensure that you can alter the database connection string. Otherwise, you can't provide a different database(s) for unit testing. The technique you use depends on how the application's DbContext expects the options to be set. The two approaches that EF Core provides for setting the options are as follows:

- The application's DbContext expects the options to be provided via its constructor. This approach is recommended for ASP.NET Core and .NET Generic Host applications.
- The application's DbContext sets the options internally in the `OnConfiguring` method. This approach is recommended for applications that don't use dependency injection.

17.2.1 The application's DbContext options are provided via its constructor

If the options are provided via the application's DbContext constructor, you don't need any changes to the application's DbContext to work with the unit test. You already have total control of the options given to the application's DbContext constructor; you can change the database connection string, the type of database provider it uses, and so on. The following listing shows the format of an application's DbContext that uses a constructor to obtain its options. The constructor is shown in bold.

Listing 17.2 An application DbContext that uses a constructor for option setting

```
public class EfCoreContext : DbContext
{
    public EfCoreContext(
        DbContextOptions<EfCoreContext> options)
        : base(options) {}

    public DbSet<Book> Books { get; set; }
    public DbSet<Author> Authors { get; set; }

    //… rest of the class left out
}
```

For this type of application's DbContext, the unit test can create the options variable and provide that value as a parameter in the application's DbContext constructor. The next listing shows an example of creating an instance of your application's DbContext in a unit test that will access an SQL Server database, with a specific connection string.

Listing 17.3 Creating a DbContext by providing the options via a constructor

```
const string connectionString
    = "Server= … content removed as too long to show";
var builder = new
    DbContextOptionsBuilder<EfCoreContext>();
builder.UseSqlServer(connectionString);
var options = builder.Options;
using (var context = new EfCoreContext(options))
{
     //… unit test starts here
```

Defines that you want to use the SQL Server database provider

Holds the connection string for the SQL Server database

You need to create the DbContextOptionsBuilder<T> class to build the options.

Builds the final DbContextOptions<EfCoreContext> options that the application's DbContext needs

Allows you to create an instance for your unit tests

17.2.2 Setting an application's DbContext options via OnConfiguring

If the database options are set in the `OnConfiguring` method inside the application's DbContext, you must modify your application's DbContext before you can use it in unit testing. But before you change the application's DbContext, I want to show you the normal arrangement of using the `OnConfiguring` method to set the options (see the bold text in the following listing).

Listing 17.4 A DbContext that uses the `OnConfiguring` method to set options

```
public class DbContextOnConfiguring : DbContext
{
    private const string connectionString
        = "Server=(localdb)\\... shortened to fit";

    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
        base.OnConfiguring(optionsBuilder);
    }
    // … other code removed
}
```

The next listing shows Microsoft's recommended way to change a DbContext that uses the `OnConfiguring` method to set up the options. As you'll see, this technique adds the same sort of constructor setup that ASP.NET Core uses while making sure that the `OnConfiguring` method still works in the normal application.

Listing 17.5 An altered DbContext allowing the connection string to be set by the unit test

```
public class DbContextOnConfiguring : DbContext
{
    private const string ConnectionString
        = "Server=(localdb)\\ … shortened to fit";

    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
            optionsBuilder
                .UseSqlServer(ConnectionString);
        }
    }

    public DbContextOnConfiguring(
        DbContextOptions<DbContextOnConfiguring>
        options)
        : base(options) { }

    public DbContextOnConfiguring() { }
    // … other code removed
}
```

Changes the OnConfigured method to run its normal setup code only if the options aren't already configured

Adds the same constructor-based options settings that the ASP.NET Core version has, which allows you to set any options you want

Adds a public, parameterless constructor so that this DbContext works normally with the application

To use this modified form, you can provide options in the same way you did with the ASP.NET Core version, as shown in the following listing.

Listing 17.6 A unit test providing a different connection string to the DbContext

```
const string connectionString
    = "Server=(localdb)\\... shortened to fit";
var builder = new
    DbContextOptionsBuilder
        <DbContextOnConfiguring>();
builder.UseSqlServer(connectionString);
var options = builder.Options;
using (var context = new
    DbContextOnConfiguring(options)
{
    //… unit test starts here
```

Holds the connection string for the database to be used for the unit test

Sets up the options you want to use

Provides the options to the DbContext via a new, one-parameter constructor

Now you're good to go for unit testing.

17.3 Three ways to simulate the database when testing EF Core applications

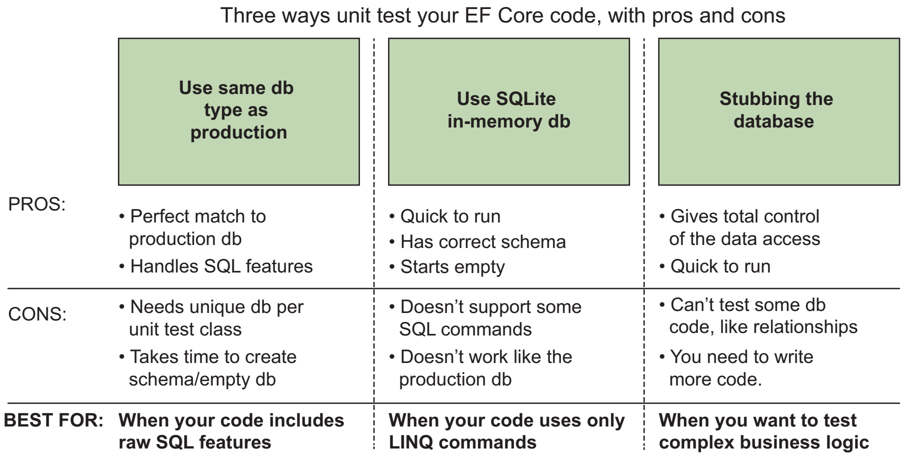
If you unit test your application, and it includes accesses to a database, you have several ways to simulate the database. Over the years, I've tried several approaches to simulating the database in a unit test, ranging from a library that mocks the DbContext in

EF6 called Effort (see https://entityframework-effort.net/overview) to using real databases. This chapter covers some of those approaches and a few new tactics that EF Core offers.

NOTE I cover stubbing and mocking in much more detail in section 17.7.

Early consideration of how to unit test with a database can save you a lot of pain later, especially if you're using EF Core. When I started writing the first edition of this book, I found that the unit testing approach I used at the start didn't work with the more SQL-based parts of the book, so I had to refactor some of my early unit tests, and that was a bit of a pain.

But that experience wasn't new. In some of my projects, I later regretted my early decisions on unit testing, as the tests started to fall apart as the projects grew. Although some reworking of early unit tests is inevitable, you want to minimize rework, because it slows you down. Therefore, I want to describe different ways to unit test code with EF Core so that you can make an informed decision about how to write your unit tests. Figure 17.3 summarizes the three main ways you can test code that contains database accesses.



Three ways unit test your EF Core code, with pros and cons

	Use same db type as production	**Use SQLite in-memory db**	**Stubbing the database**
PROS:	• Perfect match to production db • Handles SQL features	• Quick to run • Has correct schema • Starts empty	• Gives total control of the data access • Quick to run
CONS:	• Needs unique db per unit test class • Takes time to create schema/empty db	• Doesn't support some SQL commands • Doesn't work like the production db	• Can't test some db code, like relationships • You need to write more code.
BEST FOR:	**When your code includes raw SQL features**	**When your code uses only LINQ commands**	**When you want to test complex business logic**

Figure 17.3 You have three main ways to provide access to a database when you are testing your code. Each approach has pros and cons, and the main ones are listed in the figure.

DEFINITION The term *production database* refers to the database type/provider used by your application in production. If you are running an ASP.NET Core web application using EF Core, for example, and that application uses an SQL Server database, a production database type is SQL Server. In that case, using the same database type as production means that SQL Server databases will be used in your unit tests.

There is no right answer as to which approach is best for you—only a series of tradeoffs between your unit tests running in the same way as your production application and the time to write and run your unit tests. The safe solution is to use a database that is the same as your production database type. But I often use a mixture of all three of these approaches when I am unit testing some applications.

Before I describe the three approaches to simulating the database, section 17.4 digs deeper into the differences between the first two approaches. This section gives you more information to help you decide whether you can test your application with the SQLite in-memory database or need to use unit test databases of the same type as your production database.

17.4 Choosing between a production-type database and an SQLite in-memory database

In this section, I give you the information you need to decide whether to use a production-type database or an SQLite in-memory database. You should consider using an SQLite in-memory database because it is easier for unit testing, creating a new database every time. As a result

- The database schema is always up to date.
- The database is empty, which is a good starting point for a unit test.
- Running your unit tests in parallel works because each database is held locally in each test.
- Your unit tests will run successfully in the Test part of a DevOps pipeline without any other settings.
- Your unit tests are faster.

The downside is that the SQLite database doesn't support and/or match some SQL commands in your production database, so your unit tests will fail or, in a few cases, give you incorrect results. If this possibility worries you, you should ignore SQLite and use the same database type as your production database for unit testing (see section 17.5).

If you want to consider using SQLite for unit testing, you need to know how different it might be from your production database. The simple answer is "a lot," but to help you understand what might cause problems, I've prepared table 17.2. This table lists the features that could cause problems when you use SQLite for unit testing. The far-right column lists the possible outcome of using the feature:

- *Wrong answer*—The feature might work but give you the wrong answer (which, in unit testing, is the worst result). You must be careful to run the test with a production-type database or make sure that you understand the limitations and work around them.
- *Might break*—The feature might work correctly in your unit test code, but in some cases, it might throw an exception. You can test this feature with SQLite, but you might have to change to a production-type database if a unit test fails.

- *Will break*—The feature is likely to fail when the database is set up (but might work if the SQL is basic). This result rules out using an SQLite in-memory database.

Table 17.2 The SQL features that EF Core can control but that aren't going to work with SQLite, because SQLite doesn't support the feature or because SQLite uses a different format from SQL Server, MySQL, and so on

SQL feature	See section	SQLite support?	Breaks?
String compare and collations	2.8.3	Works but provides different results	Wrong answer
Different schemas	7.12.2	Not supported; ignores config	Wrong answer
SQL column default value	10.3	C# constants work; SQL is likely to fail	Might break
SQL computed columns	10.2	SQL is different; likely to fail	Will break
Any raw SQL	11.5	SQL is different; very likely to fail	Will break
SQL sequences	10.4	Not supported exception	Will break

Also, the following C# types aren't natively supported by SQLite, so they could produce the wrong value:

- `Decimal`
- `UInt64`
- `DateTimeOffset`
- `TimeSpan`

EF Core will throw an exception if you sort/filter on a property that is of type `Decimal` while running on SQLite, for example. If you still want to unit test with SQLite, you can add a value converter to convert the `Decimal` to a `double` (see section 7.13), but that approach might not return the exact `Decimal` value you saved to the database.

So if you use any of the features in table 17.2 that will break, you definitely don't want to use SQLite for unit testing. But you also need to consider what you plan to add to your application, because if you add code that uses "will break" features, you are going to have to change all your unit tests to use a database of the same type as your production database, which can be a real pain.

If you're not using, and are unlikely to use, the "will break" features shown in table 17.2, SQLite could be a good choice for most of your unit tests. You can switch to using a production-type test database for the "might break" features, which is what I do for EF Core applications that don't use much raw SQL features.

> **NOTE** I haven't yet covered the pros and cons of the third option in figure 17.3: stubbing the database (see section 17.7). Stubbing the database is a different approach from using SQLite or a production-type database because stubbing tries to remove all the database code from the unit test. For that reason,

stubbing the database doesn't test any of your EF Core code. Therefore, I start with the two approaches that do include EF Core: using an SQLite in-memory database and using a production-type database.

17.5 Using a production-type database in your unit tests

This section covers using a production-type database for your unit testing, which is the best way to unit test because your unit test databases are fully compatible with your production database. The downside is that the database is more complex to set up than in the SQLite in-memory database approach (see section 17.6), as well as slightly slower to run. You need to solve four issues to use a production-type database in your unit tests:

- Providing a connection string to the database to use for the unit test
- Providing a database per test class to allow xUnit to run tests in parallel
- Making sure that the database's schema is up to date and the database is empty
- Mimicking the database setup that the EF Core's migrations would deliver

Interestingly, the SQLite in-memory approach overcomes the first three items in the list by its design alone, and the last item, which deals with SQL embedded in your migrations, is something that the SQLite in-memory approach can't handle because the SQL code is likely to be different. The list of four issues you must meet to run a unit test is a good indication of the extra work involved in finding the best way to unit test code that includes database accesses. But help is on hand via my EfCore.TestSupport library, which provides extension methods that help you set up the database options, deal with the "database per test class" problem, and make sure that the database schema is up to date and empty of data.

NOTE The following examples use an SQL Server database, but the approaches work equally well with database types other than Cosmos DB, which has its own section (17.8).

17.5.1 Providing a connection string to the database to use for the unit test

To access any database, you need a connection string (see section 5.4.1). You could define a connection string as a constant and use that, but as you'll see, that approach isn't as flexible as you'd want. Therefore, in this section you'll mimic what ASP.NET Core does by adding to your test project a simple appsettings.json file that holds the connection string. Then you'll use some of the .NET configuration packages to access the connection string in your application. The appsettings.json file looks something like this:

```
{
  "ConnectionStrings": {
    "UnitTestConnection": "Server=(localdb)\\mssqllocaldb;Database=... etc"
  }
}
```

WARNING You should not put a connection string that contain private keys, passwords, and so on in the appsetting.json file, as those elements may leak when you store your code in source control. .NET has a feature called *user secrets* that is built into ASP.NET Core (see http://mng.bz/rmYg), and you can use user secrets in your unit tests by using the `AddUserSecrets` method.

Listing 17.7 shows the `GetConfiguration` method from my EfCore.TestSupport library. This method loads an appsettings.json file from the top-level directory of the assembly that calls it, which would be the assembly in which you're running your unit tests.

Listing 17.7 `GetConfiguration` method allowing access to the appsettings.json file

Returns IConfigurationRoot, from which you can use methods such as GetConnectionString("ConnectionName") to access the configuration information

In the TestSupport library, a method returns the absolute path of the calling assembly's top-level directory (the assembly that you're running your tests in).

```
public static IConfigurationRoot GetConfiguration()
{
    var callingProjectPath =
        TestData.GetCallingAssemblyTopLevelDir();
    var builder = new ConfigurationBuilder()
        .SetBasePath(callingProjectPath)
        .AddJsonFile("appsettings.json", optional: true);
    return builder.Build();
}
```

Uses ASP.NET Core's ConfigurationBuilder to read that appsettings.json file. It's optional, so no error is thrown if the configuration file doesn't exist.

Calls the Build method, which returns the IConfigurationRoot type

You can use the `GetConfigration` method to access the connection string and then use this code to create an application's DbContext:

```
var config = AppSettings.GetConfiguration();
config.GetConnectionString("UnitTestConnection");
var builder = new DbContextOptionsBuilder<EfCoreContext>();
builder.UseSqlServer(connectionString);
using var context = new EfCoreContext(builder.Options);
// … rest of unit test left out
```

That code solves the problem of getting a connection string, but you still have the problem of having different databases for each test class because by default, xUnit runs unit tests in parallel. This topic is covered in section 17.5.2.

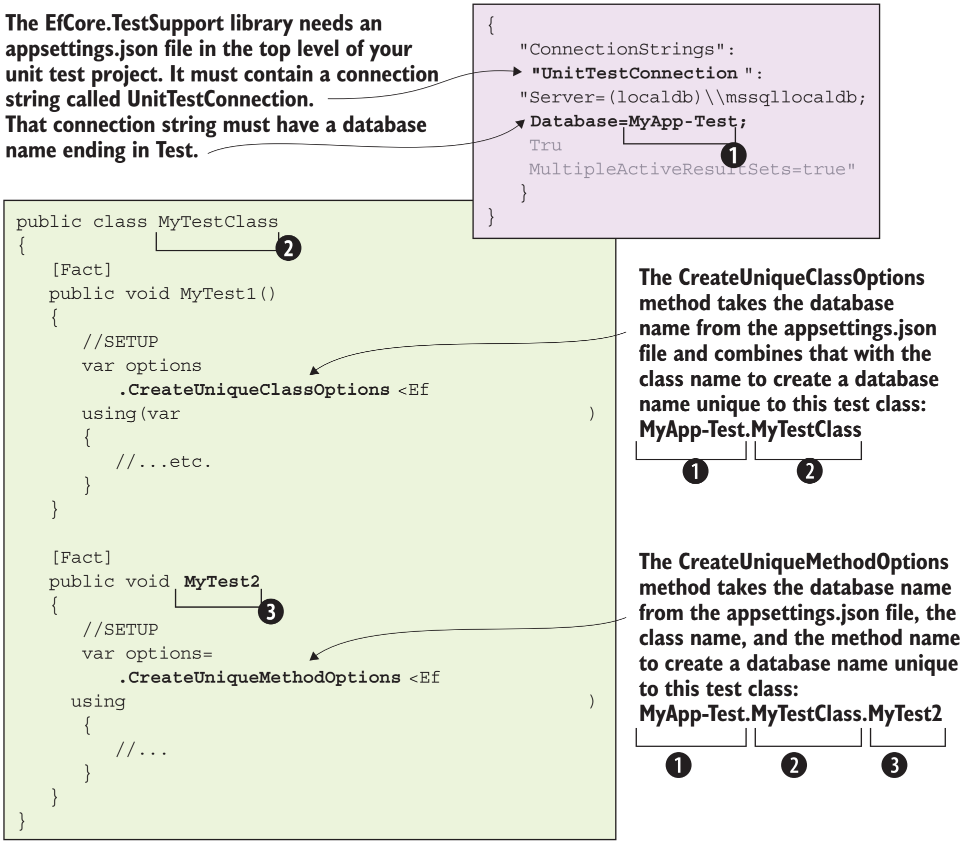
17.5.2 Providing a database per test class to allow xUnit to run tests in parallel

Because xUnit can run each class of unit tests in parallel, using one database for all your tests wouldn't work. Good unit tests need a known starting point and should return a known result, which rules out using one database, as different tests will simultaneously change the database.

NOTE You can run xUnit sequentially (see the "Changing Default Behavior" section of this xUnit documentation at https://xunit.net/docs/running-tests-in-parallel), but I don't recommend doing that because it will slow the running of your unit tests.

One common solution is to have separately named databases for each unit test class or possibly each unit test method. The EfCore.TestSupport library contains methods that produce an SQL Server `DbContextOptions<T>` result in which the database name is unique to a test class or method. Figure 17.4 shows the two methods. The first method creates a database with a name unique to this class, and the second one produces a database with a name that's unique to that class and method.

The result of using either of these classes is that each test class or method has its own uniquely named database. So when unit tests are run in parallel, each test class has its own database to test against.



The EfCore.TestSupport library needs an appsettings.json file in the top level of your unit test project. It must contain a connection string called UnitTestConnection.
That connection string must have a database name ending in Test.

```
{
   "ConnectionStrings":
    "UnitTestConnection ":
   "Server=(localdb)\\mssqllocaldb;
    Database=MyApp-Test;
    Tru
    MultipleActiveResultSets=true"
   }
}
```

```
public class MyTestClass
{
   [Fact]
   public void MyTest1()
   {
      //SETUP
      var options
         .CreateUniqueClassOptions <Ef
      using(var                                      )
      {
         //...etc.
      }
   }

   [Fact]
   public void MyTest2
   {
      //SETUP
      var options=
         .CreateUniqueMethodOptions <Ef
     using                                           )
      {
         //...
      }
   }
}
```

The CreateUniqueClassOptions method takes the database name from the appsettings.json file and combines that with the class name to create a database name unique to this test class: MyApp-Test.MyTestClass

The CreateUniqueMethodOptions method takes the database name from the appsettings.json file, the class name, and the method name to create a database name unique to this test class: MyApp-Test.MyTestClass.MyTest2

Figure 17.4 Two methods that set up the database options for an SQL Server database but alter the database name to be class-unique or class-and-method-unique. When you run multiple unit test classes, they have their own databases, so they won't interfere with each other.

TIP xUnit runs each test class in parallel; but within a class, it runs each test serially. For this reason, I normally use a class-unique database. I use a class-and-method-unique database rarely, but it's there if I need it.

The next listing shows the code inside the `CreateUniqueClassOptions` extension method. This code encapsulates all the settings of the DbContext options to save you from having to include them in every unit test.

Listing 17.8 `CreateUniqueClassOptions` extension method with a helper

```
public static DbContextOptions<T>
    CreateUniqueClassOptions<T>(
        this object callingClass,
        Action<DbContextOptionsBuilder<T>> builder = null)
    where T : DbContext
{
    return CreateOptionWithDatabaseName<T>
        (callingClass, builder);
}

private static DbContextOptions<T>
    CreateOptionWithDatabaseName<T>(
        object callingClass,
        Action<DbContextOptionsBuilder<T>> extraOptions,
        string callingMember = null)
    where T : DbContext
{
    var connectionString = callingClass
        .GetUniqueDatabaseConnectionString(callingMember);
    var builder = new DbContextOptionsBuilder<T>();
    builder.UseSqlServer(connectionString);
    builder.ApplyOtherOptionSettings();
    extraOptions?.Invoke(builder);
    return builder.Options;
}
```

It's expected that the object instance provided will be this—the class in which the unit test is running.

Returns options for an SQL Server database with a name starting with the database name in the original connection string in the appsettings.json file, but with the name of the class of the instance provided in the first parameter

This parameter allows you to add more option methods while building the options.

Calls a private method shared between this method and the CreateUnique-MethodOptions options

Builds the SQL Server part of the options, with the correct database name

These parameters are passed from Create-UniqueClassOptions. For CreateUniqueClassOptions, the calling method is left as null.

Returns the connection string from the appsetting.json file, but with the database name modified to use the calling-Class's type name as a suffix

Sets up OptionsBuilder and creates an SQL Server database provider with the connection string

Calls a general method used on all your option builders, enabling sensitive logging and better error messages

Applies any extra option methods that the caller provided

Returns the DbContextOptions<T> to configure the application's DbContext

xUnit's parallel-running feature has some other constraints. The use of static variables (static constants are fine) to carry information causes problems, for example, as different tests may set a static variable to different values in parallel. Nowadays, we don't use statics much because dependency injection fills that gap. But if you use static variables in your code, you should turn off parallel running in xUnit so that you run unit tests serially.

17.5.3 Making sure that the database's schema is up to date and the database is empty

Section 17.5.2 shows how to create unique databases for your tests, but you still have the problem of making sure that a database's schema is up to date and empty when you rerun a test. When I say that the "database's schema is up to date," I mean that the database's schema matches the database model that EF Core creates by scanning your entity classes and any EF Core configuration code you have applied to your application's DbContext.

Unlike your application, which will use some form of migration to update the entity classes or EF Core configuration, your unit tests will use EF Core's `EnsureCreate/EnsureCreatedAsync` methods to make sure that you have a database to work with. These methods create a database with an up-to-date schema using the current entity classes and configuration of your application's DbContext, but only if there isn't an existing database of the same name. After the first run of the unit test, the database's schema is fixed, so its schema will be out of date if you change the EF Core configuration or any of the entity classes. Therefore, you need a way to make sure that the database has an up-to-date schema and, at the same time, provide an empty database as a starting point for a unit test.

Let's start with a foolproof but slow method. Listing 17.9 shows Microsoft's recommended way to create an empty database with the correct schema without using migrations. These two EF Core methods delete and create databases; the create-database method builds the database schema up by using the current EF Core configuration and entity classes. The following listing shows a call to the `EnsureDeleted` method first to delete the database and then calls `EnsureCreated` to build the correct schema with no data in it.

Listing 17.9 The foolproof way to create a database that's up to date and empty

```
[Fact]
public void TestExampleSqlDatabaseOk()
{
    //SETUP
    var options = this
        .CreateUniqueClassOptions<EfCoreContext>();
    using (var context = new EfCoreContext(options))
    {
        context.Database.EnsureDeleted();
        context.Database.EnsureCreated();
        //… rest of test removed
```

Deletes the current database (if present)

Creates a new database, using the configuration inside your application's DbContext

Because listing 17.9 uses EfCore.testSupport's `CreateUniqueClassOptions` method, each unit test in that class uses the same database, but each unit test method deletes and re-creates the database in the Setup stage of the test.

This approach used to be slow (~10 seconds) for an SQL Server database, but since the new SqlClient library came out in .NET 5, it's been much quicker (~ 1.5 seconds), which makes a big difference in how long a unit test would take to run with the `EnsureDeleted/EnsureCreated` version.

> **NOTE** How long `EnsureDeleted/EnsureCreated` takes depends on the database. When I was writing the first edition of this book, a delete/create of an SQL Server database used to take about 10 seconds, but a MySQL database took only 1 second. You need to test your own database types to see how long it takes to delete and re-create the database.

Another approach, suggested by Arthur Vickers of the EF Core team, is a method that the team uses in its unit tests: `EnsureClean`. This clever method removes the current schema of the database by deleting all the SQL indexes, constraints, tables, sequences, UDFs, and so on in the database. Then, by default, it calls the `EnsureCreated` method to return a database that has the correct schema and is empty of data.

The `EnsureClean` method is deep inside EF Core's unit tests, but I extracted that code and built the other parts needed to make it useful; it is available in the EfCore.TestSupport version 5. The following listing shows how to use this method in your unit tests.

Listing 17.10 Using the `EnsureClean` method to update the database's schema

```
[Fact]
public void TestExampleSqlServerEnsureClean()
{
    //SETUP
    var options = this.
        CreateUniqueClassOptions<BookDbContext>();

    using var context = new BookDbContext(options);

    context.Database.EnsureClean();

    //… rest of test removed
}
```

Wipes the data and schema in the database and then calls EnsureCreated to set up the correct schema

`EnsureClean` approach is faster, maybe twice as fast as the `EnsureDeleted/EnsureCreated` version, which could make a big difference in how long your unit tests take to run. It's also better when your database server doesn't allow you to delete or create new databases but does allow you to read/write a database, such as when your test databases are on an SQL server on which you don't have admin privileges.

> **NOTE** At the moment, the `EnsuredClean` method works only for SQL Server, but the method could be improved to handle other database types. If a database type already has a quick `EnsureDeleted/EnsureCreated` run time, however, it's not worth extending.

The final approach to obtaining a database for use in a unit test is unusual but can be useful in some situations. It works by applying changes to the database only within a transaction. This approach works because when the transaction is disposed, if you haven't called the `transaction.Commit` method, it rolls back all the changes made in a database while the transaction is active. As a result, each unit test starts with the same data every time.

This approach is useful if you have an example database, maybe copied from the production database (with personal data anonymized, of course), that you want to test against, but you don't want the example database to be changed. I used this approach for a client who had an example database (1 TB in size and held in Azure). Using the transaction version allowed me to run some of the client's code to understand what it changed in the database without changing the database's content.

To use this transaction version, you must create a transaction immediately after you create the application's DbContext, and you must hold the transaction in a variable that will be disposed at the end of the unit test. In the following listing, I achieve that effect via the `using var` keywords.

Listing 17.11 Using a transaction to roll back any database changes made in the test

```
[Fact]
public void TestUsingTransactionToRollBackChanges()
{
    //SETUP
    var builder = new
        DbContextOptionsBuilder<BookDbContext>();
    builder.UseSqlServer(_connectionString);
    using var context =
        new BookDbContext(builder.Options);

    using var transaction =
        context.Database.BeginTransaction();

    //ATTEMPT
    var newBooks = BookTestData
        .CreateDummyBooks(10);
    context.AddRange(newBooks);
    context.SaveChanges();

    //VERIFY
    context.Books.Count().ShouldEqual(4+10);

}
```

You most likely will link to a database via a connection string.

The transaction is held in a user var variable, which means that it will be disposed when the current block ends.

Run your test ...

... and check whether it worked.

When the unit test method ends, the transaction will be disposed and will roll back the changes made in the unit test. In this case, four books were already in the database.

17.5.4 Mimicking the database setup that EF Core migration would deliver

One problem I came across in unit testing occurred when my database had extra SQL commands that EF Core didn't add. If you use a UDF in your code, for example, how do you get that SQL into your unit test database? You have three solutions:

- For simple SQL, such as a UDF, you can execute a script file after the `EnsureCreated` method.
- If you've added your SQL to the EF Core migration files (see section 9.5.2), you should call context.Database.Migrate instead of ….`EnsureCreated`.
- If you're using script-based migrations (see section 11.4), instead of calling `EnsureCreated`, you should execute the scripts to build the database.

The last two items have the solution detailed in the list, but the first item needs some code. I created a method called `ExecuteScriptFileInTransaction` in my EfCore .TestSupport library. This method executes the SQL inside an SQL script file on the database that the application's DbContext is connected to. The format of the script is in a Microsoft SQL Server Management Studio format: a set of SQL commands, each ending with a single line containing the SQL command `GO`. The following listing shows an SQL change script file that adds a UDF to a database.

Listing 17.12 An example SQL script file with `GO` at the end of each SQL command

Removes existing version of the UDF you want to add. If you don't do this, the create function will fail.

```
IF OBJECT_ID('dbo.AuthorsStringUdf') IS NOT NULL
     DROP FUNCTION dbo.AuthorsStringUdf
GO

CREATE FUNCTION AuthorsStringUdf (@bookId int)
RETURNS NVARCHAR(4000)
-- … SQL commands removed to make the example shorter
RETURN @Names
END
GO
```

Adds a user-defined function to the database

ExecuteScriptFileInTransaction looks for a line starting with GO to split out each SQL command to send to the database.

The `ExecuteScriptFileInTransaction` extension method can apply an SQL script to a database by using the format in listing 17.12. Listing 17.13 shows a typical way to apply this script to a unit test database.

> **NOTE** The `TestData.GetFilePath` method in the following listing is another EfCore.TestSupport library method; it allows you to access files in a top-level directory called TestData in your Test project.

Listing 17.13 An example of applying an SQL script to a unit test database

```
[Fact]
public void TestApplyScriptExampleOk()
{
    var options = this
        .CreateUniqueClassOptions<EfCoreContext>();
    var filepath = TestData.GetFilePath(
        "AddUserDefinedFunctions.sql");
    using (var context = new EfCoreContext(options))
    {
        context.Database.EnsureDeleted();
        context.Database.EnsureCreated();
```

Gets the file path of the SQL script file via your TestData's GetFilePath method

```
        context
            .ExecuteScriptFileInTransaction(
            filepath);

    //… the rest of the unit test left out
     }
}
```

Applies your script to the database by using the ExecuteScriptFileInTransaction method

17.6 Using an SQLite in-memory database for unit testing

SQLite has a useful option for creating an in-memory database. This option allows a unit test to create a new database in-memory, which means that it's isolated from any other database. This approach solves all the problems of running parallel tests, having an up-to-date schema, and ensuring that the database is empty, and it's fast. But see section 17.4 for potential problems.

To make an SQLite database in-memory, you need to set `DataSource` to `":memory:"`, as shown here. The code in listing 17.14 comes from the `SqliteInMemory.CreateOptions` method in my EfCore.TestSupport library.

NOTE The `CreateOptions` method in listing 17.14 returns a class called `DbContextOptionsDisposable<T>`. This class implements the `DbContextOptionsBuilder<T>` type needed for creating an instance of your application's DbContext, and the `IDisposable` interface, which is used to dispose the SQLite connection when the application's DbContext is disposed. I cover this topic toward the end of this section.

Listing 17.14 Creating SQLite in-memory database `DbContextOptions<T>` options

This parameter allows you to add more option methods while building the options.

A class containing the SQLite in-memory options, which is also disposable

```
public static DbContextOptionsDisposable<T> CreateOptions<T>
    (Action<DbContextOptionsBuilder<T>> builder = null)
    where T : DbContext
{
    return new DbContextOptionsDisposable<T>
       (SetupConnectionAndBuilderOptions<T>(builder)
           .Options);
}
```

Gets the DbContextOptions<T> and returns a disposable version

This method builds the SQLite in-memory options.

Contains any extra option methods the user provided

```
private static DbContextOptionsBuilder<T>
    SetupConnectionAndBuilderOptions<T>
    (Action<DbContextOptionsBuilder<T>> applyExtraOption)
    where T : DbContext
{
    var connectionStringBuilder =
        new SqliteConnectionStringBuilder
            { DataSource = ":memory:" };
    var connectionString = connectionStringBuilder.ToString();
```

Creates an SQLite connection string with the DataSource set to ":memory:"

Turns the SQLiteConnectionStringBuilder into a connection string

```
    var connection = new SqliteConnection(connectionString);
    connection.Open();

    // create in-memory context
    var builder = new DbContextOptionsBuilder<T>();
    builder.UseSqlite(connection);
    builder.ApplyOtherOptionSettings();
    applyExtraOption?.Invoke(builder);

    return builder;
}
```

Forms an SQLite connection by using the connection string

You must open the SQLite connection. If you don't, the in-memory database won't work.

Builds a DbContextOptions<T> with the SQLite database provider and the open connection

Calls a general method used on all your option builders, enabling sensitive logging and better error messages

Adds any extra options the user added

Returns the DbContextOptions<T> to use in the creation of your application's DbContext

Then you can use the `SQLiteInMemory.CreateOptions` method in one of your unit tests, as shown in the next listing. You should note that in this case, you need to call only the `EnsureCreated` method, because no database currently exists.

Listing 17.15 Using an SQLite in-memory database in an xUnit unit test

```
[Fact]
public void TestSQLiteOk()
{
    //SETUP
    var options = SQLiteInMemory
        .CreateOptions<EfCoreContext>();

    using var context = new BookDbContext(options);

    context.Database.EnsureCreated();

    //ATTEMPT
    context.SeedDatabaseFourBooks();

    //VERIFY
    context.Books.Count().ShouldEqual(4);
}
```

The SQLiteInMemory.CreateOptions provides the options for an in-memory database. The options are also IDisposable.

Uses that option to create your application's DbContext

You call context.Database .EnsureCreated to create the database.

Runs a test method you've written that adds four test books to the database

Checks that your SeedDatabaseFourBooks worked and adds four books to the database

At the end of the unit test, the `context` is disposed because you used a `using var` statement to hold the application's DbContext instance. Disposing the `context` in turn disposes the options variable, which deletes the database by disposing the `SqliteConnection` connection. Disposing the `SqliteConnection` connection follows the recommended practice in the EF Core documentation; see http://mng.bz/VG7X.

> **NOTE** If you are using multiple instances of the application's DbContext, you need to postpone disposing the `SqliteConnection` connection by using the `options.StopNextDispose` or `options.TurnOffDispose` method (see section 17.10.2 for one way).

What about EF Core's in-memory database provider for unit testing?

EF Core has an in-memory database provider that the team uses in its testing, but the documentation states that this database is "not suitable for testing applications that use EF Core" (http://mng.bz/xGO8). Therefore, the team was surprised to get feedback that lots of people are using the in-memory database provider for unit testing.

When I wrote the first edition of this book, I used the in-memory database provider and quickly found its limitations. For one thing, it doesn't work like a real relational database; therefore, it doesn't catch all the problems. When I found that SQLite had an in-memory mode, I swapped over to that database. It's not perfect, but it's *much* better than the EF Core in-memory database provider.

17.7 Stubbing or mocking an EF Core database

Moving away from using an actual database, let's look at the third approach depicted in figure 17.3: stubbing or mocking the database. Here are the definitions of the two approaches:

- *Stubbing* a database means creating some code that replaces the current database. Stubbing works well when you are using a repository pattern (see section 13.5.1).
- *Mocking* usually requires a mocking library such as Moq (see https://github.com/moq/moq4), which you use to take control of the class you are mocking. This task is basically impossible for EF Core; the closest library to mocking EF Core is EF Core's in-memory database provider.

NOTE This article provides more information on stubbing and mocking: http://mng.bz/A1Wp.

Having said that mocking isn't going to work, now I'll show an example that I use with the complex business logic described in section 4.2. In this pattern, I use a per-business logic repository pattern. Because business logic can be complex, often with complicated validation rules, I find stubbing to be a useful approach to replacing the database access. The stub provides a lot more control of the database access, and you can more easily simulate various error conditions, but it does take longer to write the mocking and unit tests.

As an example of this approach, I am going to stub the database when testing the business logic that handles orders for books. The book-order business logic method uses the repository pattern to separate database access code from the business logic because it makes the business logic code simpler; it also helps with unit testing because I can replace the database access code with a test class that can replace the database with a stub that matches the repository interface. I find that stubbing gives me much better control of the data going into, and out of, the method I'm testing.

This next example is taken from my unit tests in the book's GitHub repo; here, you want to test the `PlaceOrderAction` method developed in chapter 4. The `PlaceOrderAction` class's constructor requires one parameter of type `IPlaceOrderDbAccess`, which is normally the `PlaceOrderDbAccess` class that handles the database accesses. But for testing, you replace the `PlaceOrderDbAccess` class with our test class—our stub that implements the same `IPlaceOrderDbAccess` interface. This stub class allows you to control what the `PlaceOrderAction` class can read from the database and capture what it attempts to write to the database. The following listing shows a unit test that uses this mock, which captures the order that the `PlaceOrderAction` method produces so that you can check whether the user's ID was set properly.

Listing 17.16 A unit test providing a stub instance to the `BizLogic`

Creates an instance of the mock database access code. This instance has numerous controls, but in this case, you use the default settings.

```
[Fact]
public void ExampleOfStubbingOk()
{
    //SETUP
    var lineItems = new List<OrderLineItem>
    {
        new OrderLineItem {BookId = 1, NumBooks = 4}
    };
    var userId = Guid.NewGuid();
    var input = new PlaceOrderInDto(true, userId,
        lineItems.ToImmutableList());

    var stubDbA = new StubPlaceOrderDbAccess();
    var service = new PlaceOrderAction(stubDbA);

    //ATTEMPT
    service.Action(input);

    //VERIFY
    service.Errors.Any().ShouldEqual(false);
    mockDbA.AddedOrder.CustomerId
        .ShouldEqual(userId);
}
```

Creates the input to the PlaceOrderAction method

Creates your PlaceOrderAction instance, providing it a mock of the database access code

Checks that the order placement completed successfully

Your mock database access code has captured the order that the PlaceOrderAction's method "wrote" to the database, so you can check whether it was formed properly.

Runs the PlaceOrderAction's method called Action, which takes in the input data and outputs an order

The stub class, `StubPlaceOrderDbAccess`, doesn't access the database, but it has properties or methods that you can use to control every part of the reading of data from the database. This class also captures anything the `PlaceOrderAction` method tries to write to the database, so you can check that too. Listing 17.17 shows the stub database class, `StubPlaceOrderDbAccess`. Note that I created a static method called `CreateDummyBooks` to generate a known set of `Books` to use in this test (see section 17.9).

Listing 17.17 The stub database access code used for unit testing

Mock MockPlaceOrderDbAccess implements the IPlaceOrderDbAccess, which allows it to replace the normal PlaceOrderDbAccess class.

```
public class StubPlaceOrderDbAccess
    : IPlaceOrderDbAccess
{
    public ImmutableList<Book> DummyBooks
        { get; private set; }

    public Order AddedOrder { get; private set; }

    public StubPlaceOrderDbAccess(
        bool createLastInFuture = false,
        int? promoPriceFirstBook = null)
    {
        var numBooks = createLastInFuture
            ? DateTime.UtcNow.Year -
                EfTestData.DummyBookStartDate.Year + 2
            : 10;
        var books = EfTestData.CreateDummyBooks
            (numBooks, createLastInFuture);
        if (promotionPriceForFirstBook != null)
            books.First().Promotion = new PriceOffer
            {
                NewPrice = (int) promoPriceFirstBook,
                PromotionalText = "Unit Test"
            };
        DummyBooks = books.ToImmutableList();
    }

    public IDictionary<int, Book>
        FindBooksByIdsWithPriceOffers
        (IEnumerable<int> bookIds)
    {
        return DummyBooks.AsQueryable()
            .Where(x => bookIds.Contains(x.BookId))
            .ToDictionary(key => key.BookId);
    }

    public void Add(Order newOrder)
    {
        AddedOrder = newOrder;
    }
}
```

Holds the dummy books that the mock uses, which can be useful if the test wants to compare the output with the dummy database

Will contain the Order built by the PlaceOrder-Action's method

In this case, you set up the mock via its constructor.

Allows you to check that a book that hasn't been published yet won't be accepted in an order

Works out how to create enough books so that the last one isn't published yet

Allows you to add a PriceOffer to the first book so you can check that the correct price is recorded on the order

Adds a PriceOffer to the first book, if required

Creates a method to create dummy books for your test

Called to get the books that the input selected; uses the DummyBooks generated in the constructor

Similar code to the original, but in this case reads from the DummyBooks, not the database

Called by the PlaceOrderAction's method to write the Order to the database. In this case, you capture the Order so that the unit test can inspect it.

As I said earlier, the stubbing code is long and a bit complicated to write, but because you copied the real `PlaceOrderDbAccess` class and then edited it, the job isn't too hard.

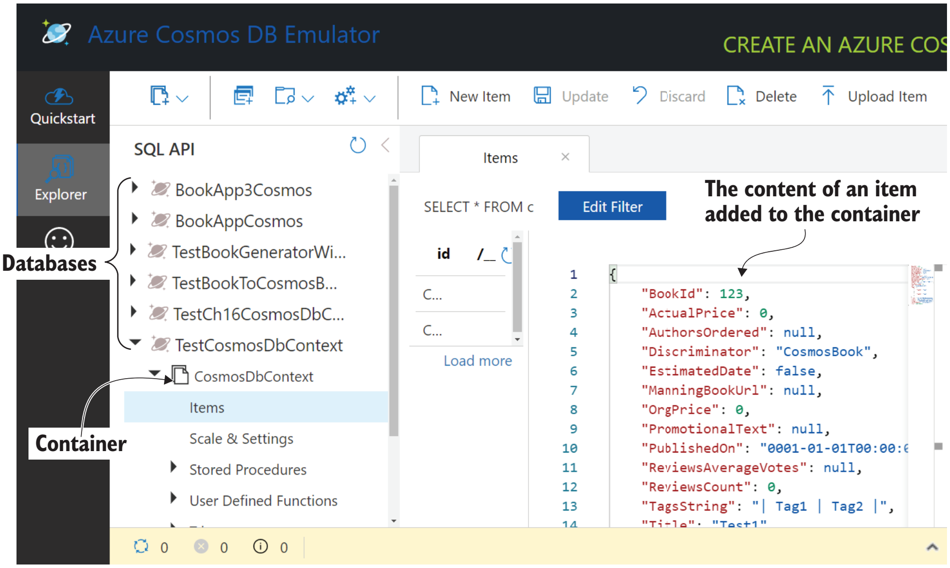
17.8 Unit testing a Cosmos DB database

Unit testing a Cosmos DB database doesn't fit any of the three approaches described in section 17.4, but it's closest to mocking the database because Microsoft has created an application called Azure Cosmos DB Emulator, which you can run on your development PC and test against. Microsoft's documentation at http://mng.bz/RK8j says

> *The Azure Cosmos DB Emulator provides a high-fidelity emulation of the Azure Cosmos DB service. It supports equivalent functionality as the Azure Cosmos DB, which includes creating data, querying data, provisioning and scaling containers, and executing stored procedures and triggers.*

You need to download the Azure Cosmos DB Emulator from http://mng.bz/4MOj and run it locally. When you run the emulator, it provides a URL that takes you to the emulator's quick-start web page, which contains details on accessing this Cosmos DB service. The emulator's website also has a useful Explorer (see figure 17.5) that provides full access and configuration of the databases, containers, and items within a container.

NOTE The Azure Cosmos DB Emulator is available only for Windows.



The content of an item added to the container

Databases

Container

```
{
    "BookId": 123,
    "ActualPrice": 0,
    "AuthorsOrdered": null,
    "Discriminator": "CosmosBook",
    "EstimatedDate": false,
    "ManningBookUrl": null,
    "OrgPrice": 0,
    "PromotionalText": null,
    "PublishedOn": "0001-01-01T00:00:
    "ReviewsAverageVotes": null,
    "ReviewsCount": 0,
    "TagsString": "| Tag1 | Tag2 |",
    "Title": "Test1"
```

Figure 17.5 When you run the Azure Cosmos DB Emulator, it gives you a URL to access information about the Cosmos DB emulator settings on the quick-start page, as well as access to the emulated Cosmos DB databases and containers via the Explorer page, as shown in this figure. The Explorer page provides full access to and configuration of the databases, containers, and items within a container.

In section 16.5, you learned that to access a Cosmos DB service, you need a connection string, and the Emulator's quick-start page provides the connection string for you. The following listing shows how.

Listing 17.18 Setting up EF Core to access a Cosmos DB database

```
public async Task AccessCosmosEmulatorViaConnectionString()
{
    //SETUP
    var connectionString =
        "AccountEndpoint=https://localhost… rest left out"
    var builder = new
            DbContextOptionsBuilder<CosmosDbContext>()
        .UseCosmos(
            connectionString,
            "MyCosmosDatabase");
    using var context = new CosmosDbContext(builder.Options);

    //… rest of the unit test left out
}
```

The connection string taken from the quick-start page of the emulator' website

Builds the options for the CosmosDbContext

The connection string is provided first.

Creates an instance of the application's DbContext

UseCosmos method is found in the Microsoft.EntityFrameworkCore.Cosmos NuGet package.

The name you want for the database

That approach works fine, but because the connection string is the same wherever you run the emulator, you can build a method to set up the options automatically. I added methods of that type to the EfCore.TestSupport version 5 NuGet packages. These methods follow the same approach as the EfCore.TestSupport SQL Server methods (see figure 17.4), where the class name (and optionally the method name) is used to form the database name.

The following listing shows the use of EfCore.TestSupport's `CreateUniqueClassCosmosDbEmulator` method to set up the options for an application DbContext called `CosmosDbContext`. This code creates a Cosmos database with the same name as the unit test class type, which makes the database unique in your project.

Listing 17.19 Unit testing Cosmos DB code by using the Cosmos DB Emulator

```
[Fact]
public async Task TestAccessCosmosEmulator()
{
    //SETUP
    var options = this.
       CreateUniqueClassCosmosDbEmulator
        <CosmosDbContext>();

    using var context = new CosmosDbContext(options);

    await context.Database.EnsureDeletedAsync();
    await context.Database.EnsureCreatedAsync();
```

This method sets up the Cosmos DB database options with the database name taken from the class name.

Creates the DbContext to access that database

Creates an empty database with the correct structure

```
    //… rest of unit test left out
}
```

As I stated in section 16.6.6, the `EnsureCreatedAsync` method is the recommended way to create an empty Cosmos DB database. Therefore, using the `EnsureDeletedAsync` and then `EnsureCreatedAsync` method is the correct way to delete and re-create a Cosmos DB database. Fortunately, the process is quick.

17.9 Seeding a database with test data to test your code correctly

Often, a unit test needs certain data in the database before you can run a test. To test the code that handles orders for books, for example, you need some `Books` in the database before you run the test. In cases like this one, you would add some code in the Setup stage of the unit test to add those books before you test the order code in the Verify stage.

My experience is that setting up the database with data to test some feature in a real application can quite complex. In fact, setting up the database with the right type can be much harder than running and verifying the test's results. Here are some tips on seeding a unit test database:

- It's OK at the start to write the setup code in the unit test, but as soon as you find yourself copying that setup code, it's time to turn that code into a method.
- I created two types of help methods in my Test project to help me set up test data, and I gave them good names so I can quickly identify what they do. The two types are
  - Ones that return the test data, with names such as `CreateFourBooks()` and `CreateDummyBooks(int numBooks = 10)`. I use these methods when I want to test adding these types to the database.
  - Ones that write the test data to the database, with names such as `SeedDatabaseFourBooks()` and `AddDummyBooksToDb()`. These methods write the test data to the database and normally return the added data so that I can get their primary keys to use in a test.
- Keep your test-data setup methods up to date, refactoring them as you come across different scenarios.
- Consider storing complex test data in a JSON file. I created a method to serialize data from a production system to a JSON file and have another method that will deserialize that data and write it to the database. But make sure that you anonymize any personal data before you save the JSON.
- The `EnsureCreated` method will also seed the database with data configured via the `HasData` configuration (see section 9.4.3).

17.10 Solving the problem of one database access breaking another stage of your test

In section 17.9, I describe how to add data to the test database, referred to as *seeding* your database, before you run your test. But a problem can arise in your test because of EF Core's relational fixup stage (see section 6.1.1) in a database query. Every tracked database query (that is, a query without the `AsNoTracking` method in it) will try to reuse the instances of any the entities already tracking by the unit test's DbContext. The effect is that any tracked query can affect any tracked query after it, so it can affect the Attempt and Verify parts of your unit test.

An example is the best way to understand this concept. Suppose that you want to test your code for adding a new `Review` to a `Book`, and you wrote the code shown in the following snippet:

```
var book = context.Books
   .OrderBy(x => x.BookId).Last();
book.Reviews.Add( new Review{NumStars = 5});
context.SaveChanges();
```

But there's a problem with this code: it has a bug. The code should have `Include(b => b.Reviews)` added to the first line to ensure that the current `Reviews` are loaded first. But if you're not careful, your unit test will work as it does in the following listing.

Listing 17.20 An INCORRECT simulation of a disconnected state, with the wrong result

```
[Fact]
public void INCORRECTtestOfDisconnectedState()
{
    //SETUP
    var options = SqliteInMemory
           .CreateOptions<EfCoreContext>();
    using var context = new EfCoreContext(options);

    context.Database.EnsureCreated();
    context.SeedDatabaseFourBooks();

    //ATTEMPT
    var book = context.Books
        .OrderBy(x => x.BookId).Last();
    book.Reviews.Add(new Review { NumStars = 5 });
    context.SaveChanges();

    //VERIFY
    //THIS IS INCORRECT!!!!!
    context.Books
        .OrderBy(x => x.BookId).Last()
        .Reviews.Count.ShouldEqual(3);
}
```

Sets up the test database with test data consisting of four books

Reads in the last book from your test set, which you know has two reviews

Adds another Review to the book, which shouldn't work but does because the seed data is still being tracked by the DbContext instance

Saves the Review to the database

Checks that you have three Reviews, which works, but the unit test should have failed with an exception

In fact, this unit test has two errors because of tracked entities:

- *Attempt* stage—Should have failed because the `Reviews` navigational property was `null`, but works because of relational fixup from the Setup stage
- *Verify stage*—Should fail if a `context.SaveChanges` call was left out, but works because of relational fixup from the Attempt stage

To my mind, the worst outcome—even worse than not having a unit test—is a unit test that works when it shouldn't so that you think something is fine when it isn't. Let's look at ways to change the incorrect unit test in listing 17.20 so that it will fail properly. Previously, there was only one way to handle this problem, but another approach has been possible since EF Core 5. The two approaches are

- Use EF Core 5's `ChangeTracker.Clear` method to clear the tracked entities
- Use multiple instances within `using` scopes (original approach)

I find the EF Core 5's `ChangeTracker.Clear` approach to be quicker to write and shorter, so I show it first, but I also show the original multiple-instances approach for comparison purposes.

17.10.1 Test code using ChangeTracker.Clear in a disconnected state

The following listing solves the problem of the seeding data affecting the Attempt stage and the Attempt stage affecting the Verify stage. In this case, an exception is thrown, as the `Reviews` collection is `null` (assuming that you followed my recommendation in section 6.1.6). If the Attempt stage was fixed, the code in the Verify stage would be able to detect that `SaveChanges` wasn't called.

Listing 17.21 Using `ChangeTracker.Clear` to make the unit test work properly

```
[Fact]
public void UsingChangeTrackerClear()
{
    //SETUP
    var options = SqliteInMemory
        .CreateOptions<EfCoreContext>();
    using var context = new EfCoreContext(options);

    context.Database.EnsureCreated();
    context.SeedDatabaseFourBooks();

    context.ChangeTracker.Clear();

    //ATTEMPT
    var book = context.Books
        .OrderBy(x => x.BookId).Last();
    book.Reviews.Add(new Review { NumStars = 5 });
```

Sets up the test database with test data consisting of four books

Calls ChangeTracker .Clear to stop tracking all entities

Reads in the last book from your test set, which you know has two reviews

When you try to add the new Review, EF Core throws a NullReferenceException because the Book's Review collection isn't loaded and therefore is null.

```
    context.SaveChanges();

    //VERIFY
    context.ChangeTracker.Clear();

    context.Books.Include(b => b.Reviews)
        .OrderBy(x => x.BookId).Last()
        .Reviews.Count.ShouldEqual(3);
}
```

Saves the Review to the database

Calls ChangeTracker.Clear to stop tracking all entities

Reloads the book with its Reviews to check whether there are three Reviews

If you compare listing 17.21 with listing 17.22, you see that the code is shorter by nine lines, mainly because you don't need all the scoped `using` blocks in listing 17.22. I also find this approach to be slightly easier to read without all the scoped blocks.

17.10.2 Test code by using multiple DbContext instances in a disconnected state

The following listing uses two instances of the application's DbContext: one to set up the database and one to run the test. The test fails because an exception is thrown, as the `Reviews` collection is `null` (assuming that you followed my recommendation in section 6.1.6).

Listing 17.22 Three separate DbContext instances that make the test work properly

```
[Fact]
public void UsingThreeInstancesOfTheDbcontext()
{
    //SETUP
    var options = SqliteInMemory
        .CreateOptions<EfCoreContext>();
    options.StopNextDispose();
    using (var context = new EfCoreContext(options))
    {
        context.Database.EnsureCreated();
        context.SeedDatabaseFourBooks();
    }
    options.StopNextDispose();
    using (var context = new EfCoreContext(options))
    {
        //ATTEMPT
        var book = context.Books
            .Include(x => x.Reviews)
            .OrderBy(x => x.BookId).Last();
        book.Reviews.Add(new Review { NumStars = 5 });

        context.SaveChanges();
    }
    using (var context = new EfCoreContext(options))
    {
```

Creates the in-memory SQLite options in the same way as the preceding example

Stops the SQLite connection from being disposed after the next instance of the application's DbContext is disposed

Creates the first instance of the application's DbContext

Sets up the test database with test data consisting of four books, but this time in a separate DbContext instance

Reads in the last book from your test set, which you know has two Reviews

When you try to add the new Review, EF Core throws a NullReferenceException because the Book's Review collection isn't loaded and therefore is null.

Calls SaveChanges to update the database

Closes that last instance and opens a new instance of the application's DbContext. The new instance doesn't have any tracked entities that could alter how the test runs.

```
        //VERIFY
        context.Books.Include(b => b.Reviews)
            .OrderBy(x => x.BookId).Last()
            .Reviews.Count.ShouldEqual(3);
    }
}
```

Reloads the Book with its Reviews to check whether there are three Reviews

17.11 Capturing the database commands sent to a database

Sometimes, it's helpful to see what EF Core is doing when it accesses a real database, and EF Core provides a couple of ways to do that. Inspecting the EF Core logging from your running application is one way, but it can be hard to find the exact log among all the other logs. Another, more focused approach is to write unit tests that test specific parts of your EF Core queries by capturing SQL commands that EF Core would use to query the database.

The EF Core logs often contain the SQL commands but also carry other information, such as warnings of possible problems and timings (how long the database access took). Also, even if you don't know the SQL language well, it's not hard to check whether the configuration changes you made created the expected changes in the database. EF Core 5 added two new features that make capturing database commands much easier than in previous versions:

- The `LogTo` option extension, which makes it easy to filter and capture EF Core logging
- The `ToQueryString` method, which shows the SQL generated from a LINQ query

17.11.1 Using the LogTo option extension to filter and capture EF Core logging

Before EF Core 5, getting logs out of EF Core required you to build an `ILoggerProvider` class and register that logger provider via the `UseLoggerFactory` options extension method. This technique wasn't easy. The EF Core 5's `LogTo` option extension method makes it much easier to get log output and adds some features that filter the logs you want to see.

The `LogTo` method typically returns each log via an `Action<string>` type, and you can add the logs to a `List<string>` variable or output to some console. In xUnit, you would use the xUnit `ITestOutputHelper`'s `WriteLine` method, as shown in the following listing.

Listing 17.23 Outputting logs from an xUnit test by using the `LogTo` method

```
public class TestLogTo
{
    private readonly ITestOutputHelper _output;
```

The class holding your unit tests of LogTo

An xUnit interface that allows output to the unit test runner

```
    public TestLogTo(ITestOutputHelper output)
    {
        _output = output;
    }

    [Fact]
    public void TestLogToDemoToConsole()
    {
        //SETUP
        var connectionString =
            this.GetUniqueDatabaseConnectionString();
        var builder =
            new DbContextOptionsBuilder<BookDbContext>()
            .UseSqlServer(connectionString)
            .EnableSensitiveDataLogging()
            .LogTo(_output.WriteLine);

        using var context = new BookDbContext(builder.Options);
        // … rest of unit test left out
    }
}
```

xUnit will inject the ITestOutputHelper via the class's constructor.

This method contains a test of LogTo.

Provides a database connection where the database name is unique to this class

Sets up the option builder to an SQL Server database

It is good to turn on EnableSensitiveDataLogging in your unit tests.

Adds the simplest form of the LogTo method, which calls an Action<string> method

The default has the following format:

- `LINE1: <loglevel(4 chars)> <DateTime.Now> <EventId> <Category>`
- `LINE2: <the log message>`

The following code snippet shows one of the logs in this format:

- LINE1: `warn: 10/12/2020 11:59:38.658 CoreEventId.SensitiveDataLoggingEnabledWarning[10400] (Microsoft.EntityFrameworkCore.Infrastructure)`
- LINE2: `Sensitive data logging is enabled. Log entries and exception messages may include sensitive application data; this mode should only be enabled during development.`

As well as outputting the logs, the `LogTo` method can filter by the following types:

- `LogLevel`, such as `LogLevel.Information` or `LogLevel.Warning`
- `EventIds`, which define a specific log output, such as `CoreEventId.ContextInitialized` and `RelationalEventId.CommandExecuted`
- Category names, which EF Core defines for commands in groups, such as `DbLoggerCategory.Database.Command.Name`
- Functions that take in the `EventId` and the `LogLevel` and return `true` for the logs you want to be output

This method is great, but there are so many options to choose from for adding the `LogTo` feature to the EfCore.TestSupport library that I built a class called `LogToOptions` to handle all the settings (along with code to throw an exception if the combination you picked isn't supported). The `LogToOptions` class also includes some different defaults from `LogTo`'s defaults, which are based on my experience with logging in unit tests. The changes are

- The default `LogLevel` should be `Information`. (I find `Debug` `LogLevel` logs to be useful only if I am trying to find a bug.)
- I don't want a `DataTime` in a log, because that means I can't compare a log with a constant string, so I set the `DbContextLoggerOptions` parameter to `None`. (The `DbContextLoggerOptions` controls the log output and can add extra information to the log string.)
- Most times, I don't want to see logs of the Setup stage of the unit test, so I added a `bool ShowLog` property (defaults to `true`) to allow you to control when the `Action<string>` parameter is called.

Here is a listing of the `LogToOptions` class with comments on each property.

Listing 17.24 The `LogToOptions` class with all the settings for the `LogTo` method

```
public class LogToOptions
{
    public bool ShowLog { get; set; }
        = true;

    public LogLevel LogLevel { get; set; }
         = LogLevel.Information;

    public string[] OnlyShowTheseCategories
         { get; set; }

    public EventId[] OnlyShowTheseEvents
         { get; set; }

    public Func<EventId, LogLevel, bool>
         FilterFunction { get; set; }

    public DbContextLoggerOptions
         LoggerOptions { get; set; }
             = DbContextLoggerOptions.None
}
```

If false, your Action<string> method isn't called; defaults to true

Only logs at or higher than the LogLevel property will be output; defaults to LogLevel.Information

If not null, returns only logs with a Category name in this array; defaults to null

If not null, returns only logs with an EventId in this array; defaults to null

If not null, this function is called, and logs only where this function returns true are returned; defaults to null

Controls the format of the EF Core log. The default setting does not prefix the log with extra information, such as LogLevel, DateTime, and so on.

Now let's use the `LogToOptions` class with EfCore.TestSupport's `SqliteInMemory.CreateOptionsWithLogTo` method. In the following listing, you use the `ShowLog` property in the `LogToOptions` class to display the logs only after the Setup stage of the unit test has finished.

Listing 17.25 Turning off log output until the `//SETUP` stage of the unit test is finished

```
[Fact]
public void TestEfCoreLoggingCheckSqlOutputShowLog()
{
    //SETUP
```

```
    var logToOptions = new LogToOptions
    {
        ShowLog = false
    };
    var options = SqliteInMemory
        .CreateOptionsWithLogTo
        <BookDbContext>(
            _output.WriteLine,
            logToOptions);

    using var context = new BookDbContext(options);
    context.Database.EnsureCreated();
    context.SeedDatabaseFourBooks();

    //ATTEMPT
    logToOptions.ShowLog = true;
    var book = context.Books.Count();

    //VERIFY
}
```

The parameter is your Action<string> method and must be provided.

In this case, you want to change the default LogToOptions to set the ShowLog to false.

This method sets up the SQLite in-memory options and adds LogTo to those options.

This setup and seed section doesn't produce any output because the ShowLog property is false.

Turns on the logging output by setting the ShowLog property to true

This query produces one log output, which will be sent to the xUnit runner's window.

The second parameter is optional, but in this case, you want to provide the logToOptions to control the output.

The result is that instead of wading through the logs from creating the database and seeding the database, you see only one log output in the xUnit runner's window, as shown in the following code snippet:

```
Executed DbCommand (0ms) [Parameters=[],
        CommandType='Text', CommandTimeout='30']
SELECT COUNT(*)
FROM "Books" AS "b"
WHERE NOT ("b"."SoftDeleted")
```

17.11.2 Using the ToQueryString method to show the SQL generated from a LINQ query

The logging output is great and contains lots of useful information, but if you simply want to see what your query looks like, you have a much simpler way. If you have built a database query that returns an `IQueryable` result, you can use the `ToQueryString` method. The following listing incorporates the output of the `ToQueryString` method in the test.

Listing 17.26 A unit test containing the `ToQueryString` method

```
[Fact]
public void TestToQueryStringOnLinqQuery()
{
    //SETUP
    var options = SqliteInMemory.CreateOptions<BookDbContext>();
    using var context = new BookDbContext(options);
```

```
    context.Database.EnsureCreated();
    context.SeedDatabaseFourBooks();

    //ATTEMPT
    var query = context.Books.Select(x => x.BookId);
    var bookIds = query.ToArray();

    //VERIFY
    _output.WriteLine(query.ToQueryString());
    query.ToQueryString().ShouldEqual(
        "SELECT \"b\".\"BookId\"\r\n" +
        "FROM \"Books\" AS \"b\"\r\n" +
        "WHERE NOT (\"b\".\"SoftDeleted\")");
    bookIds.ShouldEqual(new []{1,2,3,4});
}
```

You provide the LINQ query without an execution part.

Then you run the LINQ query by adding ToArray on the end.

Outputs the SQL for your LINQ query

Tests whether the SQL is what you expected

Tests the output of the query

Summary

- Unit testing is a way to test a *unit* of your code—a small piece of code that can be logically isolated in your application.
- Unit testing is a great way to catch bugs when you develop your code and, more important, when you or someone else refactors your code.
- I recommend using xUnit because it is widely used (EF Core uses xUnit and has ~70,000 tests), well supported, and fast. I also have built a library called EfCore.TestSupport that provides methods to make testing EF Core code in xUnit easier.
- An application's DbContext designed to work with an ASP.NET Core application is ready for unit testing, but any application's DbContext that uses the `OnConfiguring` method to set options needs to be modified to allow unit testing.
- There are three main ways to simulate a database when unit testing, each with its own trade-offs:
  - *Using the same type of database as your production database*—This approach is the safest, but you need to deal with out-of-date database schemas and managing databases to allow parallel running of unit test classes.
  - *Using an SQLite in-memory database*—This approach is the fastest and easiest, but it doesn't mimic every SQL feature of your production database.
  - *Stubbing the database*—When you have a repository pattern for accessing the database, such as in business logic (see section 4.4.3), stubbing that repository gives you fast and comprehensive control of the data for unit testing, but it typically needs more test code to be written.
- Cosmos DB has a handy Azure Cosmos DB Emulator that you can download and run locally. This application allows you to unit test Cosmos DB without needing an Azure Cosmos DB service.
- Many unit tests need the test database to contain some data to be used in the test, so it's worth spending time to design a suite of test methods that will create test data to use in your unit tests.

- Your unit tests might say that the code under test is correct when it's not. This situation can happen if one section of your unit test is picking up tracked instances from a previous stage of the test. You have two ways to ensure that this problem doesn't happen: use separate DbContext instances or use `ChangeChanger.Clear`.
- EF Core 5 has added two methods that make capturing the SQL produced from your code much easier: the `LogTo` option to capture logging output and the `ToQueryString` method to convert LINQ queries to database commands.

appendix A
A brief introduction to LINQ

This appendix is for anyone who is new to Microsoft's Language Integrated Query (LINQ) feature or anyone who wants a quick recap of how LINQ works. The LINQ language bridges the gap between the world of objects and the world of data, and is used by EF Core to build database queries. Understanding the LINQ language is key to using EF Core to access a database.

This appendix starts with the two syntaxes you can use to write LINQ code. You'll also learn the types of commands available in LINQ, with examples of how those commands can manipulate collections of in-memory data.

Then you'll explore the related .NET type `IQueryable<T>`, which holds LINQ code in a form that can be executed later. This type allows developers to split complex queries into separate parts and change the LINQ query dynamically. The `IQueryable<T>` type also allows EF Core to translate the LINQ code into commands that can be run on the database server. Finally, you'll learn what an EF Core query, with its LINQ part, looks like.

A.1 An introduction to the LINQ language

You can manipulate collections of data by using LINQ's methods to sort, filter, select, and so on. These collections can be in-memory data (such as an array of integers, XML data, or JSON data) and of course databases, via libraries such as EF Core. The LINQ feature is available in Microsoft's languages C#, F#, and Visual Basic; you can create readable code by using LINQ's functional programming approach.

TIP If you haven't come across functional programming, it's worth taking a look at it. See http://mng.bz/97CY or, for a more in-depth, .NET-focused book, Enrico Buonanno's *Functional Programming in C#* (Manning, 2017; http://mng.bz/Q2Qv).

A.1.1 The two ways you can write LINQ queries

LINQ has two syntaxes for writing LINQ queries: the *method* syntax and the *query* syntax. This section presents the two syntaxes and points out which one is used in this book. You'll write the same LINQ query, a filter, and a sort of array of integers in both syntaxes.

Listing A.1 uses what is known as the LINQ *method*, or *lambda*, syntax. This code is a simple LINQ statement. Even if you haven't seen LINQ before, the names of the LINQ methods, such as `Where` and `OrderBy`, provide a good clue to what's going on.

Listing A.1 Your first look at the LINQ language using the method/lambda syntax

```
int[] nums = new[] {1, 5, 4, 2, 3, 0};

int[] result = nums
    .Where(x => x > 3)
    .OrderBy(x => x)
    .ToArray();
```

Creates an array of integers from 0 to 5, but in a random order

Applies LINQ commands and returns a new array of integers

Orders the numbers

Filters out all the integers 3 and below

Turns the query back into an array. The result is an array of ints { 4, 5 }.

The *lambda* name comes from lambda syntax, introduced in C# 3. The lambda syntax allows you to write a method without all the standard method definition syntax. The `x => x > 3` part inside the `Where` method is equivalent to the following method:

```
private bool AnonymousFunc(int x)
{
    return x > 3;
}
```

As you can see, the lambda syntax can save a significant amount of typing. I use lambdas in all of my EF Core queries and in lots of other code I wrote for this book.

The next listing shows the other way of writing LINQ code, called the *query* syntax. This code achieves the same result as listing A.1 but returns a slightly different result type.

Listing A.2 Your first look at the LINQ language using the query syntax

```
int[] nums = new[] { 1, 5, 4, 2, 3, 0};

IOrderedEnumerable<int> result =
    from num in nums
    where num > 3
    orderby num
    select num;
```

Creates an array of integers from 0 to 5, but in random order

The result returned here is an IOrderedEnumerable<int>.

The query syntax starts with a from <item> in <collection>.

Filters out all the integers 3 and below

Orders the numbers

Applies a select to choose what you want. The result is an IOrderedEnumerable<int> containing { 4, 5 }.

You can use either syntax; the choice is up to you. I use the method syntax because it involves slightly less typing and because I like the way that commands are chained together, one after the other. The rest of the examples in this book use the method syntax.

Before I leave the topic of the LINQ syntax, I want to introduce the concept of precalculating values in a LINQ query. The query syntax has a feature specifically to handle this task: the `let` keyword. This keyword allows you to calculate a value once and then use that value multiple times in the query, making the query more efficient. This listing shows code that converts an integer value to its word/string equivalent and then uses that string in both the sort and filter parts of the query.

Listing A.3 Using the `let` keyword in a LINQ query syntax

```
int[] nums = new[] { 1, 5, 4, 2, 3, 0 };
string [] numLookop = new[]
    {"zero","one","two","three","four","five"};

IEnumerable<int> result =
    from num in nums
    let numString = numLookop[num]
    where numString.Length > 3
    orderby numString
    select num;
```

Creates an array of integers from 0 to 5, but in random order

A lookup to convert a number to its word format

The result returned here is an IEnumerable<int>.

The query syntax starts with a from <item> in <collection>.

Orders the number by the word form

The let syntax allows you to calculate a value once and use it multiple times in the query.

Applies a select to choose what you want. The result is an IEnumerable<int> containing { 5,4,3,0 }.

Filters out all the numbers indicating that the word is shorter than three letters

The equivalent in the method syntax is the LINQ `Select` operator earlier in the query, as shown in the following listing. (Section A.1.2 provides more details about the LINQ `Select` operator.)

Listing A.4 Using the LINQ `Select` operator to hold a calculated value

```
int[] nums = new[] { 1, 5, 4, 2, 3, 0 };
string[] numLookop = new[]
    {"zero","one","two","three","four","five"};

IEnumerable<int> result = nums
    .Select( num => new
        {
            num,
            numString = numLookop[num]
        })
    .Where(r => r.numString.Length > 3)
    .OrderBy(r => r.numString)
    .Select(r => r.num);
```

Applies another Select to choose what you want. The result is an IEnumerable<int> containing { 5,4,3,0 }.

Creates an array of integers from 0 to 5, but in random order

A lookup to convert a number to its word format

The result returned here is an IEnumerable<int>.

Uses an anonymous type to hold the original integer value and your numString word lookup

Orders the number by the word form

Filters out all the numbers indicating that the word is shorter than three letters

EF6 EF6.x used the `let` or the `Select` as a hint to precalculate a value only once in the database. EF Core doesn't have that performance feature, so it recalculates every occurrence of a value.

A.1.2 The data operations you can do with LINQ

The LINQ feature has many methods, referred to as *operators*. Most operators have names and functions that clearly indicate what's going on. Table A.1 lists some of the most common LINQ operators; similar operators are grouped to help you see where they might be used. The list is not exhaustive; the aim is to show you some of the most common operators to give you a feel for what LINQ can do.

Table A.1 Examples of LINQ operators, grouped by purpose

Group	Examples (not all operators shown)
Sorting	`OrderBy, OrderByDescending, Reverse`
Filtering	`Where`
Select element	`First, FirstOrDefault`
Projection	`Select`
Aggregation	`Max, Min, Sum, Count, Average`
Partition	`Skip, Take`
Boolean tests	`Any, All, Contains`

Listing A.4 shows a LINQ query that sorts and filters an array of `int` numbers. Now we are going to look at some examples in which the LINQ query works on a C# class. First, you need to define a new class called `Review` with data to help with the examples, as shown in the following listing.

Listing A.5 A `Review` class and a `ReviewsList` variable containing two Reviews

```
class Review
{
    public string VoterName { get; set; }
    public int NumStars { get; set; }
    public string Comment { get; set; }
}

List<Review> ReviewsList = new List<Review>
{
    new Review
    {
        VoterName = "Jack",
        NumStars = 5,
        Comment = "A great book!"
    },
    new Review
```

```
    {
        VoterName = "Jill",
        NumStars = 1,
        Comment = "I hated it!"
    }
};
```

The `ReviewsList` field in LINQ code is shown in table A.2. This table should give you a feel for how various LINQ operators work.

Table A.2 Four uses of LINQ on the ReviewsList field as data. The result of each LINQ operator is shown in the Result value column.

LINQ group	Code using LINQ operators	Result value

Projection	`string[] result = ReviewsList`
	`.Select(p => p.VoterName)`
	`.ToArray();`
		`string[]{"Jack", "Jill"}`

Aggregation	`double result = ReviewsList` `.Average(p => p.NumStars);`	`3 (average of 5 and 1)`
Select element	`string result = ReviewsList` `.First().VoterName;`	`"Jack" (first voter)`
Boolean test	`bool result = ReviewsList` `.Any(p => p.NumStars == 1);`	`true (Jill voted 1)`

A.2 Introduction to IQueryable<T> type, and why it's useful

Another important part of LINQ is the generic interface `IQueryable<T>`. LINQ is rather special, in that whatever set of LINQ operators you provide isn't executed straightaway but is held in a type called `IQueryable<T>`, awaiting a final command to execute it. This `IQueryable<T>` form has two benefits:

- You can split a complex LINQ query into separate parts by using the `IQueryable<T>` type.
- Instead of executing the `IQueryable<T>`'s internal form, EF Core can translate it into database access commands.

A.2.1 Splitting up a complex LINQ query by using the IQueryable<T> type

In the book, you learn about Query Objects (see section 2.6), and you build a complex book list query by chaining together three Query Objects. This operation works because of the `IQueryable<T>` type's ability to hold the code in a specialized form, called an *expression tree*, so that other LINQ operators can be appended to it.

As an example, you're going to improve the code from listing A.1 by adding your own method that contains the sorting part of the query, allowing you to alter the sort order of the final LINQ query. You'll create this method as an extension method, which allows you to chain the method in the same way that the LINQ operators do. (LINQ operators are extension methods.)

> **DEFINITION** An *extension method* is a static method in a static class; the first parameter of the method has the keyword `this` in front of it. To allow chaining, the method must also return a type that other methods can use as an input.

Listing A.6 shows the extension method `MyOrder`, which takes in an `IQueryable<int>` type as its first parameter and returns an `IQueryable<int>` result. It also has a second `boolean` parameter called `ascending` that sets the sort order to ascending or descending.

Listing A.6 Your method encapsulates part of your LINQ code via `IQueryable<int>`

```
public static class LinqHelpers
{
    public static IQueryable<int> MyOrder
        (this IQueryable<int> queryable,
        bool ascending)
    {
        return ascending
            ? queryable
                .OrderBy(num => num)
            : queryable
                .OrderByDescending(num => num);
    }
}
```

Provides a second parameter that allows you to change the order of the sorting

Extension method needs to be defined in a static class

Static method Order returns an IQueryable<int> so other extension methods can chain on

Extension method's first parameter is of IQueryable and starts with the this keyword

Ascending parameter is true, so you add the OrderBy LINQ operator to the IQueryable input

Uses the Boolean parameter ascending to control whether you add the OrderBy or OrderByDescending LINQ operator to the IQueryable result

Ascending parameter is false, so you add the OrderByDescending LINQ operator to the IQueryable input

This listing uses this `IQueryable<int>` extension method to replace the `OrderBy` LINQ operator in the original code in listing A.1.

Listing A.7 Using the `MyOrder IQueryable<int>` method in LINQ code

```
var numsQ = new[] { 1, 5, 4, 2, 3 }
    .AsQueryable();

var result = numsQ
    .MyOrder(true)
    .Where(x => x > 3)
    .ToArray();
```

Turns an array of integers into a queryable object

Filters out all the numbers 3 and below

Calls the MyOrder IQueryable<int> method, with true, giving you an ascending sort of the data

Executes the IQueryable and turns the result into an array. The result is an array of ints { 4, 5 }.

Extension methods, such as the `MyOrder` example, provide two useful features:

- *They make your LINQ code dynamic.* By changing the parameter into the `MyOrder` method, you can change the sort order of the final LINQ query. If you didn't have that parameter, you'd need two LINQ queries—one using `OrderBy` and one using `OrderByDescending`—and then you'd have to pick which one you wanted to run by using an `if` statement. That approach isn't good software practice, as you'd be needlessly repeating some LINQ code, such as the `Where` part.

- *They allow you to split complex queries into a series of separate extension methods that you can chain.* This approach makes it easier to build, test, and understand complex queries. In section 2.9, you split your Book App's book list query, which is rather complicated, into separate *Query Objects*. The following listing shows this process again, with each Query Object highlighted in bold.

Listing A.8 The book list query with select, order, filter, and page Query Objects

```
public IQueryable<BookListDto> SortFilterPage
    (SortFilterPageOptions options)
{
    var booksQuery = _context.Books
        .AsNoTracking()
        .MapBookToDto()
        .OrderBooksBy(options.OrderByOptions)
        .FilterBooksBy(options.FilterBy,
                       options.FilterValue);

    options.SetupRestOfDto(booksQuery);

    return booksQuery.Page(options.PageNum-1,
                           options.PageSize);
}
```

The book list query uses both features I've mentioned: it allows you to change the sorting, filtering, and paging of the book list dynamically, and it hides some of the complex code behind an aptly named method that tells you what it's doing.

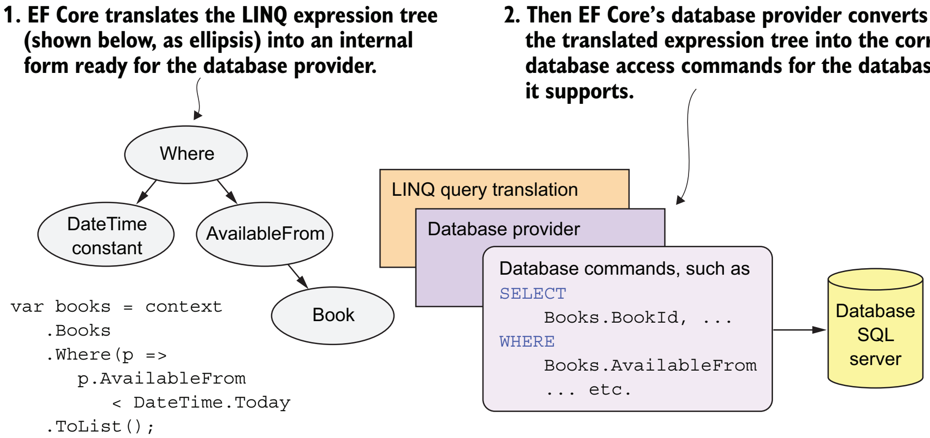
A.2.2 How EF Core translates IQueryable<T> into database code

EF Core translates your LINQ code into database code that can run on the database server. It can do this because the `IQueryable<T>` type holds all the LINQ code as an expression tree, which EF Core can translate into database access code. Figure A.1 shows what EF Core is doing behind the scenes when it translates a LINQ query into database access code.

EF Core provides many extra extension methods to extend the LINQ operators available to you. EF Core methods add to the LINQ expression tree, such as `Include`, `ThenInclude` (see section 2.4.1), and so on. Other EF methods provide async versions (see section 5.10) of the LINQ methods, such as `ToListAsync` and `LastAsync`.

A.3 Querying an EF Core database by using LINQ

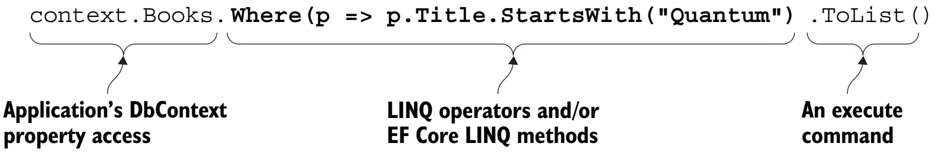
Using LINQ in an EF Core database query requires three parts, as shown in figure A.2. The query relies on an application's DbContext, which is described in section 2.2.1. This section concentrates on only the format of an EF Core database query, with the LINQ operators shown in bold.



```
var books = context
    .Books
    .Where(p =>
       p.AvailableFrom
          < DateTime.Today
    .ToList();
```

```
SELECT
    Books.BookId, ...
WHERE
    Books.AvailableFrom
    ... etc.
```

Figure A.1 Some book query code (bottom left) with its expression tree above it. EF Core takes the expression tree through two stages of translation before it ends up in the right form for the database that the application is targeting.



```
context.Books.Where(p => p.Title.StartsWith("Quantum")) .ToList()
```

Figure A.2 An example database access, with the three parts

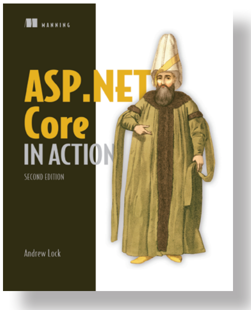
These three parts of an EF Core database query are as follows:

- *Application's DbContext property access*—In your application's DbContext, you define a property by using a `DbSet<T>` type. This type returns an `IQueryable<T>` data source to which you can add LINQ operators to create a database query.
- *LINQ operators and/or EF Core LINQ methods*—Your database LINQ query code goes here.
- *The execute command*—Commands such as `ToList` and `First` trigger EF Core to translate the LINQ commands into database access commands that are run on the database server.

In chapter 2 and onward, you'll see much more complex queries, but they all use the three parts shown in figure A.2.

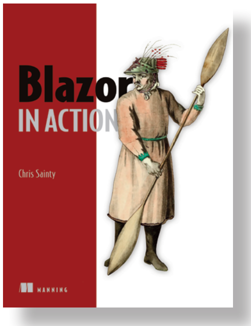
index

RELATED MANNING TITLES



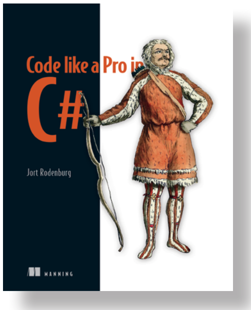
ASP.NET Core in Action, Second Edition
by Andrew Lock

ISBN 9781617298301
832 pages, $69.99
March 2021



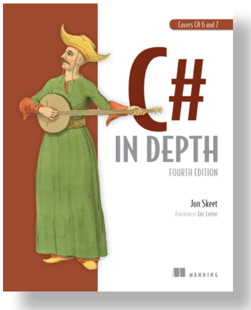
Blazor in Action
by Chris Sainty

ISBN 9781617298646
400 pages (estimated), $59.99
July 2021 (estimated)



Code like a Pro in C#
by Jort Rodenburg

ISBN 9781617298028
391 pages (estimated), $59.99
Summer 2021 (estimated)



C# in Depth, Fourth Edition
by Jon Skeet
Foreword by Eric Lippert

ISBN 9781617294532
528 pages, $49.99
March 2019

For ordering information go to www.manning.com

Here is a checklist of EF Core performance issues, with the section of the book that deals with that issue.

	Section
Analyze your performance	
Have you picked the right features to performance tune?	14.1.2
Have you diagnosed your performance issue?	14.2
Have you measured the user's experience?	14.2.1
Have you inspected the SQL produces for poor performance?	14.2.3
Querying your database	
Are you loading too many columns?	14.4.1
Are you loading too many rows?	14.4.2
Are you using lazy loading?	14.4.3
Are you telling EF Core that your query is read-only?	14.4.4
Are you making too many calls to the database?	14.5.1
Have you added indexes to properties you sort/filter on?	14.5.2
Are you using the fastest way to load an entity?	14.5.3
Is part of your query being run in software?	14.5.4
Can you move any calculations to run in the database?	14.5.5
Have you replaced any suboptimal SQL produced by a LINQ query?	14.5.6
Can you precompile frequently used queries?	14.5.7
Writing to the database	
Are you calling `SaveChanges` multiple times?	14.6.1
Are you making `DetectChanges` work too hard?	14.6.2
Have you used `HashSet<T>` for navigational properties?	14.6.3
Are you calling the `Update` method when you don't need to?	14.6.4
The scalability of your application	
Are you using DbContext pooling?	14.7.1
Are you using async/await throughout your application?	14.7.2
Have you picked the right architecture for high scalability?	14.7.4
Have you considered using Cosmos DB as a frontend cache?	16.3

.NET/DATABASES

Entity Framework Core IN ACTION

Second Edition

Jon P Smith

Entity Framework radically simplifies data access in .NET applications. This easy-to-use object-relational mapper (ORM) lets you write database code in pure C#. It automatically maps classes to database tables and enables queries with standard LINQ commands. It even generates SQL, so you don't have to!

Entity Framework Core in Action, Second Edition teaches you to write flawless database interactions for .NET applications. Following relevant examples from author Jon Smith's extensive experience, you'll progress quickly from EF basics to advanced techniques. In addition to the latest EF features, this book addresses performance, security, refactoring, and unit testing. This updated edition also contains new material on NoSQL databases.

What's Inside

- Configure EF to define every table and column
- Update your schema as your app grows
- Integrating EF with existing C# application
- Write and test business logic for database access
- Applying a Domain-Driven Design to EF Core
- Getting the best performance out of EF Core

For .NET developers familiar with relational databases.

Jon P Smith is a freelance software developer and architect with a special focus on .NET and Azure.

Register this print book to get free access to all ebook formats.
Visit https://www.manning.com/freebook

> “The most comprehensive reference for EF Core that does—or ever will—exist.”
>
> —Stephen Byrne, Intel Corporation

> “The definitive guide to EF Core. Filled with real world examples, it's the most practical way to enhance your EF Core skill set.”
>
> —Paul Brown
> Diversified Services Network

> “I firmly believe anyone using EF Core will find something to up their game in this book.”
>
> —Anne Epstein, Headspring

> “Remains a go-to resource for me while working with Entity Framework.”
>
> —Foster Haines, J2 Interactive

Free eBook
See first page

ISBN: 978-1-61729-836-3

55999

9 781617 298363

MANNING

$59.99 / Can $79.99 [INCLUDING eBOOK]