Entity Framework Core IN ACTION

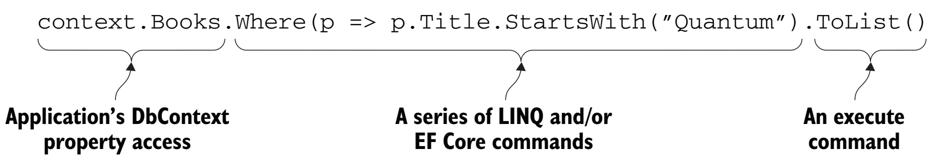
Jon P Smith

Sample Chapter

MANNING

Key topics covered in this book: the primary chapter covering each topic is listed first. Key figures that go with the topic are also listed.

Topics	Chapters	Key figures
Setting up EF Core	1, 2, 6, 7, 8, 5	1.4, 1.5, 2.6
Query the database	2, 5, 10	2.5, 2.9,
Create, update, delete	3, 5, 7, 10	3.1, 3.2, 3.3, 3.4
Business logic	4, 5, 10	4.2, 5.1, 5.4
ASP.NET Core	5, 2	5.1, 5.4
Dependency injection	5, 14, 15	5.2, 5.3
Async/await	5, 12	5.8, 5.9, 5.10
Configure non-relational	6	6.1, 6.2
Configure relationships	7, 8	7.1, 7.2, 7.3, 7.4
Configure table mappings	7	7.10, 7.11
Concurrency issues	8, 13	8.3, 8.4, 8.5, 8.6, 8.7
How EF Core works inside	1, 9, 14	1.6, 1.8, 1.10, 9.1
Design patterns	10, 4, 12	5.1, 10.1, 10.5, 10.6, 14.1, 14.2
Domain-driven design	10, 4	4.2, 10.5, 10.6
Database migrations	11, 5	11.1, 11.2, 11.3, 11.4, 11.5, 11.6
Performance tuning	12, 13, 14	12.1, 11.2, 11.4, 13.7, 14.5
Different databases	14	
Data validation	6, 4, 10	10.7
Unit testing	15	15.2
LINQ language	Appendix A, 2	A.2, A.1



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

Application's DbContext property access
A series of LINQ and/or EF Core commands
An execute command

An example of an Entity Framework Core database query

Entity Framework Core in Action

by Jon P Smith

Chapter 1

brief contents

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 allows software developers 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 the 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.

EF Core, which Microsoft released in 2016, 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. (But EF Core can be used with the existing .NET Framework too—see the note in section 1.10.5.) EF Core, ASP.NET Core (a web server-side application), and .NET Core are also all open source, each with an active issues page for interacting with development teams.

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. As a previous user of EF5 and EF6.x, I can see where EF Core has been improved, as well as where it's still missing features of the old EF6.x library that I liked (although those features are on the roadmap).

This book is intended for both software developers who've never used Entity Framework and seasoned EF6.x developers, plus anyone who wants to know what EF Core is capable of. I do assume that you're familiar with .NET development with 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: www.w3schools.com/sql/sql_intro.asp. The SQL set of commands is vast, and EF Core queries use only a small subset (for example, `SELECT`, `WHERE`, and `INNER JOIN`), so that's 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 is split into three parts. In addition to this chapter, part 1 has four other chapters that cover:

- Querying the database with EF Core
- Updating the database with EF Core (creating, updating, and deleting data)
- Using EF Core in business logic
- Building an ASP.NET Core web application that uses EF Core

By the end of part 1, you should be able to build a .NET application that uses a relational database. But the way the database is organized is left to EF Core; for instance, EF Core's default configuration sets the type and size of the database columns, which can be a bit wasteful on space.

Part 2 covers how and why you can change the defaults, and looks deeper into some of the EF Core commands. After part 2, you'll be able to use EF Core to create a database in exactly the way you want it, or link to an existing database that has a specific schema, or design. In addition, by using some of EF Core's advanced features, you can change the way the database data is exposed inside your .NET application—for instance, controlling software access to data more carefully or building code to automatically track database changes.

Part 3 is all about improving your skills and making you a better developer, and debugger, of EF Core applications. I present real-world applications of EF Core, starting with a range of known patterns and practices that you can use. You'll read chapters on unit testing EF Core applications, extending EF Core, and most important, finding and fixing EF Core performance issues.

1.2 My "lightbulb moment" with Entity Framework

Before we get into the nitty-gritty, let me tell you 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 work 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 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 had applied good software practices and made sure 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 into 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 10.)

As a result of my review, I built a library called GenericServices (https://github.com/JonPSmith/GenericServices) to use with EF6.x. This 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 Light 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
= very quick development of database access code

The site isn't that pretty, but that wasn't the point. My GenericServices library allowed me to quickly implement a whole range of database Create, Read, Update, and Delete (CRUD) commands. Definitely a "lightbulb moment," and I was hooked on EF. You can find the site at http://complex.samplemvcwebapp.net/.

Since then, I've built other libraries, some open source and some private, and used them on several projects. These libraries significantly speed up the development of 90% of database accesses, leaving me to concentrate on the harder topics, such as building great frontend interfaces, writing custom business logic to meet the client's specific requirements, and performance tuning where necessary.

1.3 Some words for existing EF6.x developers

TIME-SAVER If you're new to Entity Framework, you can skip this section.

If you're a reader who knows EF6.x, much of EF Core will be familiar to you. To help you navigate quickly through this book, 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. They 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 became a bit of a problem at the start of 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 approach is similar, but in some areas, it isn'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

EF Core can be used 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 most of this for you, but your nice .NET classes get "polluted" by these keys, and their values matter. In most cases, EF Core is going to work fine, but sometimes you need to do things a little differently to a software-only solution to suit the database. One example you'll see in chapter 2 is a many-to-many relationship: easy in C#, but a bit more work in a database.

The second issue is that an O/RM—and especially an O/RM as comprehensive as EF Core—hides the database so well that you can sometimes forget about what's going on underneath. This problem can cause you to write code that works great in your test application, but performs terribly in the real world when the database is complex and has many simultaneous users.

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, know what to do when it 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 13 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.

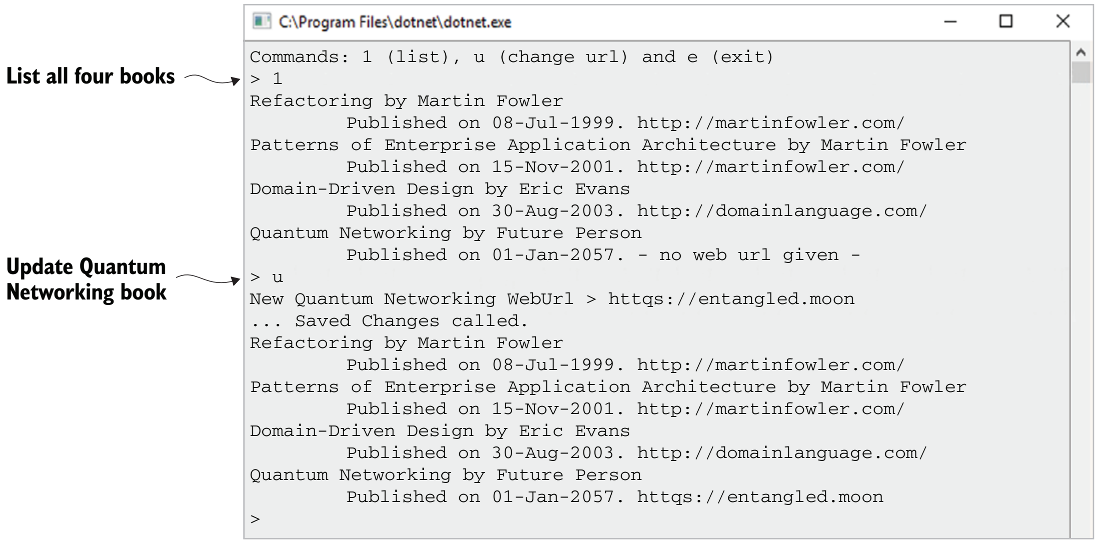
EF Core is designed to handle both relational and nonrelational databases—a departure from EF6.x, which was designed around relational databases only. Many of the principles covered in this book apply to both types of databases, but because relational databases are inherently much more complex than nonrelational databases, more commands are needed to use relational databases. You'll see whole chapters dedicated to commands that are used only in a relational database. Chapter 7, for instance, is all about modeling database relationships.

EF Core 2.0 will contain a preview database provider for the Azure NoSQL database, Cosmos DB. The aim is to use this as a learning exercise for handling NoSQL databases, with a robust solution coming out in EF Core 2.2. More NoSQL database providers are likely to be written for EF Core over time, either by Microsoft or the writers of NoSQL databases.

NOTE In section 14.2, you'll build an application using both an SQL/relational database and a NoSQL database in a Command Query Responsibility Segregation (CQRS) architectural pattern to get a higher-performing application.

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, 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.


C:\Program Files\dotnet\dotnet.exe
List all four books
Update Quantum Networking book

```
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
>
```

Figure 1.1 The output from the console application you'll use to look at how EF Core works

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 in order to help you understand what's going on later in this book.

You can download this example application from the Chapter01 branch of the Git repo at http://mng.bz/KTjz. 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

You can use two main development tools to develop a .NET Core application: Visual Studio 2017 (VS 2017) or Visual Studio Code (VS Code). I describe using VS 2017 for your first application, because it's slightly easier to use for newcomers to .NET development.

You need to install Visual Studio 2017 (VS 2017) from www.visualstudio.com. Numerous versions exist, including a free community version, but you need to read the license to make sure you qualify; see www.visualstudio.com/vs/community/.

When you install VS 2017, make sure you include the .NET Core Cross-Platform Development feature, which is under the Other Toolsets section during the Install Workloads stage. This installs .NET Core on your system. Then you're ready to build a .NET Core application. See http://mng.bz/2x0T for more information.

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

I know 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 Core console application MyFirstEfCoreApp by using Visual Studio 2017.

Creating a .NET Core console application

The first thing you need to do is create a .NET Core console application. Using VS 2017, here are the steps:

1. In the top menu of VS 2017, click File > New > Project to open the New Project form.
2. From the installed templates, select Visual C# > .NET Core > Console App (.NET Core).
3. Type in the name of your program (in this case, `MyFirstEfCoreApp`) and make sure the location is sensible. By default, VS 2017 will put your application in a directory ending with \Source\Repos.
4. Make sure the Create Directory for Solution box is ticked so that your application has its own folder.
5. If you want to create a Git repo for this project, make sure the Create New Git Repository box is selected too. Then click OK.

At this point, you've created a console application, and the editor should be in the file called Program.cs.

TIP You can find out which level of .NET Core your application is using by choosing Project > MyFirstEfCoreApp Properties from the main menu; the Application tab shows the Target Framework.

Adding the EF Core library to your application

You need to install the correct EF Core NuGet library for the database you're going to use. For local development, Microsoft.EntityFrameworkCore.SqlServer is the best choice, because it'll use the development SQL Server that was installed when you installed VS 2017.

You can install the NuGet library in various ways. The more visual way is to use the NuGet Package Manager. The steps are as follows:

1. In the Solution Explorer, typically on the right-hand side of VS 2017, right-click the Dependencies line in your console application and select the Manage NuGet Packages option.
2. At the top right of the NuGet Package Manager page that appears, click the Browse link.
3. In the Search box below the Browse link, type `Microsoft.EntityFramework-Core.SqlServer` and then select the NuGet package with that name.
4. A box appears to the right of the list of NuGet packages with the name Microsoft.EntityFrameworkCore.SqlServer at the top and an Install button below it, showing which version will install.
5. Click the Install button and then accept the license agreements. The package installs. Installation could take a little while, depending on your internet connection speed.

Downloading and running the example application from the Git repo

You have two options for downloading and running the MyFirstEfCoreApp console application found in the Git repo: either VS 2017 or VS Code. I describe both.

Using Visual Studio 2017, version 15.3.3 or above (VS 2017), follow these steps:

1. *Clone the Git repo.* First you need to select the Team Explorer view and select the Manage Connections tab. In the Local Git Repositories section, click the Clone button. This opens a form containing an input line saying "Enter the URL of a Git repo to clone" in which you should input the URL https://github.com/JonPSmith/EfCoreInAction. The local directory path shown below the URL should update to end with EfCoreInAction. Now click the Clone button at the bottom of the form.
2. *Select the right branch.* After the clone has finished, the list of local Git repositories should have a new entry called EfCoreInAction. Double-click this, and the Home tab appears. Currently, the Git repo will be on the master branch, which doesn't have any code. You need to select the remotes/origin > Chapter01 branch: click the Branches button, click the Remotes/Origin drop-down, and select Chapter01. Next, click the Home button. You'll see a Solution called EfCoreInAction.sln, which you need to click. That loads the local solution, and you're ready to run the application.
3. *Run the application.* Go to the Solutions Explorer window, which shows you the code. Click any of the classes to see the code. If you press F5 (Start Debugging), the console application will start in a new command-line window. The first line shows you the commands you can type. Have fun!

Using Visual Studio Code (VS Code), follow these steps:

Note: I assume that you've set up VS Code to support C# development.

1. *Clone the Git repo.* In the command palette (Ctrl-Shift-P), type `Git: Clone`. This presents you with a Repository Url input line, in which you should place the https://github.com/JonPSmith/EfCoreInAction URL and then press the Return key. You'll then see a Parent Directory input line; indicate the directory that will contain the Git repo and then press the Return key. This clones the Git repo to your local storage, in a directory called EfCoreInAction.
2. *Select the right branch.* After the clone, you'll see a message asking, "Would you like to open the cloned repository?" Click the Open Repository button to do that. You should see just a few files in the master branch, but no code. Select the Chapter01 branch by typing `Git: Checkout to` in the command palette (Ctrl-Shift-P) and selecting the origin/Chapter01 branch. The files change, and you'll now have the code for the MyFirstEfCoreApp console application.
3. *Run the application.* I've already set up the tasks.json and launch.json files for this project, so you can press F5 to start debugging. The console application starts in a new command-line window. The first line shows the commands you can type. Have fun!

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, known as *code-first*, or you can provide an existing database you built outside EF Core, known as *database-first*.

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

In this chapter, we're going to skip over how I created the database for the MyFirstEf-CoreApp application and simply assume it exists.

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 2 covers how to get EF Core to create a database properly, and chapter 11 presents the whole issue of creating and changing databases.

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

NOTE The Books table name comes from the `DbSet<Book>` property name of `Books` in the application's DbContext, which I show in figure 1.5. The Author table name doesn't have a `DbSet<T>` property in the application's DbContext, so the table defaults to the class name, `Author`. Section 6.10.1 covers these configuration rules in more detail.

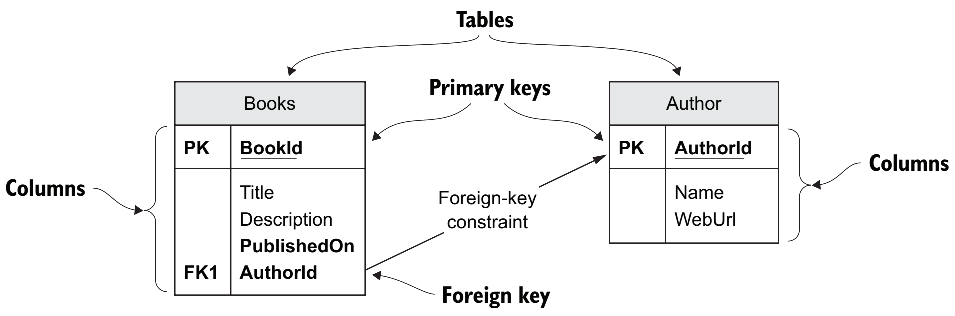
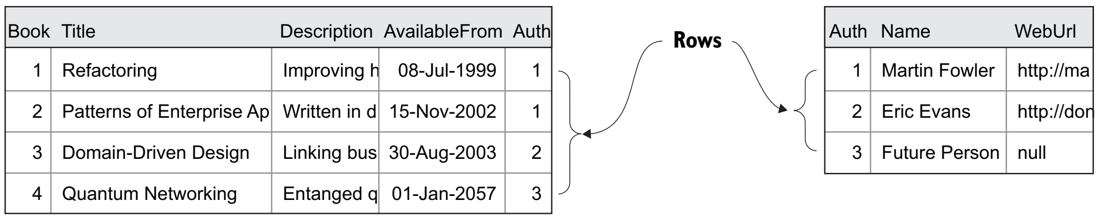


Figure 1.2 Our example relational database with two tables: Books and Author

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://dor
3	Future Person	null

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

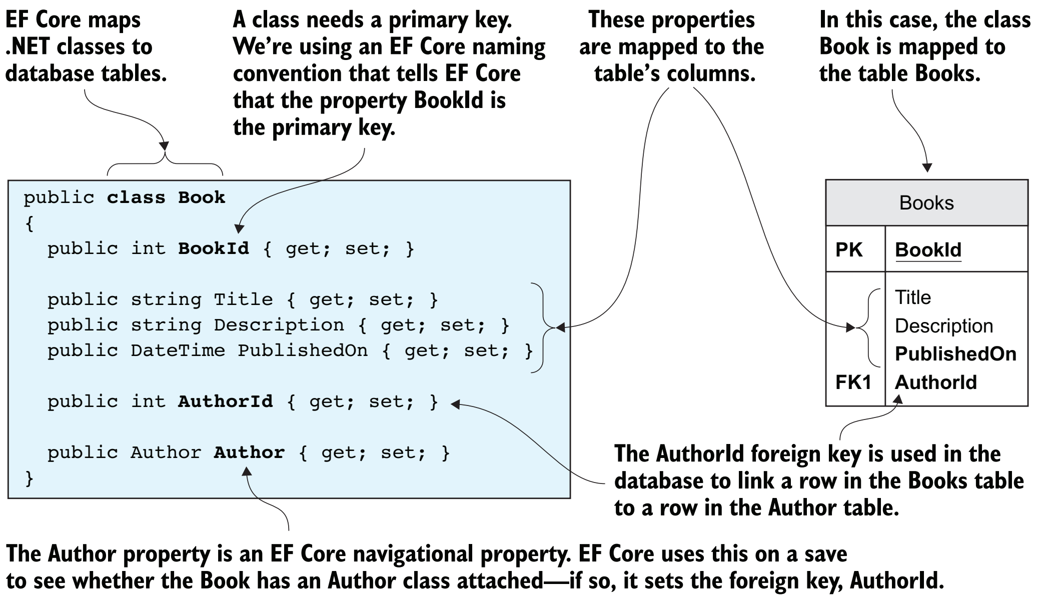
1.8 Setting up the MyFirstEfCoreApp application

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

1. The classes that you want EF Core to map to the tables in your database
2. 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 later in the book), but figure 1.4 gives the typical format of a class that's mapped to a database table.



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

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

  public int AuthorId { get; set; }

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

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.
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.

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 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 6.3.15). The `Book` class has a property called `AuthorId`, which EF Core knows is a foreign key because it has the same name as the `Author` primary key.

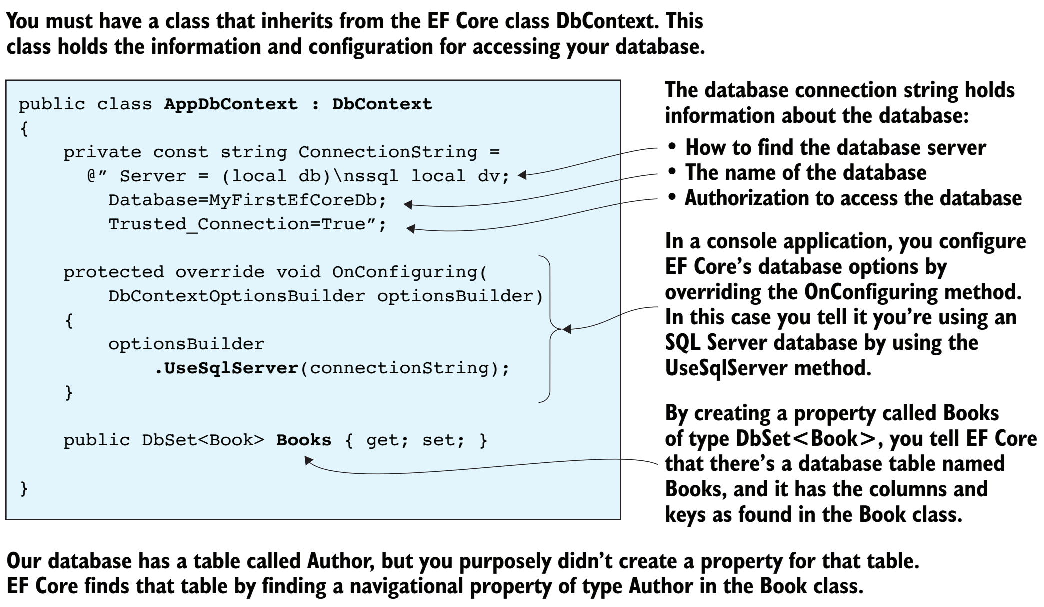
Listing 1.1 The `Author` class from MyFirstEfCoreApp

```
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 its DbContext. This is a class that you create that inherits from EF Core's `DbContext` class. This 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 = (local db)\nssql local dv;
        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 to define what type of database to use and where it can be found. Second, the `DbSet<T>` property(s) that tell 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 both chapter 6 and chapter 7 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 is a bigger problem, 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 built your 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 with a mental model of what happens when a database access code uses EF Core. This should help as you dig into 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 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. The resulting model is then used by EF Core 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). This 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 more precisely configure your database, but for now you'll use the default configurations.

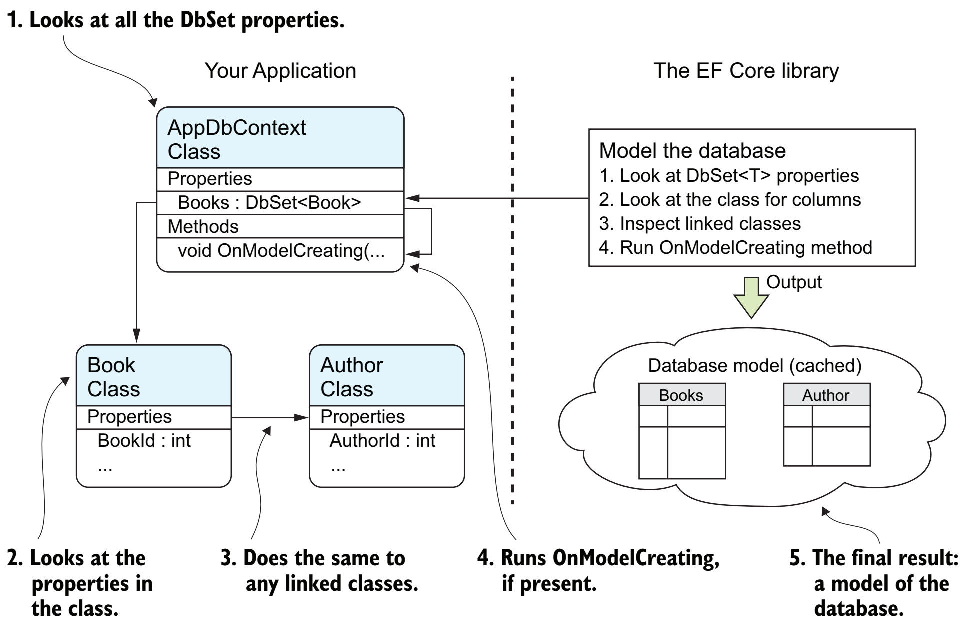

1. Looks at all the DbSet properties.
Your Application
The EF Core library
AppDbContext Class
Properties
Books : DbSet<Book>
Methods
void OnModelCreating(...
Model the database
1. Look at DbSet<T> properties
2. Look at the class for columns
3. Inspect linked classes
4. Run OnModelCreating method
Output
Book Class
Properties
BookId : int
...
Author Class
Properties
AuthorId : int
...
Database model (cached)
Books
Author
2. Looks at the properties in the class.
3. Does the same to any linked classes.
4. Runs OnModelCreating, if present.
5. The final result: a model of the database.

Figure 1.6 How EF Core models the database

Figure 1.6 shows the modeling steps that EF Core uses on our `AppDbContext`. The following text gives a more detailed description of the process:

1. EF Core looks at the application's DbContext and finds all the public `DbSet<T>` properties. From this, it defines the initial name for the one table it finds, Books.
2. 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.
3. 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 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.
4. 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.

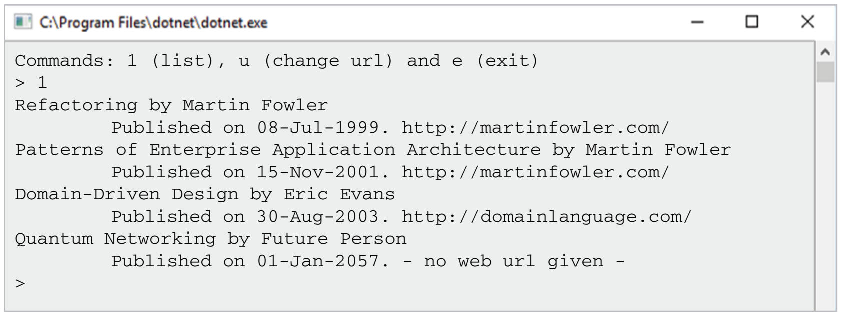
5. EF Core creates an internal model of the database based on all the information it gathered. This database model is cached so that later accesses will be quicker. This model is then used when performing all database accesses.

You might have noticed that figure 1.6 shows no database. This is because when EF Core is building its internal model, it doesn't look at the database. I emphasize that 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, which are detailed in chapter 11.

1.9.2 Reading data from the database

You're now at the point where you can access the database. Let's use the list (`l`) command, which reads the database and prints the information on the terminal. Figure 1.7 shows the result.



C:\Program Files\dotnet\dotnet.exe

```
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 -
>
```

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(a => a.Author))
        {
            var webUrl = book.Author.WebUrl == null
```

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

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

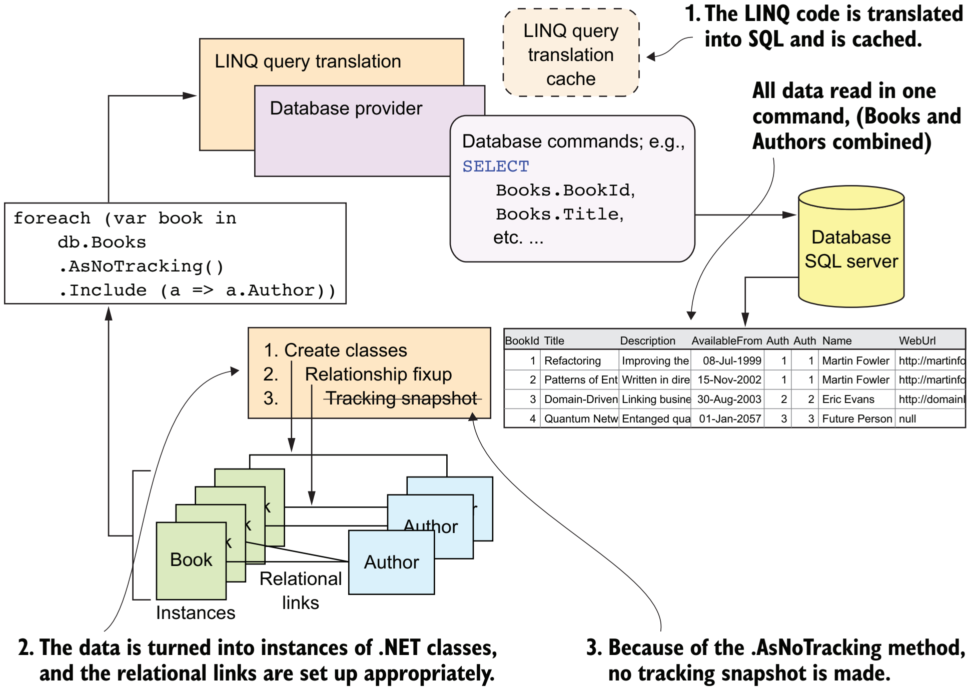
The "include" causes the author information to be eagerly loaded with each book. See chapter 2 for more on this.

```
                    ? "- 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}");
            }
        }
    }
```

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 includes minimal use of LINQ, but later in the book you'll see much more complex examples.

> **NOTE** If you're not familiar with LINQ, you'll be at a disadvantage in reading this book. Appendix A provides a brief introduction to LINQ. Plenty of online resources are also available; see https://msdn.microsoft.com/en-us/library/bb308959.aspx.

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.


LINQ query translation
Database provider
LINQ query translation cache
1. The LINQ code is translated into SQL and is cached.
All data read in one command, (Books and Authors combined)
Database commands; e.g., SELECT Books.BookId, Books.Title, etc. ...
Database SQL server
foreach (var book in db.Books .AsNoTracking() .Include (a => a.Author))
1. Create classes
2. Relationship fixup
3. ~~Tracking snapshot~~
Book
Author
Relational links
Instances
2. The data is turned into instances of .NET classes, and the relational links are set up appropriately.
3. Because of the .AsNoTracking method, no tracking snapshot is made.

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

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

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

1 The LINQ query `db.Books.AsNoTracking().Include(a => a.Author)` accesses the `DbSet<Book>` property in the application's DbContext and adds a `.Include (a => a.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.

Listing 1.3 SQL command produced to read Books and Author

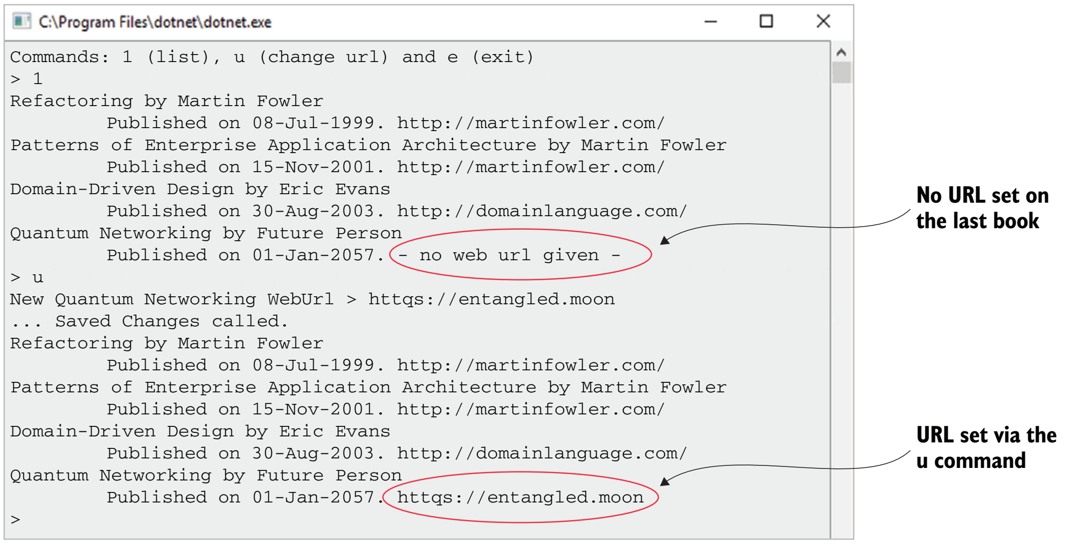
```
SELECT [b].[BookId],
[b].[AuthorId],
[b].[Description],
[b].[PublishedOn],
[b].[Title],
[a].[AuthorId],
[a].[Name],
[a].[WebUrl]
FROM [Books] AS [b]
INNER JOIN [Author] AS [a] ON
[b].[AuthorId] = [a].[AuthorId]
```

2 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 together by reference—called a *relationship fixup*. The result is a set of .NET class instances linked in the correct way. In this example, two books have the same author, Martin Fowler, so the `Author` property of those two books points to one `Author` class.

3 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; you'll see this in the example of editing the WebUrl. Because this is a read-only query, 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. You then run the command `u`, which asks for a new author URL for the last book, *Quantum Networking*. You input a new URL of `httqs://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 (the two ovals show you the before and after URLs).


C:\Program Files\dotnet\dotnet.exe
No URL set on the last book
URL set via the u command

```
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
>
```

Figure 1.9 The book information before and after the WebUrl of the last book's author is updated

The code for updating the WebUrl of the last book, *Quantum Networking*, is shown here.

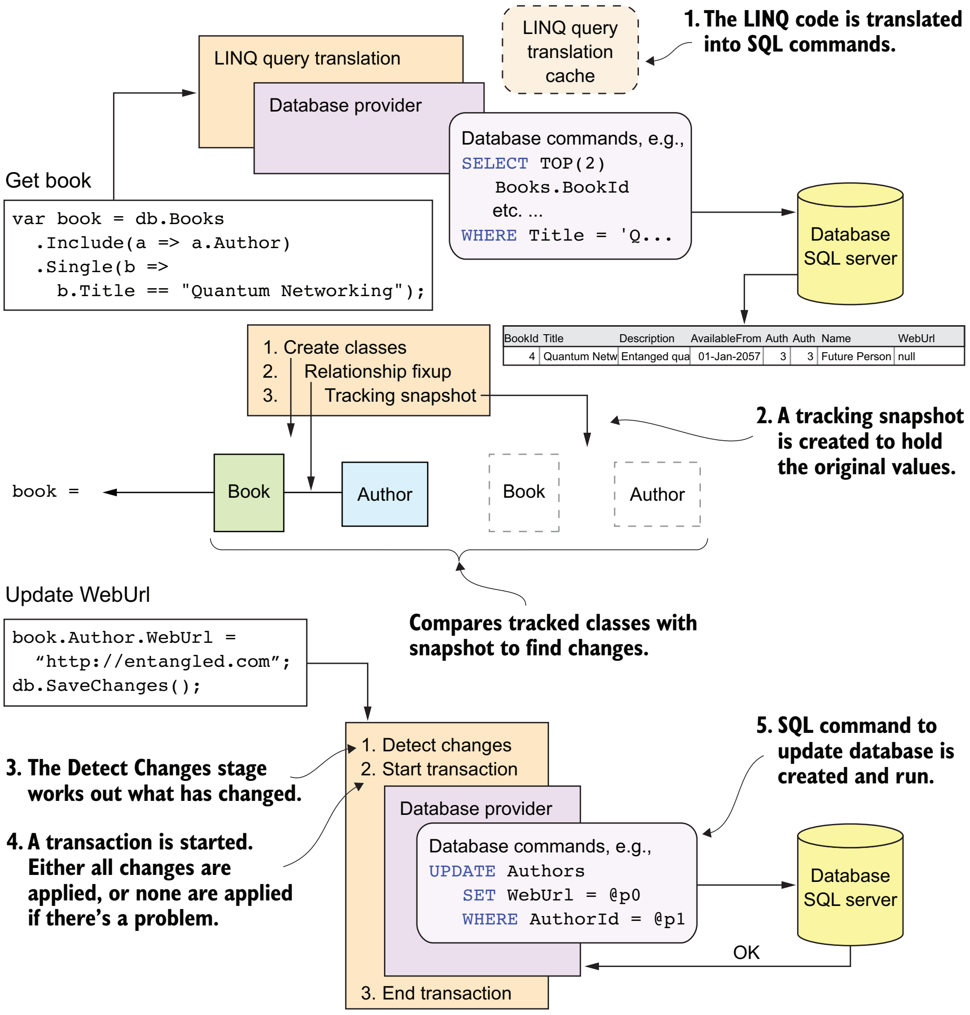
Listing 1.4 The code to update the author's WebUrl of the book *Quantum Networking*

```
public static void ChangeWebUrl()
{
    Console.Write("New Quantum Networking WebUrl > ");
    var newWebUrl = Console.ReadLine();          // Reads in from the console the new URL

    using (var db = new AppDbContext())
    {
        var book = db.Books
            .Include(a => a.Author)               // Makes sure the author information is eager loaded with the book
            .Single(b => b.Title == "Quantum Networking");   // Selects only the book with the title Quantum Networking

        book.Author.WebUrl = newWebUrl;           // To update the database, you change the data that was read in.
        db.SaveChanges();                         // 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.
        Console.WriteLine("... SavedChanges called.");
    }

    ListAll();                                    // Lists all the book information
}
```

Figure 1.10 shows what is happening inside the EF Core library and follows its progress. This 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, it sees that only the WebUrl has been updated, and from that it can create an SQL command to update only that column in the right row.



Get book

```
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

Update WebUrl

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

Database commands, e.g.,

```
UPDATE Authors
    SET WebUrl = @p0
    WHERE AuthorId = @p1
```

LINQ query translation
Database provider
LINQ query translation cache
1. The LINQ code is translated into SQL commands.
Database SQL server
1. Create classes
2. Relationship fixup
3. Tracking snapshot
2. A tracking snapshot is created to hold the original values.
book =
Book
Author
Book
Author
Compares tracked classes with snapshot to find changes.
1. Detect changes
2. Start transaction
Database provider
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 SQL server
OK

Figure 1.10 A look inside EF Core as it executes and reads, followed by a database update

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.
3. The code then 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, just the `WebUrl` property of the `Author` class that 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, they either all succeed, or they all fail. This is important, because a relational database could get into a bad state if only part of an update was applied.
5. The update request is converted by the database provider into 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 Should you use EF Core in your next project?

Having given you 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 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 our next project?" A cost is associated with learning and adopting any new library, especially complex libraries such as EF Core, so it's a valid question.

I'll give you a detailed answer, but as you can see, I think visually. Figure 1.11 captures my view of EF Core's strengths and weaknesses: good things to the right, and not-so-good to the left. The width of each block shows the time period over which I think that topic will improve—the wider the block, the longer this will take. It's only my view, so don't take it as the truth, especially if you're reading this book some time after I wrote this section. I hope that it at least helps you to think through the issues that affect your using EF Core in your project.

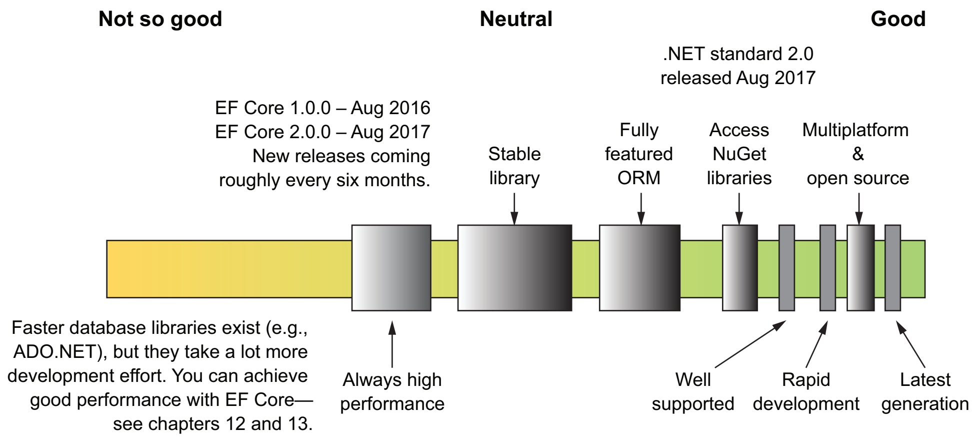

Not so good
Neutral
Good
.NET standard 2.0 released Aug 2017
EF Core 1.0.0 – Aug 2016
EF Core 2.0.0 – Aug 2017
New releases coming roughly every six months.
Stable library
Fully featured ORM
Access NuGet libraries
Multiplatform & open source
Faster database libraries exist (e.g., ADO.NET), but they take a lot more development effort. You can achieve good performance with EF Core—see chapters 12 and 13.
Always high performance
Well supported
Rapid development
Latest generation

Figure 1.11 My view of the strengths and weaknesses of EF Core

Let me give you more details about each of the blocks in figure 1.11, starting with the good stuff on the right.

1.10.1 Latest generation

I swapped from Microsoft's LINQ to SQL O/RM, which I liked, to EF4 because EF was the future, and no further effort was being put into LINQ to SQL. It's the same now for EF Core. It's where Microsoft is putting its effort, and it's going to be extended and well supported for many years. EF Core is much more lightweight and generally faster than EF6.x, and I think the improvements in its API are good.

If you're starting a new project, and .NET Core and EF Core have the necessary features your project needs, then moving to EF Core means you won't be left behind.

1.10.2 Multiplatform and open source

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. EF Core is also open source, so you have access to the source code and an open list of issues and defects—see https://github.com/aspnet/EntityFramework/issues.

1.10.3 Rapid development

In a typical data-driven application, I write a lot of database access code, some of it complex. I've found that EF6.x, and now EF Core, allow me to write data access code quickly, and in a way that's easy to understand and refactor. This is one of the main reasons I use EF.

EF Core also is developer-friendly, and tends to create working queries even if I didn't write the most efficient code. Most properly formed LINQ queries work, though maybe they won't produce the best-performing SQL—and having a query that works is a great start. Chapter 12 covers the whole area of performance tuning.

1.10.4 Well supported

EF Core has good documentation (https://docs.microsoft.com/en-us/ef/core/index) and, of course, you now have this book, which brings together the documentation with deeper explanations and examples, plus patterns and practices to make you a great developer. Because a large group of EF6.x developers will migrate to EF Core, the internet will be full of blogs on EF Core, and Stack Overflow is likely to have the answers to your problems already.

The other part of support is the development tools. Microsoft seems to have changed focus by providing support for multiple platforms, but also has created a cross-platform development environment that's free—called Visual Studio Code (https://code.visualstudio.com/). Microsoft has also made its main development tool, Visual Studio, free for 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.

1.10.5 Access to NuGet libraries

Although some early difficulties arose with .NET Core 1, the introduction of .NET Standard 2.0 in August 2017, with its *.NET Framework compatibility mode*, overcame much of this, which is what EF Core 2.0 is built on. .NET Standard 2.0 allows (most) existing NuGet libraries that use earlier .NET versions to be used. The only problem occurs if the NuGet package uses an incompatible .NET feature, such as `System.Reflection`. .NET Standard 2.0 also supports a much bigger range of system methods, which makes it easier to convert a package to .NET Standard 2.0.

NOTE If you want to stay on .NET 4.x, you can still use EF Core if you upgrade to .NET 4.6.1 or higher. For more information, see http://mng.bz/sB0y.

1.10.6 Fully featured O/RM

Entity Framework in general is a feature-rich implementation of an O/RM, and EF Core continues this trend. It allows you to write complex data access code covering most of the database features you'll want to use. As I have moved through ADO.NET, LINQ to SQL, EF 4 to 6, and now EF Core, I believe this is already a great O/RM.

But, at the time of writing this book, EF Core (version 2.0) still has some features yet to be added. That's why the block is so wide in figure 1.11. If you're a user of EF6.x, you'll notice that some features available in EF6.x aren't yet available in EF Core, but as time goes on, these will appear. I suggest you look at the Feature Comparison page on the EF Core docs site, http://mng.bz/ek4D, for the latest on what has been implemented.

1.10.7 Stable library

When I started writing this book, EF Core wasn't stable. It had bugs and missing features. I found an error on using the year part of a `DateTime` in the version 1.0.0 release, along with a whole load of other LINQ translation issues that were fixed in 1.1.0.

By the time you read this, EF Core will be much better, but still changing, albeit at a much slower rate. If you want something stable, EF6.x is a good O/RM, or there are other database access technologies. The choice is yours.

1.10.8 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 13

But you can do something about it. In my applications, I find only about 5% to 10% of my queries are the key ones that need hand-tuning. Chapters 12 and 13 are dedicated to performance tuning, plus part of chapter 14. These show that there's a lot you can do to improve the performance of EF Core database accesses.

If you're worried about EF Core's performance, I recommend you skim through chapter 13, where you'll progressively improve the performance of an application. You'll see that you can make an EF Core application perform well with little extra effort. I also have two live demo sites, http://efcoreinaction.com/ and http://cqrsravendb.efcoreinaction.com/; click the About menu to see how big the databases are.

1.11 When should you not use EF Core?

I'm obviously pro EF Core, but I won't use it on a client project unless it makes sense. So, let's look at a few blockers that might suggest you don't use 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 that needs to be quick and has a small number of database accesses, then EF Core isn't a good fit; you could use a fast, but development-time-hungry library because there isn't much to write. But if you have a large application, with lots of boring admin accesses and a few important customer-facing accesses, then a hybrid approach could work for you (see chapter 13 for an example of a mixed EF Core/Dapper application).

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 into 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. This will help you understand what the EF Core commands described in later chapters can do.

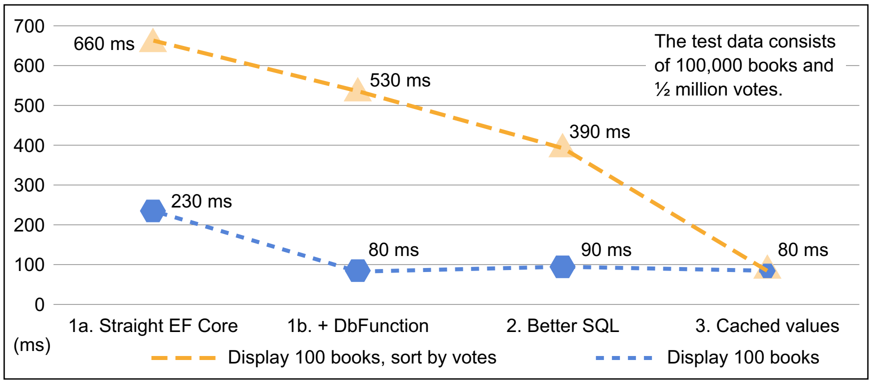
- There are many good reasons to consider using EF Core: it's built on a lot of experience, is well supported, and runs on multiple platforms.
- At the time this book was written, EF Core was at version 2.0 with added notes about the next release, EF Core 2.1. Some features that you might want may not be out yet, so check the online documentation for the latest state (https://docs.microsoft.com/en-us/ef/core/index).

For readers who are familiar with EF6.x:

- Look for EF6 notes throughout the book. They 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 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.

EFC Core performance issue checklist: the section that discusses each issue is listed.

Speed performance issues	Section
Have you picked the right feature to performance tune?	12.1.2
Are you loading too many columns?	12.4.1
Are you loading too many rows?	12.4.2
Are you using lazy loading?	12.4.3
Are you telling EF Core that your query is read-only?	12.4.4
Are you making too many calls to the database?	12.5.1
Are you calling SaveChanged multiple times?	12.5.2
Is part of your query being run in software?	12.5.3
Could you improve the SQL with a DbFunction?	12.5.4
Could pre-compiled queries help?	12.5.5
Have you checked the SQL that EF Core has produced?	12.5.6
Are you using the Find method to load via primary key?	12.5.7
Would an index help with sorting or filtering?	12.5.8
Do you have a mismatch on database types?	12.5.9
Are you making Detect Changes work too hard?	12.6.1
Would turning one DbContext into multiple DbContexts help?	12.6.2



Worked example of performance improvement with four stages, from Chapter 13

MICROSOFT .NET

Entity Framework Core IN ACTION

Jon P Smith

There's a mismatch in the way OO programs and relational databases represent data. Entity Framework is an object-relational mapper (ORM) that bridges this gap, making it radically easier to query and write to databases from a .NET application. EF creates a data model that matches the structure of your OO code so you can query and write to your database using standard LINQ commands. It will even automatically generate the model from your database schema.

Using crystal-clear explanations, real-world examples, and around 100 diagrams, **Entity Framework Core in Action** teaches you how to access and update relational data from .NET applications. You'll start with a clear breakdown of Entity Framework, along with the mental model behind ORM. Then you'll discover time-saving patterns and best practices for security, performance tuning, and even unit testing. As you go, you'll address common data access challenges and learn how to handle them with Entity Framework.

What's Inside

- Querying a relational database with LINQ
- Using EF Core in business logic
- Integrating EF with existing C# applications
- Applying domain-driven design to EF Core
- Getting the best performance out of EF Core
- Covers EF Core 2.0 and 2.1

For .NET developers with some awareness of how relational databases work.

Jon P Smith is a full-stack developer with special focus on .NET Core and Azure.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/entity-framework-core-in-action

Free eBook

See first page

"An expertly written guide to EF Core—quite possibly the only reference you'll ever need."

—Stephen Byrne, Action Point

"A solid book that deals well with the topic at hand, but also handles the wider concerns around using EF in real-world applications."

—Sebastian Rogers
Simple Innovations

"This is the next step beyond the basics. It'll help you get to the next level!"

—Jeff Smith, Agilify Automation

"Great book with excellent, real-world examples."

—Tanya Wilke, Sanlam

ISBN-13: 978-1-61729-456-3
ISBN-10: 1-61729-456-X
54999
9 781617 294563

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]