# PART 1 Getting started

# 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-word 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 databases access code quickly.

EF Core, which Microsoft released in 2016, it 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 EF 6.x. EF Core starts with years of experience build into it via feedback from these previous version, 4 to 6.x. It has kept the same type of interface as EF 6.x but has major changes underneath, such as the ability to handle nonrelational databases, which EF 6.x wasn’t designed to do. As a previous user of EF5 and EF 6.x, I can see where EF Core has been improved, as well as where it’s still missing features of the old EF 6.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 work and seasoned EF 6.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 database 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 W3School online resource: [www.w3schools.com/sql/sql\_intro.asp](http://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 EF Core will help you as you read through the rest of the book.

## 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 use 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 you .NET application – for instance, controlling software access to data more carefully or building code to automatically trace 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, staring 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.

## 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-years gap (that’s a story in itself!).

My wife, Dr. Honora Smith, is lecturer in mathematics at the University of Southampton who specializes in the modeling of the healthcare systems, especially focusing on where to locate health facilities. I had worked with here 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 chapter2.)
* *Was repetitious* – I use 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 chatper2).
* *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 chapter 10.)

As a result of my review, I build a library called GenericServices (<http://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 build 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 hook 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 access, leaving me to concentrate on the harder topics, such as building grate fronted interfaces, writing custom business logic to meet the client’s specific requirements, and performance tuning where necessary.

## 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 EF 6.x well, but that became a bit of a problem at the start of using EF Core. I was using an EF 6.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 EF 6.x develop is to approach EF Core as a new library that someone has written to mimic EF 6.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.

## 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 collations – 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 => … |

### The downsides of O/RMs

Making a good O/RM is complex. Although EF 6.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 you 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 litter 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 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 boo, 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 show how to find and improve database code that isn’t fast enough.

## 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 Table (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 EF 6.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 relational ships.

EF Core 2.0 will contain a preview database provided for the Azure NoSQL database, CosmosDB. The aim is to use this as a learning exercise for handing 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 database.

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.

## 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 access 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.



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 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.

### 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](http://www.visualstudio.com). Numerous version exists, including a free community version, but you need to read the license to make sure you qualify; see [www.visualstudio.com/vs/community/](http://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 .NET Core application. See <http://mng.bz/2x0T> for more information.

### 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.

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 Greater 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, Microsft.EntityFrameworkCore.SqlServer is the best choice, because it’ll use the development SQL Server that was installed when you install VS 2017.

You can install 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 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 Microsft.EntityFrameworkCore.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 Microsft.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 button of the form. |

## 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 build outside EF Core, known as *database-first*.

EF6 in EF6, you could use an EDMX/database designer to visual 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 MyFirstEfCoreApp application and simple assume it exists.

Note in my code, I use a basic EF Core command meant for unit testing to create 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 create a simple database, show 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 applications DbContext, so the table defaults to the class name, Author. Section 6.10.1 covers these configuration rules in more details.

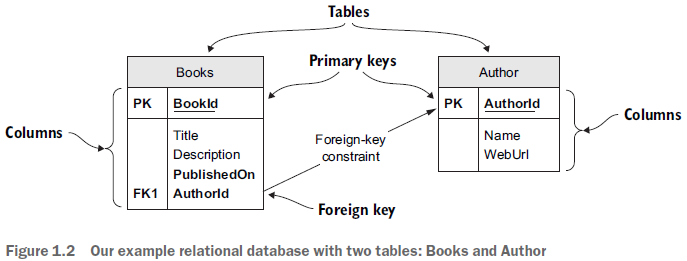
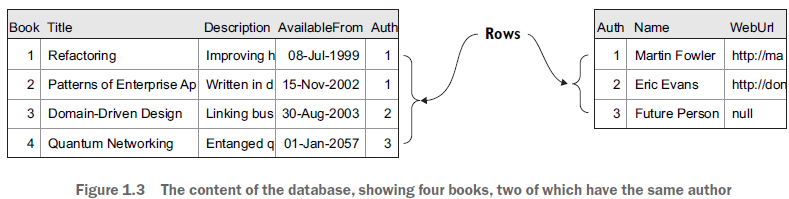


Figure 1.3 shows the content of the database. It holds only four books, the first two of which have the same author, Martin Fowler.



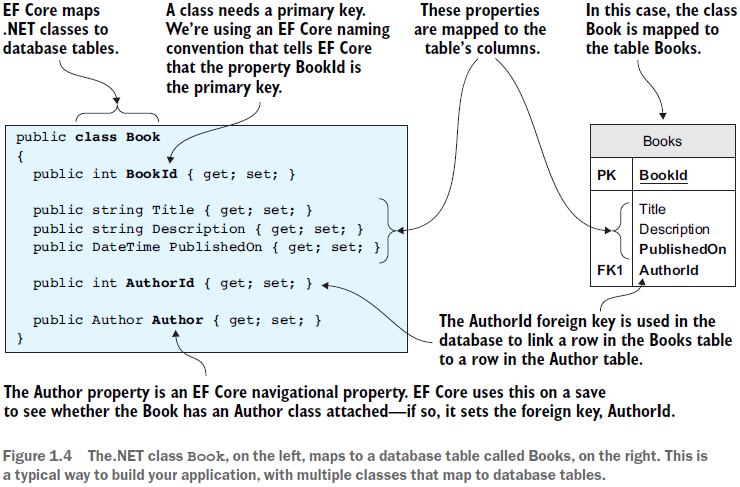
## Setting up the MyFirstEfCoreApp application

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

### 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 database table.



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 conversions of <ClassName>Id (see section 6.3.15). The Book class has a property called AuthorId, which EF Core knowns is a foreign key because it has the same name as the Author primary key.

### The application’s DbContext

## Looking under the hood of EF Core

### Modeling the database

### Reading data from the database

### Updating the database

## Should you use EF Core in your next project?

### Latest generation

### Multiplatform and open source

### Rapid development

### Well supported

### Access to NuGet libraries

### Fully featured O/RM

### Stable library

### Always high-performance

## When should you use EF Core?