Shapes should have a read-only property named Area so that when you deserialize, you can output a list of shapes, including their areas, as shown here:

```
List<Shape> loadedShapesXml =
  serializerXml.Deserialize(fileXml) as List<Shape>;

foreach (Shape item in loadedShapesXml)
{
  WriteLine("{0} is {1} and has an area of {2:N2}",
    item.GetType().Name, item.Colour, item.Area);
}
```

This is what your output should look like when you run your console application:

```
Loading shapes from XML:
Circle is Red and has an area of 19.63
Rectangle is Blue and has an area of 200.00
Circle is Green and has an area of 201.06
Circle is Purple and has an area of 475.29
Rectangle is Blue and has an area of 810.00
```

Exercise 9.3 – Explore topics

Use the links on the following page to learn more detail about the topics covered in this chapter:

https://github.com/markjprice/cs10dotnet6/blob/main/book-links.md#chapter-9---working-with-files-streams-and-serialization

Summary

In this chapter, you learned how to read from and write to text files and XML files, how to compress and decompress files, how to encode and decode text, and how to serialize an object into JSON and XML (and deserialize it back again).

In the next chapter, you will learn how to work with databases using Entity Framework Core.

10

Working with Data Using Entity Framework Core

This chapter is about reading and writing to data stores, such as Microsoft SQL Server, SQLite, and Azure Cosmos DB, by using the object-to-data store mapping technology named **Entity Framework Core** (**EF Core**).

This chapter will cover the following topics:

- Understanding modern databases
- Setting up EF Core
- Defining EF Core models
- Querying EF Core models
- Loading patterns with EF Core
- Manipulating data with EF Core
- Working with transactions
- Code First EF Core models

Understanding modern databases

Two of the most common places to store data are in a **Relational Database Management System (RDBMS)** such as Microsoft SQL Server, PostgreSQL, MySQL, and SQLite, or in a **NoSQL** database such as Microsoft Azure Cosmos DB, Redis, MongoDB, and Apache Cassandra.

Understanding legacy Entity Framework

Entity Framework (EF) was first released as part of .NET Framework 3.5 with Service Pack 1 back in late 2008. Since then, Entity Framework has evolved, as Microsoft has observed how programmers use an **object-relational mapping (ORM)** tool in the real world.

ORMs use a mapping definition to associate columns in tables to properties in classes. Then, a programmer can interact with objects of different types in a way that they are familiar with, instead of having to deal with knowing how to store the values in a relational table or another structure provided by a NoSQL data store.

The version of EF included with .NET Framework is **Entity Framework 6 (EF6)**. It is mature, stable, and supports an EDMX (XML file) way of defining the model as well as complex inheritance models, and a few other advanced features.

EF 6.3 and later have been extracted from .NET Framework as a separate package so it can be supported on .NET Core 3.0 and later. This enables existing projects like web applications and services to be ported and run cross-platform. However, EF6 should be considered a legacy technology because it has some limitations when running cross-platform and no new features will be added to it.

Using the legacy Entity Framework 6.3 or later

To use the legacy Entity Framework in a .NET Core 3.0 or later project, you must add a package reference to it in your project file, as shown in the following markup:

```
<PackageReference Include="EntityFramework" Version="6.4.4" />
```

Good Practice: Only use legacy EF6 if you have to, for example, when migrating a WPF app that uses it. This book is about modern cross-platform development so, in the rest of this chapter, I will only cover the modern Entity Framework Core. You will not need to reference the legacy EF6 package as shown above in the projects for this chapter.

Understanding Entity Framework Core

The truly cross-platform version, **EF Core**, is different from the legacy Entity Framework. Although EF Core has a similar name, you should be aware of how it varies from EF6. The latest EF Core is version 6.0 to match .NET 6.0.

EF Core 5 and later only support .NET 5 and later. EF Core 3.0 and later only run on platforms that support .NET Standard 2.1, meaning .NET Core 3.0 and later. It does not support .NET Standard 2.0 platforms like .NET Framework 4.8.

As well as traditional RDBMSs, EF Core supports modern cloud-based, nonrelational, schemaless data stores, such as Microsoft Azure Cosmos DB and MongoDB, sometimes with third-party providers.

EF Core has so many improvements that this chapter cannot cover them all. I will focus on the fundamentals that all .NET developers should know and some of the cooler new features.

There are two approaches to working with EF Core:

1. **Database First**: A database already exists, so you build a model that matches its structure and features.
2. **Code First**: No database exists, so you build a model and then use EF Core to create a database that matches its structure and features.

We will start by using EF Core with an existing database.

Creating a console app for working with EF Core

First, we will create a console app project for this chapter:

1. Use your preferred code editor to create a new solution/workspace named `Chapter10`.
2. Add a console app project, as defined in the following list:
    1. Project template: **Console Application** / `console`
    2. Workspace/solution file and folder: `Chapter10`
    3. Project file and folder: `WorkingWithEFCore`

Using a sample relational database

To learn how to manage an RDBMS using .NET, it would be useful to have a sample one so that you can practice on one that has a medium complexity and a decent amount of sample records. Microsoft offers several sample databases, most of which are too complex for our needs, so instead, we will use a database that was first created in the early 1990s known as **Northwind**.

Let's take a minute to look at a diagram of the Northwind database. You can use the following diagram to refer to as we write code and queries throughout this book:

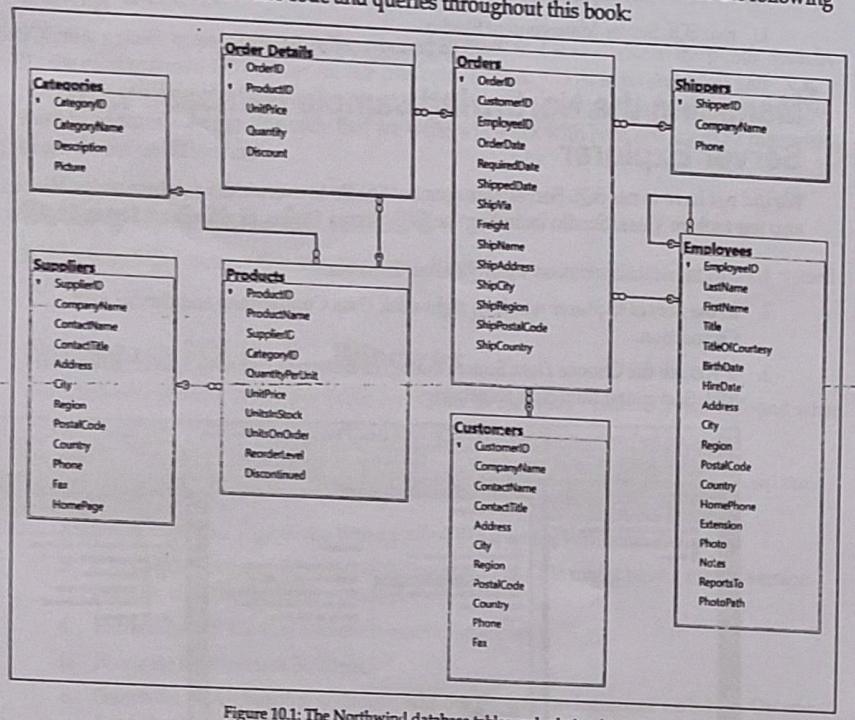

Figure 10.1: The Northwind database tables and relationships

You will write code to work with the `Categories` and `Products` tables later in this chapter and other tables in later chapters. But before we do, note that:

- Each category has a unique identifier, name, description, and picture.
- Each product has a unique identifier, name, unit price, units in stock, and other fields.
- Each product is associated with a category by storing the category's unique identifier.
- The relationship between `Categories` and `Products` is one-to-many, meaning each category can have zero or more products.

Using Microsoft SQL Server for Windows

Microsoft offers various editions of its popular and capable SQL Server product for Windows, Linux, and Docker containers. We will use a free version that can run standalone, known as SQL Server Developer Edition. You can also use the Express edition or the free SQL Server LocalDB edition that can be installed with Visual Studio for Windows.

If you do not have a Windows computer or you want to use a cross-platform database system, then you can skip ahead to the topic *Using SQLite*.

Downloading and installing SQL Server

You can download SQL Server editions from the following link:

`https://www.microsoft.com/en-us/sql-server/sql-server-downloads`

1. Download the **Developer** edition.
2. Run the installer.
3. Select the **Custom** installation type.
4. Select a folder for the installation files and then click **Install**.
5. Wait for the 1.5 GB of installer files to download.
6. In **SQL Server Installation Center**, click **Installation**, and then click **New SQL Server stand-alone installation or add features to an existing installation**.
7. Select **Developer** as the free edition and then click **Next**.
8. Accept the license terms and then click **Next**.
9. Review the install rules, fix any issues, and then click **Next**.
10. In **Feature Selection**, select **Database Engine Services**, and then click **Next**.
11. In **Instance Configuration**, select **Default instance**, and then click **Next**. If you already have a default instance configured, then you could create a named instance, perhaps called `cs10dotnet6`.
12. In **Server Configuration**, note the **SQL Server Database Engine** is configured to start automatically. Set the **SQL Server Browser** to start automatically, and then click **Next**.
13. In **Database Engine Configuration**, on the **Server Configuration** tab, set **Authentication Mode** to **Mixed**, set the **sa** account password to a strong password, click **Add Current User**, and then click **Next**.
14. In **Ready to Install**, review the actions that will be taken, and then click **Install**.
15. In **Complete**, note the successful actions taken, and then click **Close**.
16. In **SQL Server Installation Center**, in **Installation**, click **Install SQL Server Management Tools**.
17. In the browser window, click to download the latest version of SSMS.
18. Run the installer and click **Install**.
19. When the installer has finished, click **Restart** if needed or **Close**.

Creating the Northwind sample database for SQL Server

Now we can run a database script to create the Northwind sample database:

1. If you have not previously downloaded or cloned the GitHub repository for this book, then do so now using the following link: `https://github.com/markjprice/cs10dotnet6/`.
2. Copy the script to create the Northwind database for SQL Server from the following path in your local Git repository: `/sql-scripts/Northwind4SQLServer.sql` into the `WorkingWithEFCore` folder.
3. Start **SQL Server Management Studio**.
4. In the **Connect to Server** dialog, for **Server name**, enter `.` (a dot) meaning the local computer name, and then click **Connect**.

If you had to create a named instance, like `cs10dotnet6`, then enter `.\cs10dotnet6`

5. Navigate to **File | Open | File...**.
6. Browse to select the `Northwind4SQLServer.sql` file and then click **Open**.
7. In the toolbar, click **Execute**, and note the **the Command(s) completed successfully** message.
8. In **Object Explorer**, expand the **Northwind** database, and then expand **Tables**.
9. Right-click **Products**, click **Select Top 1000 Rows**, and note the returned results, as shown in *Figure 10.2*:

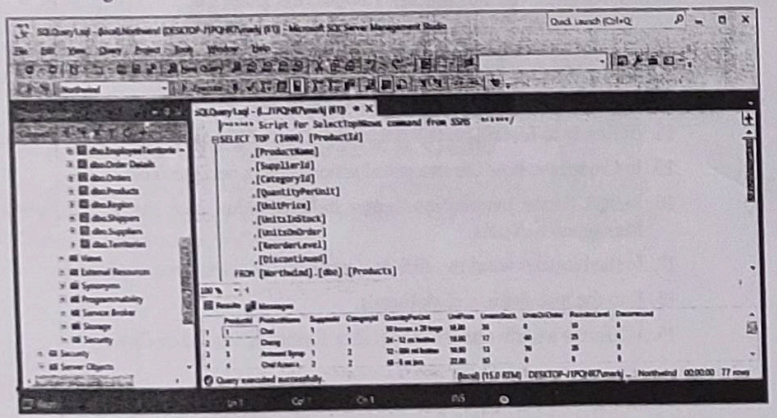

Figure 10.2: The Products table in SQL Server Management Studio

10. In the **Object Explorer** toolbar, click the **Disconnect** button.
11. Exit SQL Server Management Studio.

Managing the Northwind sample database with Server Explorer

We did not have to use SQL Server Management Studio to execute the database script. We can also use tools in Visual Studio including the **SQL Server Object Explorer** and **Server Explorer**:

1. In Visual Studio, choose **View | Server Explorer**.
2. In the **Server Explorer** window, right-click **Data Connections** and choose **Add Connection...**.
3. If you see the **Choose Data Source** dialog, as shown in *Figure 10.3*, select **Microsoft SQL Server** and then click **Continue**:

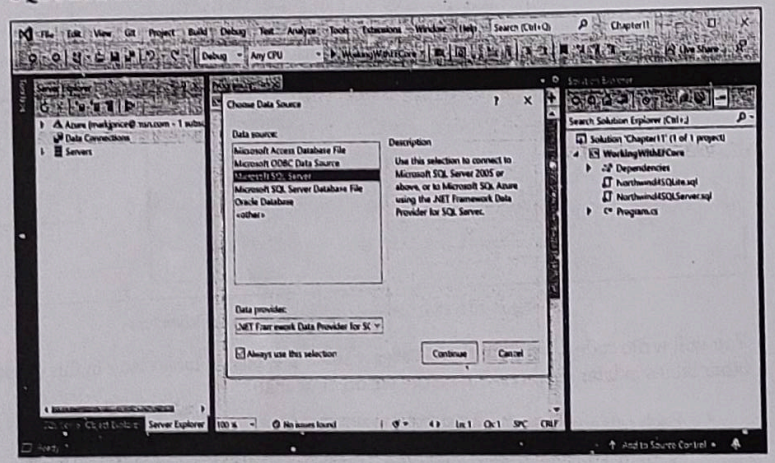

Figure 10.3: Choosing SQL Server as the data source

4. In the **Add Connection** dialog, enter the server name as `.`, enter the database name as `Northwind`, and then click **OK**.
5. In **Server Explorer**, expand the data connection and its tables. You should see 13 tables, including the **Categories** and **Products** tables.
6. Right-click the **Products** table, choose **Show Table Data**, and note the 77 rows of products are returned.
7. To see the details of the **Products** table columns and types, right-click **Products** and choose **Open Table Definition**, or double-click the table in **Server Explorer**.

Using SQLite

SQLite is a small, cross-platform, self-contained RDBMS that is available in the public domain. It's the most common RDBMS for mobile platforms such as iOS (iPhone and iPad) and Android. Even if you use Windows and set up SQL Server in the previous section, you might want to set up SQLite too. The code that we write will work with both and it can be interesting to see the subtle differences.

Setting up SQLite for macOS

SQLite is included in macOS in the `/usr/bin/` directory as a command-line application named `sqlite3`.

Setting up SQLite for Windows

On Windows, we need to add the folder for SQLite to the system path so it will be found when we enter commands at a command prompt or terminal:

1. Start your favorite browser and navigate to the following link: `https://www.sqlite.org/download.html`.
2. Scroll down the page to the **Precompiled Binaries for Windows** section.
3. Click **sqlite-tools-win32-x86-3360000.zip**. Note the file might have a higher version number after this book is published.
4. Extract the ZIP file into a folder named `C:\Sqlite\`.
5. Navigate to **Windows Settings**.
6. Search for `environment` and choose **Edit the system environment variables**. On non-English versions of Windows, please search for the equivalent word in your local language to find the setting.
7. Click the **Environment Variables** button.
8. In **System variables**, select **Path** in the list, and then click **Edit...**.
9. Click **New**, enter `C:\Sqlite`, and press *Enter*.
10. Click **OK**.
11. Click **OK**.
12. Click **OK**.
13. Close **Windows Settings**.

Setting up SQLite for other OSes

SQLite can be downloaded and installed for other OSes from the following link: `https://www.sqlite.org/download.html`.

Creating the Northwind sample database for SQLite

Now we can create the Northwind sample database for SQLite using an SQL script:

1. If you have not previously cloned the GitHub repository for this book, then do so now using the following link: `https://github.com/markjprice/cs10dotnet6/`.
2. Copy the script to create the Northwind database for SQLite from the following path in your local Git repository: `/sql-scripts/Northwind4SQLite.sql` into the `WorkingWithEFCore` folder.
3. Start a command line in the `WorkingWithEFCore` folder:
    1. On Windows, start **File Explorer**, right-click the `WorkingWithEFCore` folder, and select **New Command Prompt at Folder** or **Open in Windows Terminal**.
    2. On macOS, start **Finder**, right-click the `WorkingWithEFCore` folder, and select **New Terminal at Folder**.
4. Enter the command to execute the SQL script using SQLite and create the `Northwind.db` database, as shown in the following command:

```
sqlite3 Northwind.db -init Northwind4SQLite.sql.
```

5. Be patient because this command might take a while to create the database structure. Eventually, you will see the SQLite command prompt, as shown in the following output:

```
-- Loading resources from Northwind4SQLite.sql
SQLite version 3.36.0 2021-08-24 15:20:15
Enter ".help" for usage hints.
sqlite>
```

6. Press *Ctrl + C* on Windows or *Ctrl + D* on macOS to exit SQLite command mode.
7. Leave your terminal or command prompt window open because you will use it again soon.

Managing the Northwind sample database with SQLiteStudio

You can use a cross-platform graphical database manager named **SQLiteStudio** to easily manage SQLite databases:

1. Navigate to the following link, `https://sqlitestudio.pl`, and download and extract the application to your preferred location.
2. Start **SQLiteStudio**.
3. On the **Database** menu, choose **Add a database**.

4. In the **Database** dialog, in the **File** section, click on the yellow folder button to browse for an existing database file on the local computer, select the `Northwind.db` file in the `WorkingWithEFCore` folder, and then click **OK**.
5. Right-click on the **Northwind** database and choose **Connect to the database**. You will see the 10 tables that were created by the script. (The script for SQLite is simpler than the one for SQL Server; it does not create as many tables or other database objects.)
6. Right-click on the **Products** table and choose **Edit the table**.
7. In the table editor window, note the structure of the `Products` table, including column names, data types, keys, and constraints, as shown in *Figure 10.4*:

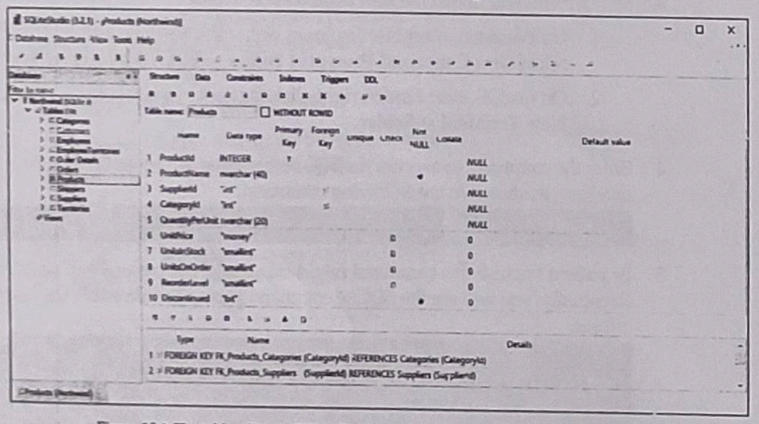

Figure 10.4: The table editor in SQLiteStudio showing the structure of the Products table

8. In the table editor window, click the **Data** tab, and you will see 77 products, as shown in *Figure 10.5*:

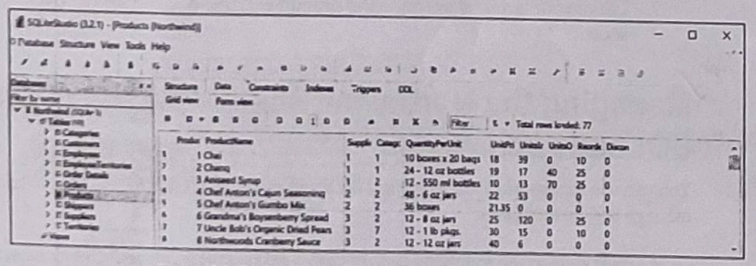

Figure 10.5: The Data tab showing the rows in the Products table

9. In the **Database** window, right-click **Northwind** and select **Disconnect from the database**.
10. Exit SQLiteStudio.

Setting up EF Core

Before we dive into the practicalities of managing data using EF Core, let's briefly talk about choosing between EF Core data providers.

Choosing an EF Core database provider

To manage data in a specific database, we need classes that know how to efficiently talk to that database.

EF Core database providers are sets of classes that are optimized for a specific data store. There is even a provider for storing the data in the memory of the current process, which can be useful for high-performance unit testing since it avoids hitting an external system.

They are distributed as NuGet packages, as shown in the following table:

To manage this data store	Install this NuGet package
Microsoft SQL Server 2012 or later	`Microsoft.EntityFrameworkCore.SqlServer`
SQLite 3.7 or later	`Microsoft.EntityFrameworkCore.SQLite`
MySQL	`MySQL.Data.EntityFrameworkCore`
In-memory	`Microsoft.EntityFrameworkCore.InMemory`
Azure Cosmos DB SQL API	`Microsoft.EntityFrameworkCore.Cosmos`
Oracle DB 11.2	`Oracle.EntityFrameworkCore`

You can install as many EF Core database providers in the same project as you need. Each package includes the shared types as well as provider-specific types.

Connecting to a database

To connect to an SQLite database, we just need to know the database filename, set using the parameter `Filename`.

To connect to an SQL Server database, we need to know multiple pieces of information, as shown in the following list:

- The name of the server (and the instance if it has one).
- The name of the database.
- Security information, such as username and password, or if we should pass the currently logged-on user's credentials automatically.

We specify this information in a **connection string**.

For backward compatibility, there are multiple possible keywords we can use in an SQL Server connection string for the various parameters, as shown in the following list:

- `Data Source` or `server` or `addr`: These keywords are the name of the server (and an optional instance). You can use a dot `.` to mean the local server.
- `Initial Catalog` or `database`: These keywords are the name of the database.
- `Integrated Security` or `trusted_connection`: These keywords are set to `true` or `SSPI` to pass the thread's current user credentials.
- `MultipleActiveResultSets`: This keyword is set to `true` to enable a single connection to be used to work with multiple tables simultaneously to improve efficiency. It is used for lazy loading rows from related tables.

As described in the list above, when you write code to connect to an SQL Server database, you need to know its server name. The server name depends on the edition and version of SQL Server that you will connect to, as shown in the following table:

SQL Server edition	Server name \ Instance name
LocalDB 2012	`(localdb)\v11.0`
LocalDB 2016 or later	`(localdb)\mssqllocaldb`
Express	`.\sqlexpress`
Full/Developer (default instance)	`.`
Full/Developer (named instance)	`.\cs10dotnet6`

Good Practice: Use a dot `.` as shorthand for the local computer name. Remember that server names for SQL Server are made of two parts: the name of the computer and the name of an SQL Server instance. You provide instance names during custom installation.

Defining the Northwind database context class

The `Northwind` class will be used to represent the database. To use EF Core, the class must inherit from `DbContext`. This class understands how to communicate with databases and dynamically generate SQL statements to query and manipulate data.

Your `DbContext`-derived class should have an overridden method named `OnConfiguring`, which will set the database connection string.

To make it easy for you to try SQLite and SQL Server, we will create a project that supports both, with a `string` field to control which is used at runtime:

1. In the `WorkingWithEFCore` project, add package references to the EF Core data provider for both SQL Server and SQLite, as shown in the following markup:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Sqlite"
    Version="6.0.0" />
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="6.0.0" />
</ItemGroup>
```


2. Build the project to restore packages.
3. Add a class file named `ProjectConstants.cs`.
4. In `ProjectConstants.cs`, define a class with a public string constant to store the database provider name that you want to use, as shown in the following code:

```
namespace Packt.Shared;

public class ProjectConstants
{
  public const string DatabaseProvider = "SQLite"; // or "SQLServer"
}
```

5. In `Program.cs`, import the `Packt.Shared` namespace and output the database provider, as shown in the following code:

```
WriteLine($"Using {ProjectConstants.DatabaseProvider} database
provider.");
```

6. Add a class file named `Northwind.cs`.
7. In `Northwind.cs`, define a class named `Northwind`, import the main namespace for EF Core, make the class inherit from `DbContext`, and in an `OnConfiguring` method, check the `provider` field to either use SQLite or SQL Server, as shown in the following code:

```
using Microsoft.EntityFrameworkCore; // DbContext, DbContextOptionsBuilder

using static System.Console;

namespace Packt.Shared;

// this manages the connection to the database
public class Northwind : DbContext
{
  protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
  {
    if (ProjectConstants.DatabaseProvider == "SQLite")
    {
      string path = Path.Combine(
        Environment.CurrentDirectory, "Northwind.db");

      WriteLine($"Using {path} database file.");
```

```
        optionsBuilder.UseSqlite($"Filename={path}");
      }
      else
      {
        string connection = "Data Source=.;" +
          "Initial Catalog=Northwind;" +
          "Integrated Security=true;" +
          "MultipleActiveResultSets=true;";

        optionsBuilder.UseSqlServer(connection);
      }
    }
  }
```

If you are using Visual Studio for Windows, then the compiled application executes in the `WorkingWithEFCore\bin\Debug\net6.0` folder so it will not find the database file.

8. In **Solution Explorer**, right-click the `Northwind.db` file and select **Properties**.
9. In **Properties**, set **Copy to Output Directory** to **Copy always**.
10. Open `WorkingWithEFCore.csproj` and note the new elements, as shown in the following markup:

```
<ItemGroup>
  <None Update="Northwind.db">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

If you are using Visual Studio Code, then the compiled application executes in the `WorkingWithEFCore` folder so it will find the database file without it being copied.

11. Run the console application and note the output showing which database provider you chose to use.

Defining EF Core models

EF Core uses a combination of conventions, annotation attributes, and Fluent API statements to build an entity model at runtime so that any actions performed on the classes can later be automatically translated into actions performed on the actual database. An entity class represents the structure of a table and an instance of the class represents a row in that table.

First, we will review the three ways to define a model, with code examples, and then we will create some classes that implement those techniques.

Using EF Core conventions to define the model

The code we will write will use the following conventions:

- The name of a table is assumed to match the name of a `DbSet<T>` property in the `DbContext` class, for example, `Products`.
- The names of the columns are assumed to match the names of properties in the entity model class, for example, `ProductId`.
- The `string` .NET type is assumed to be a `nvarchar` type in the database.
- The `int` .NET type is assumed to be an `int` type in the database.
- The primary key is assumed to be a property that is named `Id` or `ID`, or when the entity model class is named `Product`, then the property can be named `ProductId` or `ProductID`. If this property is an integer type or the `Guid` type, then it is also assumed to be an `IDENTITY` column (a column type that automatically assigns a value when inserting).

Good Practice: There are many other conventions that you should know, and you can even define your own, but that is beyond the scope of this book. You can read about them at the following link: `https://docs.microsoft.com/en-us/ef/core/modeling/`

Using EF Core annotation attributes to define the model

Conventions often aren't enough to completely map the classes to the database objects. A simple way of adding more smarts to your model is to apply annotation attributes.

Some common attributes are shown in the following table:

Attribute	Description
`[Required]`	Ensures the value is not `null`.
`[StringLength(50)]`	Ensures the value is up to 50 characters in length.
`[RegularExpression(expression)]`	Ensures the value matches the specified regular expression.
`[Column(TypeName = "money", Name = "UnitPrice")]`	Specifies the column type and column name used in the table.

For example, in the database, the maximum length of a product name is 40, and the value cannot be null, as shown highlighted in the following **Data Definition Language** (DDL) code that defines how to create a table named `Products` with its columns, data types, keys, and other constraints:

```
CREATE TABLE Products (
    ProductId       INTEGER         PRIMARY KEY,
    ProductName     NVARCHAR (40) NOT NULL,
    SupplierId      "INT",
```

```
    CategoryId      "INT",
    QuantityPerUnit NVARCHAR (20),
    UnitPrice       "MONEY"       CONSTRAINT DF_Products_UnitPrice DEFAULT (0),
    UnitsInStock    "SMALLINT"    CONSTRAINT DF_Products_UnitsInStock DEFAULT (0),
    UnitsOnOrder    "SMALLINT"    CONSTRAINT DF_Products_UnitsOnOrder DEFAULT (0),
    ReorderLevel    "SMALLINT"    CONSTRAINT DF_Products_ReorderLevel DEFAULT (0),
    Discontinued    "BIT"         NOT NULL
                                  CONSTRAINT DF_Products_Discontinued DEFAULT (0),
    CONSTRAINT FK_Products_Categories FOREIGN KEY (
        CategoryId
    )
    REFERENCES Categories (CategoryId),
    CONSTRAINT FK_Products_Suppliers FOREIGN KEY (
        SupplierId
    )
    REFERENCES Suppliers (SupplierId),
    CONSTRAINT CK_Products_UnitPrice CHECK (UnitPrice >= 0),
    CONSTRAINT CK_ReorderLevel CHECK (ReorderLevel >= 0),
    CONSTRAINT CK_UnitsInStock CHECK (UnitsInStock >= 0),
    CONSTRAINT CK_UnitsOnOrder CHECK (UnitsOnOrder >= 0)
);
```

In a Product class, we could apply attributes to specify this, as shown in the following code:

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

When there isn't an obvious map between .NET types and database types, an attribute can be used.

For example, in the database, the column type of `UnitPrice` for the `Products` table is `money`. .NET does not have a `money` type, so it should use `decimal` instead, as shown in the following code:

```
[Column(TypeName = "money")]
public decimal? UnitPrice { get; set; }
```

Another example is for the `Categories` table, as shown in the following DDL code:

```
CREATE TABLE Categories (
    CategoryId   INTEGER       PRIMARY KEY,
    CategoryName NVARCHAR (15) NOT NULL,
    Description  "NTEXT",
    Picture      "IMAGE"
);
```


The `Description` column can be longer than the maximum 8,000 characters that can be stored in a `nvarchar` variable, so it needs to map to `ntext` instead, as shown in the following code:

```
[Column(TypeName = "ntext")]
public string Description { get; set; }
```

Using the EF Core Fluent API to define the model

The last way that the model can be defined is by using the Fluent API. This API can be used instead of attributes, as well as being used in addition to them. For example, to define the `ProductName` property, instead of decorating the property with two attributes, an equivalent Fluent API statement could be written in the `OnModelCreating` method of the database context class, as shown in the following code:

```
modelBuilder.Entity<Product>()
  .Property(product => product.ProductName)
  .IsRequired()
  .HasMaxLength(40);
```

This keeps the entity model class simpler.

Understanding data seeding with the Fluent API

Another benefit of the Fluent API is to provide initial data to populate a database. EF Core automatically works out what insert, update, or delete operations must be executed.

For example, if we wanted to make sure that a new database has at least one row in the `Product` table, then we would call the `HasData` method, as shown in the following code:

```
modelBuilder.Entity<Product>()
  .HasData(new Product
  {
    ProductId = 1,
    ProductName = "Chai",
    UnitPrice = 8.99M
  });
```

Our model will map to an existing database that is already populated with data so we will not need to use this technique in our code.

Building an EF Core model for the Northwind tables

Now that you've learned about ways to define an EF Core model, let's build a model to represent two tables in the `Northwind` database.

The two entity classes will refer to each other, so to avoid compiler errors, we will create the classes without any members first:

1. In the WorkingWithEFCore project, add two class files named Category.cs and Product.cs.
2. In Category.cs, define a class named Category, as shown in the following code:

```
namespace Packt.Shared;

public class Category
{
}
```

3. In Product.cs, define a class named Product, as shown in the following code:

```
namespace Packt.Shared;

public class Product
{
}
```

Defining the Category and Product entity classes

The Category class, also known as an entity model, will be used to represent a row in the Categories table. This table has four columns, as shown in the following DDL:

```
CREATE TABLE Categories (
    CategoryId   INTEGER       PRIMARY KEY,
    CategoryName NVARCHAR (15) NOT NULL,
    Description  "NTEXT",
    Picture      "IMAGE"
);
```

We will use conventions to define:

- Three of the four properties (we will not map the Picture column).
- The primary key.
- The one-to-many relationship to the Products table.

To map the Description column to the correct database type, we will need to decorate the string property with the Column attribute.

Later in this chapter, we will use the Fluent API to define that CategoryName cannot be null and is limited to a maximum of 15 characters.

Let's go:

1. Modify the Category entity model class, as shown in the following code:

```
using System.ComponentModel.DataAnnotations.Schema; // [Column]

namespace Packt.Shared;

public class Category
{
  // these properties map to columns in the database
  public int CategoryId { get; set; }
  public string? CategoryName { get; set; }

  [Column(TypeName = "ntext")]
  public string? Description { get; set; }

  // defines a navigation property for related rows
  public virtual ICollection<Product> Products { get; set; }

  public Category()
  {
    // to enable developers to add products to a Category we must
    // initialize the navigation property to an empty collection
    Products = new HashSet<Product>();
  }
}
```

The Product class will be used to represent a row in the Products table, which has ten columns.

You do not need to include all columns from a table as properties of a class. We will only map six properties: ProductId, ProductName, UnitPrice, UnitsInStock, Discontinued, and CategoryId.

Columns that are not mapped to properties cannot be read or set using the class instances. If you use the class to create a new object, then the new row in the table will have NULL or some other default value for the unmapped column values in that row. You must make sure that those missing columns are optional or have default values set by the database or an exception will be thrown at runtime. In this scenario, the rows already have data values and I have decided that I do not need to read those values in this application.

We can rename a column by defining a property with a different name, like Cost, and then decorating the property with the [Column] attribute and specifying its column name, like UnitPrice.

The final property, CategoryId, is associated with a Category property that will be used to map each product to its parent category.

2. Modify the Product class, as shown in the following code:

```
using System.ComponentModel.DataAnnotations; // [Required], [StringLength]
using System.ComponentModel.DataAnnotations.Schema; // [Column]

namespace Packt.Shared;
```

```
public class Product
{
  public int ProductId { get; set; } // primary key

  [Required]
  [StringLength(40)]
  public string ProductName { get; set; } = null!;

  [Column("UnitPrice", TypeName = "money")]
  public decimal? Cost { get; set; } // property name != column name

  [Column("UnitsInStock")]
  public short? Stock { get; set; }

  public bool Discontinued { get; set; }

  // these two define the foreign key relationship
  // to the Categories table
  public int CategoryId { get; set; }
  public virtual Category Category { get; set; } = null!;
}
```

The two properties that relate the two entities, `Category.Products` and `Product.Category`, are both marked as `virtual`. This allows EF Core to inherit and override the properties to provide extra features, such as lazy loading.

Adding tables to the Northwind database context class

Inside your `DbContext`-derived class, you must define at least one property of the `DbSet<T>` type. These properties represent the tables. To tell EF Core what columns each table has, the `DbSet<T>` properties use generics to specify a class that represents a row in the table. That entity model class has properties that represent its columns.

The `DbContext`-derived class can optionally have an overridden method named `OnModelCreating`. This is where you can write Fluent API statements as an alternative to decorating your entity classes with attributes.

Let's write some code:

1. Modify the `Northwind` class to add statements to define two properties for the two tables and an `OnModelCreating` method, as shown highlighted in the following code:

```
public class Northwind : DbContext
{
  // these properties map to tables in the database
  public DbSet<Category>? Categories { get; set; }
  public DbSet<Product>? Products { get; set; }

  protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
  {
    ...
  }

  protected override void OnModelCreating(
    ModelBuilder modelBuilder)
  {
    // example of using Fluent API instead of attributes
    // to limit the length of a category name to 15
    modelBuilder.Entity<Category>()
      .Property(category => category.CategoryName)
      .IsRequired() // NOT NULL
      .HasMaxLength(15);

    if (ProjectConstants.DatabaseProvider == "SQLite")
    {
      // added to "fix" the lack of decimal support in SQLite
      modelBuilder.Entity<Product>()
        .Property(product => product.Cost)
        .HasConversion<double>();
    }
  }
}
```

In EF Core 3.0 and later, the `decimal` type is not supported by the SQLite database provider for sorting and other operations. We can fix this by telling the model that `decimal` values can be converted to `double` values when using the SQLite database provider. This does not actually perform any conversion at runtime.

Now that you have seen some examples of defining an entity model manually, let's see a tool that can do some of the work for you.

Setting up the dotnet-ef tool

.NET has a command-line tool named `dotnet`. It can be extended with capabilities useful for working with EF Core. It can perform design-time tasks like creating and applying migrations from an older model to a newer model and generating code for a model from an existing database.

The `dotnet ef` command-line tool is not automatically installed. You have to install this package as either a **global** or **local tool**. If you have already installed an older version of the tool, then you should uninstall any existing version:

1. At a command prompt or terminal, check if you have already installed `dotnet-ef` as a global tool, as shown in the following command:

```
dotnet tool list --global
```

2. Check in the list if an older version of the tool has been installed, like the one for .NET Core 3.1, as shown in the following output:

```
Package Id      Version      Commands
-------------------------------------
dotnet-ef       3.1.0        dotnet-ef
```

3. If an old version is already installed, then uninstall the tool, as shown in the following command:

```
dotnet tool uninstall --global dotnet-ef
```

4. Install the latest version, as shown in the following command:

```
dotnet tool install --global dotnet-ef --version 6.0.0
```

5. If necessary, follow any OS-specific instructions to add the `dotnet tools` directory to your PATH environment variable as described in the output of installing the `dotnet-ef` tool.

Scaffolding models using an existing database

Scaffolding is the process of using a tool to create classes that represent the model of an existing database using reverse engineering. A good scaffolding tool allows you to extend the automatically generated classes and then regenerate those classes without losing your extended classes.

If you know that you will never regenerate the classes using the tool, then feel free to change the code for the automatically generated classes as much as you want. The code generated by the tool is just the best approximation.

Good Practice: Do not be afraid to overrule a tool when you know better.

Let's see if the tool generates the same model as we did manually:

1. Add the `Microsoft.EntityFrameworkCore.Design` package to the `WorkingWithEFCore` project.

2. At a command prompt or terminal in the `WorkingWithEFCore` folder, generate a model for the `Categories` and `Products` tables in a new folder named `AutoGenModels`, as shown in the following command:

```
dotnet ef dbcontext scaffold "Filename=Northwind.db" Microsoft.
EntityFrameworkCore.Sqlite --table Categories --table Products --output-
dir AutoGenModels --namespace WorkingWithEFCore.AutoGen --data-annotations
--context Northwind
```

Note the following:

- The command action: `dbcontext scaffold`
- The connection string: `"Filename=Northwind.db"`
- The database provider: `Microsoft.EntityFrameworkCore.Sqlite`
- The tables to generate models for: `--table Categories --table Products`
- The output folder: `--output-dir AutoGenModels`
- The namespace: `--namespace WorkingWithEFCore.AutoGen`
- To use data annotations as well as the Fluent API: `--data-annotations`
- To rename the context from [database_name]Context: `--context Northwind`

For SQL Server, change the database provider and connection string, as shown in the following command:

```
dotnet ef dbcontext scaffold "Data Source=.;Initial
Catalog=Northwind;Integrated Security=true;" Microsoft.
EntityFrameworkCore.SqlServer --table Categories
--table Products --output-dir AutoGenModels --namespace
WorkingWithEFCore.AutoGen --data-annotations --context
Northwind
```

3. Note the build messages and warnings, as shown in the following output:

```
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string,
you should move it out of source code. You can avoid scaffolding the
connection string by using the Name= syntax to read it from configuration
- see https://go.microsoft.com/fwlink/?linkid=2131148. For more
guidance on storing connection strings, see http://go.microsoft.com/
fwlink/?LinkId=723263.
Skipping foreign key with identity '0' on table 'Products' since principal
table 'Suppliers' was not found in the model. This usually happens when
the principal table was not included in the selection set.
```

4. Open the `AutoGenModels` folder and note the three class files that were automatically generated: `Category.cs`, `Northwind.cs`, and `Product.cs`.
5. Open `Category.cs` and note the differences compared to the one you created manually, as shown in the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

namespace WorkingWithEFCore.AutoGen
{
  [Index(nameof(CategoryName), Name = "CategoryName")]
  public partial class Category
  {
    public Category()
    {
      Products = new HashSet<Product>();
    }

    [Key]
    public long CategoryId { get; set; }

    [Required]
    [Column(TypeName = "nvarchar (15)")] // SQLite
    [StringLength(15)] // SQL Server
    public string CategoryName { get; set; }

    [Column(TypeName = "ntext")]
    public string? Description { get; set; }

    [Column(TypeName = "image")]
    public byte[]? Picture { get; set; }

    [InverseProperty(nameof(Product.Category))]
    public virtual ICollection<Product> Products { get; set; }
  }
}
```

Note the following:

- It decorates the entity class with the `[Index]` attribute that was introduced in EF Core 5.0. This indicates properties that should have an index. In earlier versions, only the Fluent API was supported for defining indexes. Since we are working with an existing database, this is not needed. But if we want to recreate a new empty database from our code then this information will be needed.
- The table name in the database is `Categories` but the `dotnet-ef` tool uses the **Humanizer** third-party library to automatically singularize the class name to `Category`, which is a more natural name when creating a single entity.

- The entity class is declared using the `partial` keyword so that you can create a matching `partial` class for adding additional code. This allows you to rerun the tool and regenerate the entity class without losing that extra code.
- The `CategoryId` property is decorated with the `[Key]` attribute to indicate that it is the primary key for this entity. The data type for this property is `int` for SQL Server and `long` for SQLite.
- The `Products` property uses the `[InverseProperty]` attribute to define the foreign key relationship to the `Category` property on the `Product` entity class.

6. Open `Product.cs` and note the differences compared to the one you created manually.
7. Open `Northwind.cs` and note the differences compared to the one you created manually, as shown in the following edited-for-space code:

```
using Microsoft.EntityFrameworkCore;

namespace WorkingWithEFCore.AutoGen
{
  public partial class Northwind : DbContext
  {
    public Northwind()
    {
    }

    public Northwind(DbContextOptions<Northwind> options)
      : base(options)
    {
    }

    public virtual DbSet<Category> Categories { get; set; } = null!;
    public virtual DbSet<Product> Products { get; set; } = null!;

    protected override void OnConfiguring(
      DbContextOptionsBuilder optionsBuilder)
    {
      if (!optionsBuilder.IsConfigured)
      {
#warning To protect potentially sensitive information in your connection
string, you should move it out of source code. You can avoid scaffolding
the connection string by using the Name= syntax to read it from
configuration - see https://go.microsoft.com/fwlink/?linkid=2131148. For
more guidance on storing connection strings, see http://go.microsoft.com/
fwlink/?LinkId=723263.
        optionsBuilder.UseSqlite("Filename=Northwind.db");
      }
    }
```

```
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.Entity<Category>(entity =>
      {
        ...
      });

      modelBuilder.Entity<Product>(entity =>
      {
        ...
      });

      OnModelCreatingPartial(modelBuilder);
    }

    partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
  }
}
```

Note the following:

- The `Northwind` data context class is `partial` to allow you to extend it and regenerate it in the future.
- It has two constructors: a default parameter-less one and one that allows options to be passed in. This is useful in apps where you want to specify the connection string at runtime.
- The two `DbSet<T>` properties that represent the `Categories` and `Products` tables are set to the `null`-forgiving value to prevent static compiler analysis warnings at compile time. It has no effect at runtime.
- In the `OnConfiguring` method, if options have not been specified in the constructor, then it defaults to using a connection string that looks for the database file in the current folder. It has a compiler warning to remind you that you should not hardcode security information in this connection string.
- In the `OnModelCreating` method, the Fluent API is used to configure the two entity classes, and then a partial method named `OnModelCreatingPartial` is invoked. This allows you to implement that partial method in your own partial `Northwind` class to add your own Fluent API configuration that will not be lost if you regenerate the model classes.

8. Close the automatically generated class files.

Configuring preconvention models

Along with support for the `DateOnly` and `TimeOnly` types for use with the SQLite database provider, one of the new features introduced with EF Core 6 is configuring preconvention models.

As models become more complex, relying on conventions to discover entity types and their properties and successfully map them to tables and columns becomes harder. It would be useful if you could configure the conventions themselves before they are used to analyze and build a model.

For example, you might want to define a convention to say that all `string` properties should have a maximum length of 50 characters as a default, or any property types that implement a custom interface should not be mapped, as shown in the following code:

```
protected override void ConfigureConventions(
  ModelConfigurationBuilder configurationBuilder)
{
  configurationBuilder.Properties<string>().HaveMaxLength(50);
  configurationBuilder.IgnoreAny<IDoNotMap>();
}
```

In the rest of this chapter, we will use the classes that you manually created.

Querying EF Core models

Now that we have a model that maps to the Northwind database and two of its tables, we can write some simple LINQ queries to fetch data. You will learn much more about writing LINQ queries in *Chapter 11, Querying and Manipulating Data Using LINQ*.

For now, just write the code and view the results:

1. At the top of `Program.cs`, import the main EF Core namespace to enable the use of the `Include` extension method to prefetch from a related table:

```
using Microsoft.EntityFrameworkCore; // Include extension method
```

2. At the bottom of `Program.cs`, define a `QueryingCategories` method, and add statements to do these tasks, as shown in the following code:
   - Create an instance of the `Northwind` class that will manage the database. Database context instances are designed for short lifetimes in a unit of work. They should be disposed of as soon as possible so we will wrap it in a `using` statement. In *Chapter 14, Building Websites Using ASP.NET Core Razor Pages*, you will learn how to get a database context using dependency injection.
   - Create a query for all categories that include their related products.
   - Enumerate through the categories, outputting the name and number of products for each one:

```
static void QueryingCategories()
{
  using (Northwind db = new())
  {
    WriteLine("Categories and how many products they have:");
```

```
    // a query to get all categories and their related products
    IQueryable<Category>? categories = db.Categories?
      .Include(c => c.Products);

    if (categories is null)
    {
      WriteLine("No categories found.");
      return;
    }

    // execute query and enumerate results
    foreach (Category c in categories)
    {
      WriteLine($"{c.CategoryName} has {c.Products.Count} products.");
    }
  }
}
```

3. At the top of `Program.cs`, after outputting the database provider name, call the `QueryingCategories` method, as shown highlighted in the following code:

```
WriteLine($"Using {ProjectConstants.DatabaseProvider} database provider.");
QueryingCategories();
```

4. Run the code and view the result (if run with Visual Studio 2022 for Windows using the SQLite database provider), as shown in the following output:

```
Using SQLite database provider.
Categories and how many products they have:
Using C:\Code\Chapter10\WorkingWithEFCore\bin\Debug\net6.0\Northwind.db
database file.
Beverages has 12 products.
Condiments has 12 products.
Confections has 13 products.
Dairy Products has 10 products.
Grains/Cereals has 7 products.
Meat/Poultry has 6 products.
Produce has 5 products.
Seafood has 12 products.
```

If you run with Visual Studio Code using the SQLite database provider, then the path will be the `WorkingWithEFCore` folder. If you run using the SQL Server database provider, then there is no database file path output.

Warning! If you see the following exception when using SQLite with Visual Studio 2022, the most likely problem is that the `Northwind.db` file is not being copied to the output directory. Make sure **Copy to Output Directory** is set to **Copy always**:

```
Unhandled exception. Microsoft.Data.Sqlite.SqliteException
(0x80004005): SQLite Error 1: 'no such table: Categories'.
```

Filtering included entities

EF Core 5.0 introduced **filtered includes**, which means you can specify a lambda expression in the `Include` method call to filter which entities are returned in the results:

1. At the bottom of `Program.cs`, define a `FilteredIncludes` method, and add statements to do these tasks, as shown in the following code:
   - Create an instance of the `Northwind` class that will manage the database.
   - Prompt the user to enter a minimum value for units in stock.
   - Create a query for categories that have products with that minimum number of units in stock.
   - Enumerate through the categories and products, outputting the name and units in stock for each one:

```
static void FilteredIncludes()
{
  using (Northwind db = new())
  {
    Write("Enter a minimum for units in stock: ");
    string unitsInStock = ReadLine() ?? "10";
    int stock = int.Parse(unitsInStock);

    IQueryable<Category>? categories = db.Categories?
      .Include(c => c.Products.Where(p => p.Stock >= stock));

    if (categories is null)
    {
      WriteLine("No categories found.");
      return;
    }

    foreach (Category c in categories)
    {
      WriteLine($"{c.CategoryName} has {c.Products.Count} products with a
minimum of {stock} units in stock.");

      foreach(Product p in c.Products)
      {
```

```
            WriteLine($"  {p.ProductName} has {p.Stock} units in stock.");
          }
        }
      }
    }
```

2. In `Program.cs`, comment out the `QueryingCategories` method and invoke the `FilteredIncludes` method, as shown highlighted in the following code:

```
WriteLine($"Using {ProjectConstants.DatabaseProvider} database provider.");
// QueryingCategories();
FilteredIncludes();
```

3. Run the code, enter a minimum for units in stock like `100`, and view the result, as shown in the following output:

```
Enter a minimum for units in stock: 100
Beverages has 2 products with a minimum of 100 units in stock.
  Sasquatch Ale has 111 units in stock.
  Rhönbräu Klosterbier has 125 units in stock.
Condiments has 2 products with a minimum of 100 units in stock.
  Grandma's Boysenberry Spread has 120 units in stock.
  Sirop d'érable has 113 units in stock.
Confections has 0 products with a minimum of 100 units in stock.
Dairy Products has 1 products with a minimum of 100 units in stock.
  Geitost has 112 units in stock.
Grains/Cereals has 1 products with a minimum of 100 units in stock.
  Gustaf's Knäckebröd has 104 units in stock.
Meat/Poultry has 1 products with a minimum of 100 units in stock.
  Pâté chinois has 115 units in stock.
Produce has 0 products with a minimum of 100 units in stock.
Seafood has 3 products with a minimum of 100 units in stock.
  Inlagd Sill has 112 units in stock.
  Boston Crab Meat has 123 units in stock.
  Röd Kaviar has 101 units in stock.
```

Unicode characters in the Windows console

There is a limitation with the console provided by Microsoft on versions of Windows before the Windows 10 Fall Creators Update. By default, the console cannot display Unicode characters, for example, in the name Rhönbräu.

If you have this issue, then you can temporarily change the code page (also known as the character set) in a console to Unicode UTF-8 by entering the following command at the prompt before running the app:

```
chcp 65001
```


Filtering and sorting products

Let's explore a more complex query that will filter and sort data:

1. At the bottom of `Program.cs`, define a `QueryingProducts` method, and add statements to do the following, as shown in the following code:
   - Create an instance of the `Northwind` class that will manage the database.
   - Prompt the user for a price for products. Unlike the previous code example, we will loop until the input is a valid price.
   - Create a query for products that cost more than the price using LINQ.
   - Loop through the results, outputting the Id, name, cost (formatted in US dollars), and the number of units in stock:

```
static void QueryingProducts()
{
  using (Northwind db = new())
  {
    WriteLine("Products that cost more than a price, highest at top.");
    string? input;
    decimal price;

    do
    {
      Write("Enter a product price: ");
      input = ReadLine();
    } while (!decimal.TryParse(input, out price));

    IQueryable<Product>? products = db.Products?
      .Where(product => product.Cost > price)
      .OrderByDescending(product => product.Cost);

    if (products is null)
    {
      WriteLine("No products found.");
      return;
    }

    foreach (Product p in products)
    {
      WriteLine(
        "{0}: {1} costs {2:$#,##0.00} and has {3} in stock.",
        p.ProductId, p.ProductName, p.Cost, p.Stock);
    }
  }
}
```

2. In `Program.cs`, comment out the previous method, and call the `QueryingProducts` method
3. Run the code, enter `50` when prompted to enter a product price, and view the result, as shown in the following output:

```
Products that cost more than a price, highest at top.
Enter a product price: 50
38: Côte de Blaye costs $263.50 and has 17 in stock.
29: Thüringer Rostbratwurst costs $123.79 and has 0 in stock.
9: Mishi Kobe Niku costs $97.00 and has 29 in stock.
20: Sir Rodney's Marmalade costs $81.00 and has 40 in stock.
18: Carnarvon Tigers costs $62.50 and has 42 in stock.
59: Raclette Courdavault costs $55.00 and has 79 in stock.
51: Manjimup Dried Apples costs $53.00 and has 20 in stock.
```

Getting the generated SQL

You might be wondering how well written the SQL statements are that are generated from the C# queries we write. EF Core 5.0 introduced a quick and easy way to see the SQL generated:

1. In the `FilteredIncludes` method, before using the `foreach` statement to enumerate the query, add a statement to output the generated SQL, as shown highlighted in the following code:

```
WriteLine($"ToQueryString: {categories.ToQueryString()}");

foreach (Category c in categories)
```

2. In `Program.cs`, comment out the call to the `QueryingProducts` method and uncomment the call to the `FilteredIncludes` method.
3. Run the code, enter a minimum for units in stock like `99`, and view the result (when run with SQLite), as shown in the following output:

```
Enter a minimum for units in stock: 99
Using SQLite database provider.
ToQueryString: .param set @_stock_0 99

SELECT "c"."CategoryId", "c"."CategoryName", "c"."Description",
"t"."ProductId", "t"."CategoryId", "t"."UnitPrice", "t"."Discontinued",
"t"."ProductName", "t"."UnitsInStock"
FROM "Categories" AS "c"
LEFT JOIN (
    SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
    FROM "Products" AS "p"
    WHERE ("p"."UnitsInStock" >= @_stock_0)
) AS "t" ON "c"."CategoryId" = "t"."CategoryId"
ORDER BY "c"."CategoryId", "t"."ProductId"
Beverages has 2 products with a minimum of 99 units in stock.
  Sasquatch Ale has 111 units in stock.
  Rhönbräu Klosterbier has 125 units in stock.
...
```


Note the SQL parameter named `@_stock_0` has been set to a minimum stock value of `99`.

For SQL Server, the SQL generated is slightly different, for example, it uses square brackets instead of double-quotes around object names, as shown in the following output:

```
Enter a minimum for units in stock: 99
Using SqlServer database provider.
ToQueryString: DECLARE @__stock_0 smallint = CAST(99 AS smallint);

SELECT [c].[CategoryId], [c].[CategoryName], [c].[Description], [t].[ProductId],
[t].[CategoryId], [t].[UnitPrice], [t].[Discontinued], [t].[ProductName], [t].
[UnitsInStock]
FROM [Categories] AS [c]
LEFT JOIN (
    SELECT [p].[ProductId], [p].[CategoryId], [p].[UnitPrice], [p].
[Discontinued], [p].[ProductName], [p].[UnitsInStock]
    FROM [Products] AS [p]
    WHERE [p].[UnitsInStock] >= @__stock_0
) AS [t] ON [c].[CategoryId] = [t].[CategoryId]
ORDER BY [c].[CategoryId], [t].[ProductId]
```

Logging EF Core using a custom logging provider

To monitor the interaction between EF Core and the database, we can enable logging. This requires the following two tasks:

- The registering of a **logging provider**.
- The implementation of a **logger**.

Let's see an example of this in action:

1. Add a file to your project named `ConsoleLogger.cs`.
2. Modify the file to define two classes, one to implement `ILoggerProvider` and one to implement `ILogger`, as shown in the following code, and note the following:
    - `ConsoleLoggerProvider` returns an instance of `ConsoleLogger`. It does not need any unmanaged resources, so the `Dispose` method does not do anything, but it must exist.
    - `ConsoleLogger` is disabled for log levels `None`, `Trace`, and `Information`. It is enabled for all other log levels.

- ConsoleLogger implements its Log method by writing to Console:

```
using Microsoft.Extensions.Logging; // ILoggerProvider, ILogger, LogLevel

using static System.Console;

namespace Packt.Shared;

public class ConsoleLoggerProvider : ILoggerProvider
{
  public ILogger CreateLogger(string categoryName)
  {
    // we could have different logger implementations for
    // different categoryName values but we only have one
    return new ConsoleLogger();
  }

  // if your logger uses unmanaged resources,
  // then you can release them here
  public void Dispose() { }
}

public class ConsoleLogger : ILogger
{
  // if your logger uses unmanaged resources, you can
  // return the class that implements IDisposable here
  public IDisposable BeginScope<TState>(TState state)
  {
    return null;
  }

  public bool IsEnabled(LogLevel logLevel)
  {
    // to avoid overLogging, you can filter on the Log Level
    switch(logLevel)
    {
      case LogLevel.Trace:
      case LogLevel.Information:
      case LogLevel.None:
        return false;
      case LogLevel.Debug:
      case LogLevel.Warning:
      case LogLevel.Error:
      case LogLevel.Critical:
      default:
        return true;
    };
  }

  public void Log<TState>(LogLevel logLevel,
    EventId eventId, TState state, Exception? exception,
    Func<TState, Exception, string> formatter)
  {
    // Log the level and event identifier
    Write($"Level: {logLevel}, Event Id: {eventId.Id}");

    // only output the state or exception if it exists
    if (state != null)
    {
      Write($", State: {state}");
    }

    if (exception != null)
    {
      Write($", Exception: {exception.Message}");
    }
    WriteLine();
  }
}
```


3. At the top of Program.cs, add statements to import the namespaces needed for logging, as shown in the following code:

```
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
```

4. We already used the ToQueryString method to get the SQL for FilteredIncludes so we do not need to add logging to that method. To both the QueryingCategories and QueryingProducts methods, add statements immediately inside the using block for the Northwind database context to get the logging factory and register your custom console logger, as shown highlighted in the following code:

```
using (Northwind db = new())
{
  ILoggerFactory loggerFactory = db.GetService<ILoggerFactory>();
  loggerFactory.AddProvider(new ConsoleLoggerProvider());
```

5. At the top of Program.cs, comment out the call to the FilteredIncludes method and uncomment the call to the QueryingProducts method.
6. Run the code and view the logs, which are partially shown in the following output:

```
...
Level: Debug, Event Id: 20000, State: Opening connection to database
'main' on server '/Users/markjprice/Code/Chapter10/WorkingWithEFCore/
Northwind.db'.
Level: Debug, Event Id: 20001, State: Opened connection to database 'main'
on server '/Users/markjprice/Code/Chapter10/WorkingWithEFCore/Northwind.
db'.
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[@__
price_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE "p"."UnitPrice" > @__price_0
ORDER BY "product"."UnitPrice" DESC
...
```

Your logs might vary from those shown above based on your chosen database provider and code editor, and future improvements to EF Core. For now, note that different events like opening a connection or executing a command have different event ids.

Filtering logs by provider-specific values

The event id values and what they mean will be specific to the .NET data provider. If we want to know how the LINQ query has been translated into SQL statements and is executing, then the event Id to output has an `Id` value of `20100`:

1. Modify the `Log` method in `ConsoleLogger` to only output events with an `Id` of `20100`, as highlighted in the following code:

```
public void Log<TState>(LogLevel logLevel, EventId eventId,
  TState state, Exception? exception,
  Func<TState, Exception, string> formatter)
{
  if (eventId.Id == 20100)
  {
    // log the level and event identifier
    Write("Level: {0}, Event Id: {1}, Event: {2}",
      logLevel, eventId.Id, eventId.Name);

    // only output the state or exception if it exists
    if (state != null)
    {
      Write($", State: {state}");
    }
```



```
    if (exception != null)
    {
      Write($", Exception: {exception.Message}");
    }
    WriteLine();
  }
}
```

2. In `Program.cs`, uncomment the `QueryingCategories` method and comment out the other methods so that we can monitor the SQL statements that are generated when joining two tables.
3. Run the code, and note the following SQL statements that were logged, as shown in the following output that has been edited for space:

```
Using SQLServer database provider.
Categories and how many products they have:
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
SELECT [c].[CategoryId], [c].[CategoryName], [c].[Description], [p].
[ProductId], [p].[CategoryId], [p].[UnitPrice], [p].[Discontinued], [p].
[ProductName], [p].[UnitsInStock]
FROM [Categories] AS [c]
LEFT JOIN [Products] AS [p] ON [c].[CategoryId] = [p].[CategoryId]
ORDER BY [c].[CategoryId], [p].[ProductId]
Beverages has 12 products.
Condiments has 12 products.
Confections has 13 products.
Dairy Products has 10 products.
Grains/Cereals has 7 products.
Meat/Poultry has 6 products.
Produce has 5 products.
Seafood has 12 products.
```

Logging with query tags

When logging LINQ queries, it can be tricky to correlate log messages in complex scenarios. EF Core 2.2 introduced the query tags feature to help by allowing you to add SQL comments to the log.

You can annotate a LINQ query using the `TagWith` method, as shown in the following code:

```
IQueryable<Product>? products = db.Products?
  .TagWith("Products filtered by price and sorted.")
  .Where(product => product.Cost > price)
  .OrderByDescending(product => product.Cost);
```

This will add an SQL comment to the log, as shown in the following output:

```
-- Products filtered by price and sorted.
```

Pattern matching with Like

EF Core supports common SQL statements including `Like` for pattern matching:

1. At the bottom of `Program.cs`, add a method named `QueryingWithLike`, as shown in the following code, and note:
   - We have enabled logging.
   - We prompt the user to enter part of a product name and then use the `EF.Functions.Like` method to search anywhere in the `ProductName` property.
   - For each matching product, we output its name, stock, and if it is discontinued:

```
static void QueryingWithLike()
{
  using (Northwind db = new())
  {
    ILoggerFactory loggerFactory = db.GetService<ILoggerFactory>();
    loggerFactory.AddProvider(new ConsoleLoggerProvider());

    Write("Enter part of a product name: ");
    string? input = ReadLine();

    IQueryable<Product>? products = db.Products?
      .Where(p => EF.Functions.Like(p.ProductName, $"%{input}%"));

    if (products is null)
    {
      WriteLine("No products found.");
      return;
    }

    foreach (Product p in products)
    {
      WriteLine("{0} has {1} units in stock. Discontinued? {2}",
        p.ProductName, p.Stock, p.Discontinued);
    }
  }
}
```

2. In `Program.cs`, comment out the existing methods, and call `QueryingWithLike`.

3. Run the code, enter a partial product name such as `che`, and view the result, as shown in the following output:

```
Using SQLServer database provider.
Enter part of a product name: che
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[@__
Format_1='?' (Size = 40)], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock" FROM "Products"
AS "p"
WHERE "p"."ProductName" LIKE @__Format_1
Chef Anton's Cajun Seasoning has 53 units in stock. Discontinued? False
Chef Anton's Gumbo Mix has 0 units in stock. Discontinued? True
Queso Manchego La Pastora has 86 units in stock. Discontinued? False
Gumbär Gummibärchen has 15 units in stock. Discontinued? False
```

EF Core 6.0 introduces another useful function, `EF.Functions.Random`, that maps to a database function returning a pseudo-random number between 0 and 1 exclusive. For example, you could multiply the random number by the count of rows in a table to select one random row from that table.

Defining global filters

Northwind products can be discontinued, so it might be useful to ensure that discontinued products are never returned in results, even if the programmer does not use `Where` to filter them out in their queries:

1. In `Northwind.cs`, modify the `OnModelCreating` method to add a global filter to remove discontinued products, as shown highlighted in the following code:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  ...

  // global filter to remove discontinued products
  modelBuilder.Entity<Product>()
    .HasQueryFilter(p => !p.Discontinued);
}
```

2. Run the code, enter the partial product name `che`, view the result, and note that **Chef Anton's Gumbo Mix** is now missing, because the SQL statement generated includes a filter for the `Discontinued` column, as shown highlighted in the following output:

```
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
```

```
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND "p"."ProductName" LIKE @__Format_1
Chef Anton's Cajun Seasoning has 53 units in stock. Discontinued? False
Queso Manchego La Pastora has 86 units in stock. Discontinued? False
Gumbär Gummibärchen has 15 units in stock. Discontinued? False
```

Loading patterns with EF Core

There are three loading patterns that are commonly used with EF Core:

- **Eager loading**: Load data early.
- **Lazy loading**: Load data automatically just before it is needed.
- **Explicit loading**: Load data manually.

In this section, we're going to introduce each of them.

Eager loading entities

In the `QueryingCategories` method, the code currently uses the `Categories` property to loop through each category, outputting the category name and the number of products in that category.

This works because when we wrote the query, we enabled eager loading by calling the `Include` method for the related products.

Let's see what happens if we do not call `Include`:

1. Modify the query to comment out the `Include` method call, as shown in the following code:

```
IQueryable<Category>? categories =
  db.Categories; //.Include(c => c.Products);
```

2. In `Program.cs`, comment out all methods except `QueryingCategories`.
3. Run the code and view the result, as shown in the following partial output:

```
Beverages has 0 products.
Condiments has 0 products.
Confections has 0 products.
Dairy Products has 0 products.
Grains/Cereals has 0 products.
Meat/Poultry has 0 products.
Produce has 0 products.
Seafood has 0 products.
```


Each item in `foreach` is an instance of the `Category` class, which has a property named `Products`, that is, the list of products in that category. Since the original query is only selected from the `Categories` table, this property is empty for each category.

Enabling lazy loading

Lazy loading was introduced in EF Core 2.1, and it can automatically load missing related data. To enable lazy loading, developers must:

- Reference a NuGet package for proxies.
- Configure lazy loading to use a proxy.

Let's see this in action:

1. In the `WorkingWithEFCore` project, add a package reference for EF Core proxies, as shown in the following markup:

```
<PackageReference
  Include="Microsoft.EntityFrameworkCore.Proxies"
  Version="6.0.0" />
```

2. Build the project to restore packages.
3. Open `Northwind.cs`, and call an extension method to use lazy loading proxies at the top of the `OnConfiguring` method, as shown highlighted in the following code:

```
protected override void OnConfiguring(
  DbContextOptionsBuilder optionsBuilder)
{
  optionsBuilder.UseLazyLoadingProxies();
```

Now, every time the loop enumerates, and an attempt is made to read the `Products` property, the lazy loading proxy will check if they are loaded. If not, it will load them for us "lazily" by executing a `SELECT` statement to load just that set of products for the current category, and then the correct count will be returned to the output.

4. Run the code and note that the product counts are now correct. But you will see that the problem with lazy loading is that multiple round trips to the database server are required to eventually fetch all the data, as shown in the following partial output:

```
Categories and how many products they have:
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
SELECT "c"."CategoryId", "c"."CategoryName", "c"."Description" FROM
"Categories" AS "c"
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[@
p_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
```

```
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryId" = @_0)
Beverages has 12 products.
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[@
_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryId" = @_0)
Condiments has 12 products.
```

Explicit loading entities

Another type of loading is explicit loading. It works in a similar way to lazy loading, with the difference being that you are in control of exactly what related data is loaded and when:

1. At the top of Program.cs, import the change tracking namespace to enable us to use the CollectionEntry class to manually load related entities, as shown in the following code:

   ```
   using Microsoft.EntityFrameworkCore.ChangeTracking; // CollectionEntry
   ```

2. In the QueryingCategories method, modify the statements to disable lazy loading and then prompt the user as to whether they want to enable eager loading and explicit loading, as shown in the following code:

   ```
   IQueryable<Category>? categories;
     // = db.Categories;
     // .Include(c => c.Products);

   db.ChangeTracker.LazyLoadingEnabled = false;

   Write("Enable eager loading? (Y/N): ");
   bool eagerLoading = (ReadKey().Key == ConsoleKey.Y);
   bool explicitLoading = false;
   WriteLine();

   if (eagerLoading)
   {
     categories = db.Categories?.Include(c => c.Products);
   }
   else
   {
     categories = db.Categories;

     Write("Enable explicit loading? (Y/N): ");
     explicitLoading = (ReadKey().Key == ConsoleKey.Y);
     WriteLine();
   }
   ```


3. In the foreach loop, before the WriteLine method call, add statements to check if explicit loading is enabled, and if so, prompt the user as to whether they want to explicitly load each individual category, as shown in the following code:

   ```
   if (explicitLoading)
   {
     Write($"Explicitly load products for {c.CategoryName}? (Y/N): ");
     ConsoleKeyInfo key = ReadKey();
     WriteLine();
     if (key.Key == ConsoleKey.Y)
     {
       CollectionEntry<Category, Product> products =
         db.Entry(c).Collection(c2 => c2.Products);
       if (!products.IsLoaded) products.Load();
     }
   }
   WriteLine($"{c.CategoryName} has {c.Products.Count} products.");
   ```

4. Run the code:
   1. Press N to disable eager loading.
   2. Then press Y to enable explicit loading.
   3. For each category, press Y or N to load its products as you wish.

I chose to load products for only two of the eight categories, Beverages and Seafood, as shown in the following output that has been edited for space:

```
Categories and how many products they have:
Enable eager loading? (Y/N): N
Enable explicit loading? (Y/N): Y
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[],
CommandType='Text', CommandTimeout='30']
SELECT "c"."CategoryId", "c"."CategoryName", "c"."Description" FROM
"Categories" AS "c"
Explicitly load products for Beverages? (Y/N): Y
Level: Debug, Event Id: 20100, State: Executing DbCommand [Parameters=[@
_p_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryId" = @_p_0)
Beverages has 11 products.
Explicitly load products for Condiments? (Y/N): n
Condiments has 0 products.
```

```
Explicitly load products for Confections? (Y/N): n
Confections has 0 products.
Explicitly load products for Dairy Products? (Y/N): n
Dairy Products has 0 products.
Explicitly load products for Grains/Cereals? (Y/N): n
Grains/Cereals has 0 products.
Explicitly load products for Meat/Poultry? (Y/N): n
Meat/Poultry has 0 products.
Explicitly load products for Produce? (Y/N): n
Produce has 0 products.
Explicitly load products for Seafood? (Y/N): y
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[@
p_0='?'], CommandType='Text', CommandTimeout='30']
SELECT "p"."ProductId", "p"."CategoryId", "p"."UnitPrice",
"p"."Discontinued", "p"."ProductName", "p"."UnitsInStock"
FROM "Products" AS "p"
WHERE ("p"."Discontinued" = 0) AND ("p"."CategoryId" = @ p_0)
Seafood has 12 products.
```

Good Practice: Carefully consider which loading pattern is best for your code. Lazy loading could literally make you a lazy database developer! Read more about loading patterns at the following link: https://docs.microsoft.com/en-us/ef/core/querying/related-data

Manipulating data with EF Core

Inserting, updating, and deleting entities using EF Core is an easy task to accomplish.

`DbContext` maintains change tracking automatically, so the local entities can have multiple changes tracked, including adding new entities, modifying existing entities, and removing entities. When you are ready to send those changes to the underlying database, call the `SaveChanges` method. The number of entities successfully changed will be returned.

Inserting entities

Let's start by looking at how to add a new row to a table:

1. In `Program.cs`, create a new method named `AddProduct`, as shown in the following code:

```
static bool AddProduct(
  int categoryId, string productName, decimal? price)
{
  using (Northwind db = new())
  {
    Product p = new()
```



```
    {
      CategoryId = categoryId,
      ProductName = productName,
      Cost = price
    };

    // mark product as added in change tracking
    db.Products.Add(p);

    // save tracked change to database
    int affected = db.SaveChanges();
    return (affected == 1);
  }
}
```

2. In `Program.cs`, create a new method named `ListProducts` that outputs the Id, name, cost, stock, and discontinued properties of each product sorted with: the costliest first, as shown in the following code:

```
static void ListProducts()
{
  using (Northwind db = new())
  {
    WriteLine("{0,-3} {1,-35} {2,8} {3,5} {4}",
      "Id", "Product Name", "Cost", "Stock", "Disc.");

    foreach (Product p in db.Products
      .OrderByDescending(product => product.Cost))
    {
      WriteLine("{0:000} {1,-35} {2,8:$#,##0.00} {3,5} {4}",
        p.ProductId, p.ProductName, p.Cost, p.Stock, p.Discontinued);
    }
  }
}
```

Remember that `1,-35` means left-align argument 1 within a 35-character-wide column and `3,5` means right-align argument 3 within a 5-character-wide column.

3. In `Program.cs`, comment out previous method calls, and then call `AddProduct` and `ListProducts`, as shown in the following code:

```
// QueryingCategories();
// FilteredIncludes();
// QueryingProducts();
// QueryingWithLike();

if (AddProduct(categoryId: 6,
  productName: "Bob's Burgers", price: 500M))
```

```
{
  WriteLine("Add product successful.");
}

ListProducts();
```

4. Run the code, view the result, and note the new product has been added, as shown in the following partial output:

```
Add product successful.
Id  Product Name              Cost Stock Disc.
078 Bob's Burgers          $500.00       False
038 Côte de Blaye          $263.50    17 False
020 Sir Rodney's Marmalade 'S81.00    40 False
...
```

Updating entities

Now, let's modify an existing row in a table:

1. In `Program.cs`, add a method to increase the price of the first product with a name that begins with a specified value (we'll use Bob in our example) by a specified amount like $20, as shown in the following code:

```
static bool IncreaseProductPrice(
  string productNameStartsWith, decimal amount)
{
  using (Northwind db = new())
  {
    // get first product whose name starts with name
    Product updateProduct = db.Products.First(
      p => p.ProductName.StartsWith(productNameStartsWith));

    updateProduct.Cost += amount;

    int affected = db.SaveChanges();
    return (affected == 1);
  }
}
```

2. In `Program.cs`, comment out the whole `if` block that calls `AddProduct`, and add a call to `IncreaseProductPrice` before the call to list products, as shown highlighted in the following code:

```
/*
if (AddProduct(categoryId: 6,
  productName: "Bob's Burgers", price: 500M))
{
  WriteLine("Add product successful.");
}
*/

if (IncreaseProductPrice(
  productNameStartsWith: "Bob", amount: 20M))
{
  WriteLine("Update product price successful.");
}

ListProducts();
```

3. Run the code, view the result, and note that the existing entity for Bob's Burgers has increased in price by $20, as shown in the following partial output:

```
Update product price successful.
Id  Product Name              Cost Stock Disc.
078 Bob's Burgers          $520.00       False
038 Côte de Blaye          $263.50    17 False
020 Sir Rodney's Marmalade $81.00     40 False
...
```

Deleting entities

You can remove individual entities with the `Remove` method. `RemoveRange` is more efficient when you want to delete multiple entities.

Now let's see how to delete rows from a table:

1. At the bottom of `Program.cs`, add a method to delete all products with a name that begins with a specified value (Bob in our example), as shown in the following code:

```
static int DeleteProducts(string productNameStartsWith)
{
  using (Northwind db = new())
  {
    IQueryable<Product>? products = db.Products?.Where(
      p => p.ProductName.StartsWith(productNameStartsWith));

    if (products is null)
    {
      WriteLine("No products found to delete.");
      return 0;
    }
    else
    {
      db.Products.RemoveRange(products);
```

```
    }

    int affected = db.SaveChanges();
    return affected;
  }
}
```

2. In `Program.cs`, comment out the whole `if` statement block that calls `IncreaseProductPrice`, and add a call to `DeleteProducts`, as shown in the following code:

```
int deleted = DeleteProducts(productNameStartsWith: "Bob");
WriteLine($"{deleted} product(s) were deleted.");
```

3. Run the code and view the result, as shown in the following output:

```
1 product(s) were deleted.
```

If multiple product names started with Bob, then they are all deleted. As an optional challenge, modify the statements to add three new products that start with Bob and then delete them.

Pooling database contexts

The `DbContext` class is disposable and is designed following the single-unit-of-work principle. In the previous code examples, we created all the `DbContext`-derived Northwind instances in a `using` block so that `Dispose` is properly called at the end of each unit of work.

A feature of ASP.NET Core that is related to EF Core is that it makes your code more efficient by pooling database contexts when building websites and services. This allows you to create and dispose of as many `DbContext`-derived objects as you want, knowing that your code is still as efficient as possible.

Working with transactions

Every time you call the `SaveChanges` method, an **implicit transaction** is started so that if something goes wrong, it will automatically roll back all the changes. If the multiple changes within the transaction succeed, then the transaction and all changes are committed.

Transactions maintain the integrity of your database by applying locks to prevent reads and writes while a sequence of changes is occurring.

Transactions are **ACID**, which is an acronym explained in the following list:

- **A is for atomic**. Either all the operations in the transaction commit, or none of them do.
- **C is for consistent**. The state of the database before and after a transaction is consistent. This is dependent on your code logic; for example, when transferring money between bank accounts, it is up to your business logic to ensure that if you debit $100 in one account, you credit $100 in the other account.

- **I is for isolated**. During a transaction, changes are hidden from other processes. There are multiple isolation levels that you can pick from (refer to the following table). The stronger the level, the better the integrity of the data. However, more locks must be applied, which will negatively affect other processes. Snapshot is a special case because it creates multiple copies of rows to avoid locks, but this will increase the size of your database while transactions occur.
- **D is for durable**. If a failure occurs during a transaction, it can be recovered. This is often implemented as a two-phase commit and transaction logs. Once the transaction is committed it is guaranteed to endure even if there are subsequent errors. The opposite of durable is volatile.

Controlling transactions using isolation levels

A developer can control transactions by setting an **isolation level**, as described in the following table:

Isolation level	Lock(s)	Integrity problems allowed
`ReadUncommitted`	None	Dirty reads, nonrepeatable reads, and phantom data
`ReadCommitted`	When editing, it applies read lock(s) to block other users from reading the record(s) until the transaction ends	Nonrepeatable reads and phantom data
`RepeatableRead`	When reading, it applies edit lock(s) to block other users from editing the record(s) until the transaction ends	Phantom data
`Serializable`	Applies key-range locks to prevent any action that would affect the results, including inserts and deletes	None
`Snapshot`	None	None

Defining an explicit transaction

You can control explicit transactions using the `Database` property of the database context:

1. In `Program.cs`, import the EF Core storage namespace to use the `IDbContextTransaction` interface, as shown in the following code:

```
using Microsoft.EntityFrameworkCore.Storage; // IDbContextTransaction
```

2. In the `DeleteProducts` method, after the instantiation of the `db` variable, add statements to start an explicit transaction and output its isolation level. At the bottom of the method, commit the transaction, and close the brace, as shown highlighted in the following code:

```
static int DeleteProducts(string name)
{
```

```
using (Northwind db = new())
{
  using (IDbContextTransaction t = db.Database.BeginTransaction())
  {
    WriteLine("Transaction isolation level: {0}",
      arg0: t.GetDbTransaction().IsolationLevel);

    IQueryable<Product>? products = db.Products?.Where(
      p => p.ProductName.StartsWith(name));

    if (products is null)
    {
      WriteLine("No products found to delete.");
      return 0;
    }
    else
    {
      db.Products.RemoveRange(products);
    }

    int affected = db.SaveChanges();
    t.Commit();
    return affected;
  }
}
}
```

3. Run the code and view the result using SQL Server, as shown in the following output:

```
Transaction isolation level: ReadCommitted
```

4. Run the code and view the result using SQLite, as shown in the following output:

```
Transaction isolation level: Serializable
```

Code First EF Core models

Sometimes you will not have an existing database. Instead, you define the EF Core model as Code First, and then EF Core can generate a matching database using create and drop APIs.

Good Practice: The create and drop APIs should only be used during development. Once you release the app, you do not want it to delete a production database!

For example, we might need to create an application for managing students and courses for an academy. One student can sign up to attend multiple courses. One course can be attended by multiple students. This is an example of a many-to-many relationship between students and courses.

Let's model this example:

1. Use your preferred code editor to add a new console app named `CoursesAndStudents` to the `Chapter10` solution/workspace.
2. In Visual Studio, set the startup project for the solution to the current selection.
3. In Visual Studio Code, select `CoursesAndStudents` as the active OmniSharp project.
4. In the `CoursesAndStudents` project, add package references for the following packages:
   - `Microsoft.EntityFrameworkCore.Sqlite`
   - `Microsoft.EntityFrameworkCore.SqlServer`
   - `Microsoft.EntityFrameworkCore.Design`
5. Build the `CoursesAndStudents` project to restore packages.
6. Add classes named `Academy.cs`, `Student.cs`, and `Course.cs`.
7. Modify `Student.cs`, and note that it is a POCO (plain old CLR object) with no attributes decorating the class, as shown in the following code:

```
namespace CoursesAndStudents;

public class Student
{
  public int StudentId { get; set; }
  public string? FirstName { get; set; }
  public string? LastName { get; set; }

  public ICollection<Course>? Courses { get; set; }
}
```

8. Modify `Course.cs`, and note that we have decorated the `Title` property with some attributes to provide more information to the model, as shown in the following code:

```
using System.ComponentModel.DataAnnotations;

namespace CoursesAndStudents;

public class Course
{
  public int CourseId { get; set; }
```

In the following six chapters, you will learn the details about how to build the following:

- Simple websites using static HTML pages and dynamic Razor Pages.
- Complex websites using the Model-View-Controller (MVC) design pattern.
- Web services that can be called by any platform that can make an HTTP request and client websites that call those web services.
- Blazor user interface components that can be hosted on a web server, in the browser, or on hybrid web-native mobile and desktop apps.
- Services that implement remote procedure calls using gRPC.
- Services that implement real-time communication using SignalR.
- Services that provide easy and flexible access to an EF Core model.
- Serverless nano services hosted in Azure Functions.
- Cross-platform native mobile and desktop apps using .NET MAUI.

14

Building Websites Using ASP.NET Core Razor Pages

This chapter is about building websites with a modern HTTP architecture on the server side using Microsoft ASP.NET Core. You will learn about building simple websites using the ASP.NET Core Razor Pages feature introduced with ASP.NET Core 2.0 and the Razor class library feature introduced with ASP.NET Core 2.1.

This chapter will cover the following topics:

- Understanding web development
- Understanding ASP.NET Core
- Exploring ASP.NET Core Razor Pages
- Using Entity Framework Core with ASP.NET Core
- Using Razor class libraries
- Configuring services and the HTTP request pipeline

Understanding web development

Developing for the web means developing with **Hypertext Transfer Protocol (HTTP)**, so we will start by reviewing this important foundational technology.

Understanding HTTP

To communicate with a web server, the client, also known as the **user agent**, makes calls over the network using HTTP. As such, HTTP is the technical underpinning of the web. So, when we talk about websites and web services, we mean that they use HTTP to communicate between a client (often a web browser) and a server.

A client makes an HTTP request for a resource, such as a page, uniquely identified by a **Uniform Resource Locator (URL)**, and the server sends back an HTTP response, as shown in *Figure 14.1*:

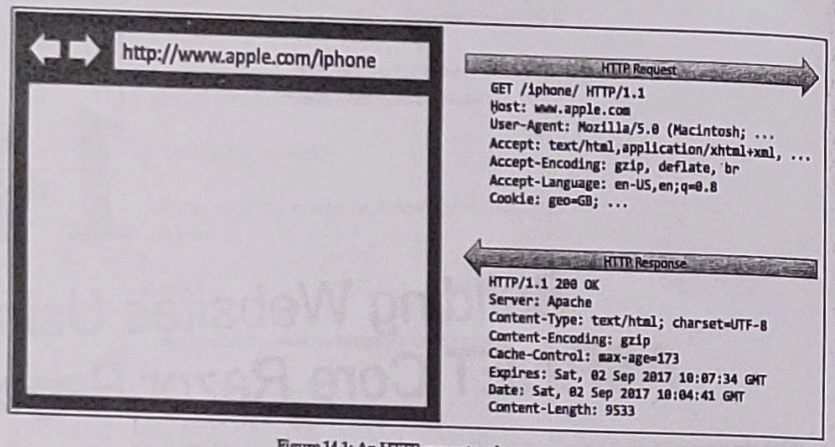

Figure 14.1: An HTTP request and response

You can use Google Chrome and other browsers to record requests and responses.

Good Practice: Google Chrome is available on more operating systems than any other browser, and it has powerful, built-in developer tools, so it is a good first choice of browser for testing your websites. Always test your web application with Chrome and at least two other browsers, for example, Firefox and Safari for macOS and iPhone. Microsoft Edge switched from using Microsoft's own rendering engine to using Chromium in 2019, so it is less important to test with it. If Microsoft's Internet Explorer is used at all, it tends to mostly be inside organizations for intranets.

Understanding the components of a URL

A URL is made up of several components:

- **Scheme**: `http` (clear text) or `https` (encrypted).
- **Domain**: For a production website or service, the **top-level domain (TLD)** might be `example.com`. You might have subdomains such as `www`, `jobs`, or `extranet`. During development, you typically use `localhost` for all websites and services.

- **Port number**: For a production website or service, `80` for `http`, `443` for `https`. These port numbers are usually inferred from the scheme. During development, other port numbers are commonly used, such as `5000`, `5001`, and so on, to differentiate between websites and services that all use the shared domain `localhost`.
- **Path**: A relative path to a resource, for example, `/customers/germany`.
- **Query string**: A way to pass parameter values, for example, `?country=Germany&searchtext=shoes`.
- **Fragment**: A reference to an element on a web page using its `id`, for example, `#toc`.

Assigning port numbers for projects in this book

In this book, we will use the domain `localhost` for all websites and services, so we will use port numbers to differentiate projects when multiple need to execute at the same time, as shown in the following table:

Project	Description	Port numbers
`Northwind.Web`	ASP.NET Core Razor Pages website	`5000` HTTP, `5001` HTTPS
`Northwind.Mvc`	ASP.NET Core MVC website	`5000` HTTP, `5001` HTTPS
`Northwind.WebApi`	ASP.NET Core Web API service	`5002` HTTPS, `5008` HTTP
`Minimal.WebApi`	ASP.NET Core Web API (minimal)	`5003` HTTPS
`Northwind.OData`	ASP.NET Core OData service	`5004` HTTPS
`Northwind.GraphQL`	ASP.NET Core GraphQL service	`5005` HTTPS
`Northwind.gRPC`	ASP.NET Core gRPC service	`5006` HTTPS
`Northwind.AzureFuncs`	Azure Functions nanoservice	`7071` HTTP

Using Google Chrome to make HTTP requests

Let's explore how to use Google Chrome to make HTTP requests:

1. Start Google Chrome.
2. Navigate to **More tools | Developer tools**.

3. Click the **Network** tab, and Chrome should immediately start recording the network traffic between your browser and any web servers (note the red circle), as shown in *Figure 14.2*:

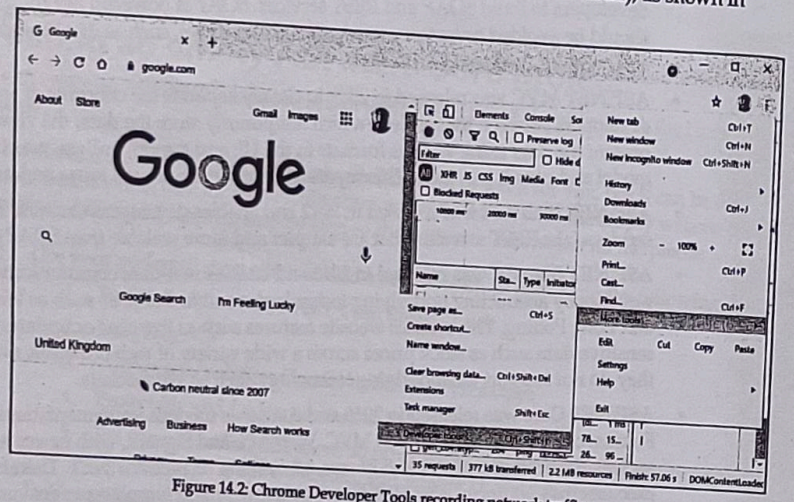

Figure 14.2: Chrome Developer Tools recording network traffic

4. In Chrome's address box, enter the address of Microsoft's website for learning ASP.NET, as shown in the following URL:

```
https://dotnet.microsoft.com/learn/aspnet
```

5. In Developer Tools, in the list of recorded requests, scroll to the top and click on the first entry, the row where the **Type** is **document**, as shown in *Figure 14.3*:

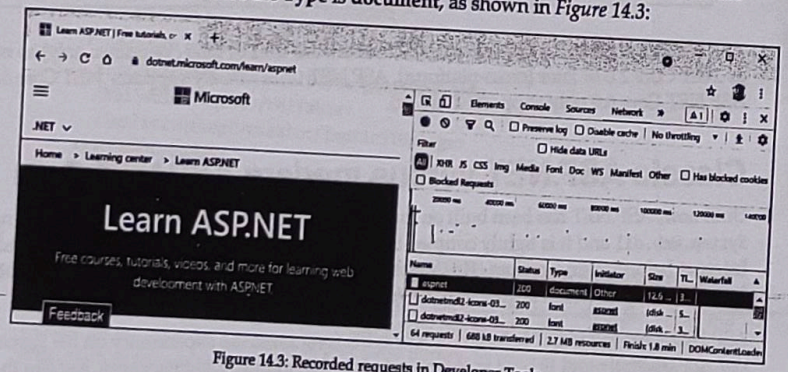

Figure 14.3: Recorded requests in Developer Tools

6. On the right-hand side, click on the **Headers** tab, and you will see details about **Request Headers** and **Response Headers**, as shown in *Figure 14.4*:

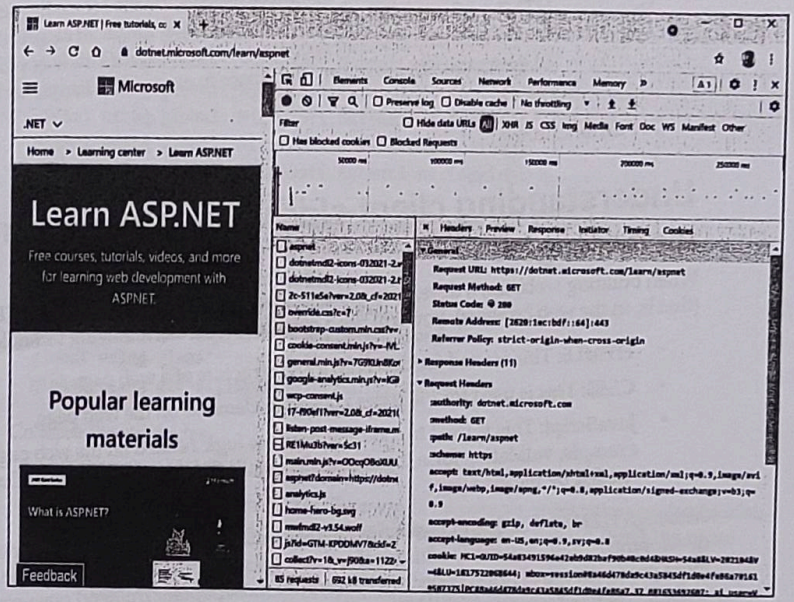

Figure 14.4: Request and response headers

Note the following aspects:

- **Request Method** is `GET`. Other HTTP methods that you could see here include `POST`, `PUT`, `DELETE`, `HEAD`, and `PATCH`.
- **Status Code** is `200 OK`. This means that the server found the resource that the browser requested and has returned it in the body of the response. Other status codes that you might see in response to a `GET` request include `301 Moved Permanently`, `400 Bad Request`, `401 Unauthorized`, and `404 Not Found`.
- **Request Headers** sent by the browser to the web server include:
  - **accept**, which lists what formats the browser accepts. In this case, the browser is saying it understands HTML, XHTML, XML, and some image formats, but it will accept all other files (`*/*`). Default weightings, also known as quality values, are `1.0`. XML is specified with a quality value of `0.9` so it is preferred less than HTML or XHTML. All other file types are given a quality value of `0.8` so are least preferred.
  - **accept-encoding**, which lists what compression algorithms the browser understands, in this case, GZIP, DEFLATE, and Brotli.
  - **accept-language**, which lists the human languages it would prefer the content to use. In this case, US English, which has a default quality value of `1.0`, then any dialect of English that has an explicitly specified quality value of `0.9`, and then any dialect of Swedish that has an explicitly specified quality value of `0.8`.

- **Response Headers**, `content-encoding` tells me the server has sent back the HTML web page response compressed using the GZIP algorithm because it knows that the client can decompress that format. (This is not visible in *Figure 14.4* because there is not enough space to expand the **Response Headers** section.)

7. Close Chrome.

Understanding client-side web development technologies

When building websites, a developer needs to know more than just C# and .NET. On the client (that is, in the web browser), you will use a combination of the following technologies:

- **HTML5**: This is used for the content and structure of a web page.
- **CSS3**: This is used for the styles applied to elements on the web page.
- **JavaScript**: This is used to code any business logic needed on the web page, for example, validating form input or making calls to a web service to fetch more data needed by the web page.

Although HTML5, CSS3, and JavaScript are the fundamental components of frontend web development, there are many additional technologies that can make frontend web development more productive, including Bootstrap, the world's most popular frontend open-source toolkit, and CSS preprocessors such as SASS and LESS for styling, Microsoft's TypeScript language for writing more robust code, and JavaScript libraries such as jQuery, Angular, React, and Vue. All these higher-level technologies ultimately translate or compile to the underlying three core technologies, so they work across all modern browsers.

As part of the build and deploy process, you will likely use technologies such as Node.js; Node Package Manager (npm) and Yarn, which are both client-side package managers; and webpack, which is a popular module bundler, a tool for compiling, transforming, and bundling website source files.

Understanding ASP.NET Core

Microsoft ASP.NET Core is part of a history of Microsoft technologies used to build websites and services that have evolved over the years:

- **Active Server Pages (ASP)** was released in 1996 and was Microsoft's first attempt at a platform for dynamic server-side execution of website code. ASP files contain a mix of HTML and code that executes on the server written in the VBScript language.
- **ASP.NET Web Forms** was released in 2002 with the .NET Framework and was designed to enable non-web developers, such as those familiar with Visual Basic, to quickly create websites by dragging and dropping visual components and writing event-driven code in Visual Basic or C#. Web Forms should be avoided for new .NET Framework web projects in favor of ASP.NET MVC.

- **Windows Communication Foundation (WCF)** was released in 2006 and enables developers to build SOAP and REST services. SOAP is powerful but complex, so it should be avoided unless you need advanced features, such as distributed transactions and complex messaging topologies.
- **ASP.NET MVC** was released in 2009 to cleanly separate the concerns of web developers between the **models**, which temporarily store the data; the **views**, which present the data using various formats in the UI; and the **controllers**, which fetch the model and pass it to a view. This separation enables improved reuse and unit testing.
- **ASP.NET Web API** was released in 2012 and enables developers to create HTTP services (aka REST services) that are simpler and more scalable than SOAP services.
- **ASP.NET SignalR** was released in 2013 and enables real-time communication in websites by abstracting underlying technologies and techniques, such as WebSockets and Long Polling. This enables website features such as live chat or updates to time-sensitive data such as stock prices across a wide variety of web browsers, even when they do not support an underlying technology such as WebSockets.
- **ASP.NET Core** was released in 2016 and combines modern implementations of .NET Framework technologies such as MVC, Web API, and SignalR, with newer technologies such as Razor Pages, gRPC, and Blazor, all running on modern .NET. Therefore, it can execute cross-platform. ASP.NET Core has many project templates to get you started with its supported technologies.

Good Practice: Choose ASP.NET Core to develop websites and services because it includes web-related technologies that are modern and cross-platform.

ASP.NET Core 2.0 to 2.2 can run on .NET Framework 4.6.1 or later (Windows only) as well as .NET Core 2.0 or later (cross-platform). ASP.NET Core 3.0 only supports .NET Core 3.0. ASP.NET Core 6.0 only supports .NET 6.0.

Classic ASP.NET versus modern ASP.NET Core

Until now, ASP.NET has been built on top of a large assembly in the .NET Framework named `System.Web.dll` and it is tightly coupled to Microsoft's Windows-only web server named **Internet Information Services (IIS)**. Over the years, this assembly has accumulated a lot of features, many of which are not suitable for modern cross-platform development.

ASP.NET Core is a major redesign of ASP.NET. It removes the dependency on the `System.Web.dll` assembly and IIS and is composed of modular lightweight packages, just like the rest of modern .NET. Using IIS as the web server is still supported by ASP.NET Core but there is a better option.

You can develop and run ASP.NET Core applications cross-platform on Windows, macOS, and Linux. Microsoft has even created a cross-platform, super-performant web server named **Kestrel**, and the entire stack is open source.

ASP.NET Core 2.2 or later projects default to the new in-process hosting model. This gives a 400% performance improvement when hosting in Microsoft IIS, but Microsoft still recommends using Kestrel for even better performance.

Creating an empty ASP.NET Core project

We will create an ASP.NET Core project that will show a list of suppliers from the Northwind database.

The dotnet tool has many project templates that do a lot of work for you, but it can be difficult to know which works best for a given situation, so we will start with the empty website project template and then add features step by step so that you can understand all the pieces:

1. Use your preferred code editor to add a new project, as defined in the following list:
    1. Project template: **ASP.NET Core Empty** / `web`
    2. Language: C#
    3. Workspace/solution file and folder: `PracticalApps`
    4. Project file and folder: `Northwind.Web`
    5. For Visual Studio 2022, leave all other options as their defaults, for example, **Configure for HTTPS** selected, and **Enable Docker** cleared
2. In Visual Studio Code, select `Northwind.Web` as the active OmniSharp project.
3. Build the `Northwind.Web` project.
4. Open the `Northwind.Web.csproj` file and note that the project is like a class library except that the SDK is `Microsoft.NET.Sdk.Web`, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

5. If you are using Visual Studio 2022, in **Solution Explorer**, toggle **Show All Files**.
6. Expand the `obj` folder, expand the `Debug` folder, expand the `net6.0` folder, select the `Northwind.Web.GlobalUsings.g.cs` file, and note the implicitly imported namespaces include all the ones for a console app or class library, as well as some ASP.NET Core ones, such as `Microsoft.AspNetCore.Builder`, as shown in the following code:

```
// <autogenerated />
global using global::Microsoft.AspNetCore.Builder;
global using global::Microsoft.AspNetCore.Hosting;
global using global::Microsoft.AspNetCore.Http;
global using global::Microsoft.AspNetCore.Routing;
global using global::Microsoft.Extensions.Configuration;
global using global::Microsoft.Extensions.DependencyInjection;
global using global::Microsoft.Extensions.Hosting;
global using global::Microsoft.Extensions.Logging;
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Net.Http.Json;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

7. Collapse the `obj` folder.
8. Open `Program.cs`, and note the following:
    - An ASP.NET Core project is like a top-level console application, with a hidden `Main` method as its entry point that has an argument passed using the name `args`.
    - It calls `WebApplication.CreateBuilder`, which creates a host for the website using defaults for a web host that is then built.
    - The website will respond to all HTTP `GET` requests with plain text: `Hello World!`.
    - The call to the `Run` method is a blocking call, so the hidden `Main` method does not return until the web server stops running, as shown in the following code:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

9. At the bottom of the file, add a statement to write a message to the console after the call to the `Run` method and therefore after the web server has stopped, as shown highlighted in the following code:

```
app.Run();

Console.WriteLine("This executes after the web server has stopped!");
```

Testing and securing the website

We will now test the functionality of the ASP.NET Core Empty website project. We will also enable encryption of all traffic between the browser and web server for privacy by switching from HTTP to HTTPS. HTTPS is the secure encrypted version of HTTP.

1. For Visual Studio:
   1. In the toolbar, make sure that Northwind.Web is selected rather than IIS Express or WSL, and switch the Web Browser (Microsoft Edge) to Google Chrome, as shown in *Figure 14.5*:

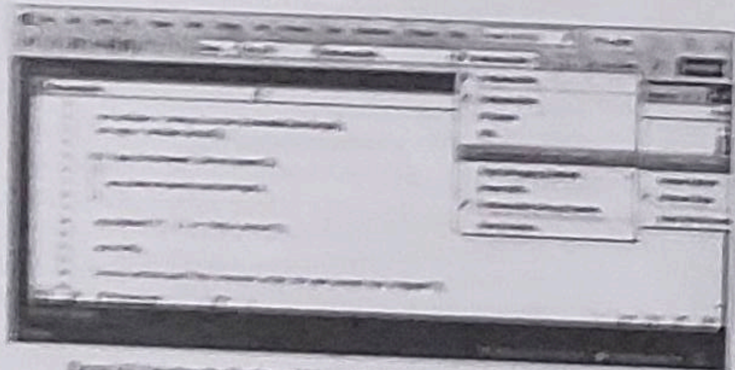

Figure 14.5: Selecting the Northwind.Web profile with its Kestrel web server in Visual Studio

   2. Navigate to **Debug** | **Start Without Debugging...**.
   3. The first time you start a secure website, you will be prompted that your project is configured to use SSL, and to avoid warnings in the browser you can choose to trust the self-signed certificate that ASP.NET Core has generated. Click **Yes**.
   4. When you see the **Security Warning** dialog box, click **Yes** again.
2. For Visual Studio Code, in **TERMINAL**, enter the `dotnet run` command.
3. In either Visual Studio's command prompt window or Visual Studio Code's terminal, note the Kestrel web server has started listening on random ports for HTTP and HTTPS, that you can press Ctrl + C to shut down the Kestrel web server, and the hosting environment is Development, as shown in the following output:

```
info: Microsoft.Hosting.Lifetime[14]
      [illegible]
info: Microsoft.Hosting.Lifetime[14]
      [illegible]
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Code\PracticalApps\Northwind.Web
```

> Visual Studio will also start your chosen browser automatically. If you are using Visual Studio Code, you will have to start Chrome manually.

4. Leave the web server running.
5. In Chrome, show **Developer Tools**, and click the **Network** tab.
6. Enter the address `http://localhost:5000/`, or whatever port number was assigned to HTTP, and note the response is `Hello World!` in plain text, from the cross-platform Kestrel web server, as shown in *Figure 14.6*:

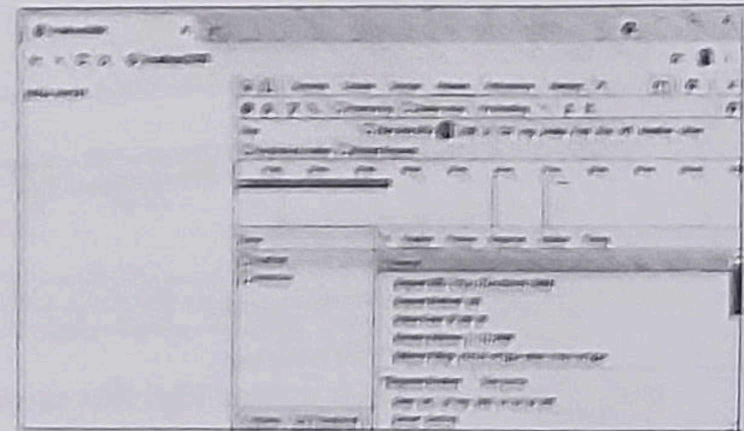

Figure 14.6: Plain text response from http://localhost:5000/

> Chrome also requests a `favicon.ico` file automatically to show in the browser tab but this is missing so it shows as a `404 Not Found` error.

7. Enter the address `https://localhost:5001/`, or whatever port number was assigned to HTTPS, and note if you are not using Visual Studio or if you clicked **No** when prompted to trust the SSL certificate, then the response is a privacy error, as shown in *Figure 14.7*:

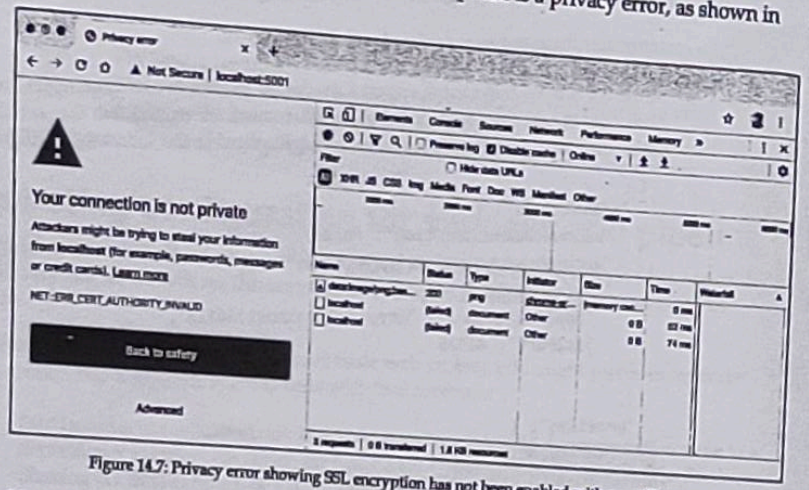

Figure 14.7: Privacy error showing SSL encryption has not been enabled with a certificate

You will see this error when you have not configured a certificate that the browser can trust to encrypt and decrypt HTTPS traffic (and so if you do not see this error, it is because you have already configured a certificate).

In a production environment, you would want to pay a company such as Verisign for an SSL certificate because they provide liability protection and technical support.

> **For Linux Developers**: If you use a Linux variant that cannot create self-signed certificates or you do not mind reapplying for a new certificate every 90 days, then you can get a free certificate from the following link: `https://letsencrypt.org`

During development, you can tell your OS to trust a temporary development certificate provided by ASP.NET Core.

8. At the command line or in **TERMINAL**, press *Ctrl + C* to shut down the web server, and note the message that is written, as shown highlighted in the following output:

```
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
This executes after the web server has stopped!

C:\Code\PracticalApps\Northwind.Web\bin\Debug\net6.0\Northwind.Web.exe
(process 19888) exited with code 0.
```


9. If you need to trust a local self-signed SSL certificate, then at the command line or in **TERMINAL**, enter the `dotnet dev-certs https --trust` command, and note the message, **Trusting the HTTPS development certificate was requested**. You might be prompted to enter your password and a valid HTTPS certificate may already be present.

Enabling stronger security and redirect to a secure connection

It is good practice to enable stricter security and automatically redirect requests for HTTP to HTTPS.

> **Good Practice: HTTP Strict Transport Security (HSTS)** is an opt-in security enhancement that you should always enable. If a website specifies it and a browser supports it, then it forces all communication over HTTPS and prevents the visitor from using untrusted or invalid certificates.

Let's do that now:

1. In `Program.cs`, add an `if` statement to enable HSTS when not in development, as shown in the following code:

```
if (!app.Environment.IsDevelopment())
{
  app.UseHsts();
}
```

2. Add a statement before the call to `app.MapGet` to redirect HTTP requests to HTTPS, as shown in the following code:

```
app.UseHttpsRedirection();
```

3. Start the **Northwind.Web** website project.
4. If Chrome is still running, close and restart it.
5. In Chrome, show **Developer Tools**, and click the **Network** tab.

6. Enter the address `http://localhost:5000/`, or whatever port number was assigned to HTTP, and note how the server responds with a `307 Temporary Redirect` to port `5001` and that the certificate is now valid and trusted, as shown in *Figure 14.8*:

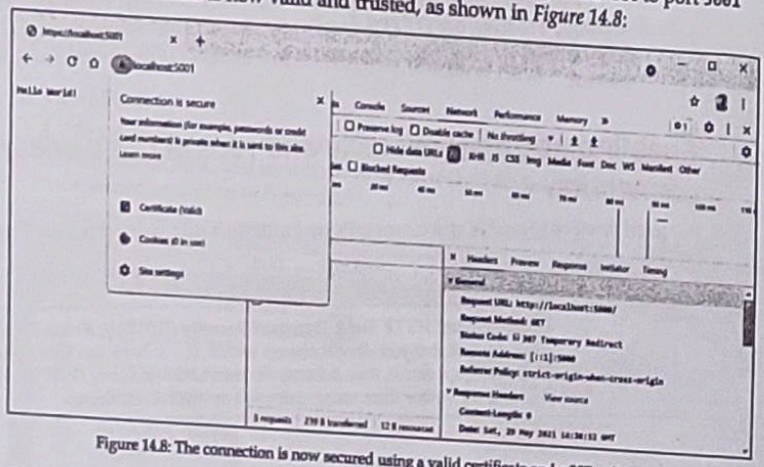

Figure 14.8: The connection is now secured using a valid certificate and a 307 redirect

7. Close Chrome.
8. Shut down the web server.

Good Practice: Remember to shut down the Kestrel web server whenever you have finished testing a website.

Controlling the hosting environment

In earlier versions of ASP.NET Core, the project template set a rule to say that while in development mode, any unhandled exceptions will be shown in the browser window for the developer to see the details of the exception, as shown in the following code:

```
if (app.Environment.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
}
```

With ASP.NET Core 6 and later, this code is executed automatically by default so it is not included in the project template.

How does ASP.NET Core know when we are running in development mode so that the `IsDevelopment` method returns `true`? Let's find out.

ASP.NET Core can read from environment variables to determine what hosting environment to use, for example, `DOTNET_ENVIRONMENT` or `ASPNETCORE_ENVIRONMENT`.

You can override these settings during local development:

1. In the `Northwind.Web` folder, expand the folder named `Properties`, open the file named `launchSettings.json`, and note the profile named `Northwind.Web` that sets the hosting environment to `Development`, as shown highlighted in the following configuration:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:56111",
      "sslPort": 44329
    }
  },
  "profiles": {
    "Northwind.Web": {
      "commandName": "Project",
      "dotnetRunMessages": "true",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

2. Change the randomly assigned port numbers for HTTP to `5000` and HTTPS to `5001`.
3. Change the environment to `Production`. Optionally, change `launchBrowser` to `false` to prevent Visual Studio from automatically launching a browser.
4. Start the website and note the hosting environment is `Production`, as shown in the following output:

```
info: Microsoft.Hosting.Lifetime[0]
  Hosting environment: Production
```

5. Shut down the web server.
6. In `launchSettings.json`, change the environment back to `Development`.

The `launchSettings.json` file also has a configuration for using IIS as the web server using random port numbers. In this book, we will only be using Kestrel as the web server since it is cross-platform.

Separating configuration for services and pipeline

Putting all code to initialize a simple web project in `Program.cs` can be a good idea, especially for web services, so we will see this style again in *Chapter 16, Building and Consuming Web Services*.

However, for anything more than the most basic web project, you might prefer to separate configuration into a separate `Startup` class with two methods:

- `ConfigureServices(IServiceCollection services)`: to add dependency services to a dependency injection container, such as Razor Pages support, **Cross-Origin Resource Sharing (CORS)** support, or a database context for working with the Northwind database.
- `Configure(IApplicationBuilder app, IWebHostEnvironment env)`: to set up the HTTP pipeline through which requests and responses flow. Call various `Use` methods on the `app` parameter to construct the pipeline in the order the features should be processed.

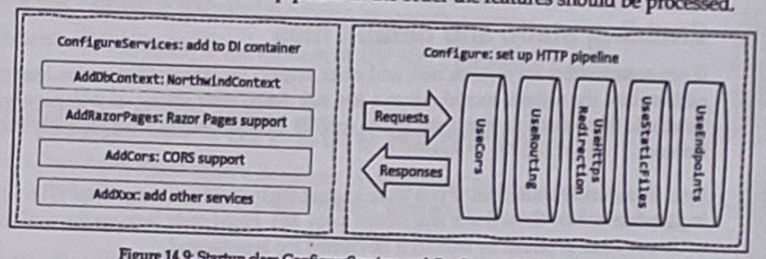

Figure 14.9: Startup class ConfigureServices and Configure methods diagram

Both methods will get called automatically by the runtime.

Let's create a `Startup` class now:

1. Add a new class file to the `Northwind.Web` project named `Startup.cs`.
2. Modify `Startup.cs`, as shown in the following code:

```
namespace Northwind.Web;

public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
  }

  public void Configure(
    IApplicationBuilder app, IWebHostEnvironment env)
  {
    if (!env.IsDevelopment())
    {
      app.UseHsts();
    }

    app.UseRouting(); // start endpoint routing

    app.UseHttpsRedirection();

    app.UseEndpoints(endpoints =>
    {
      endpoints.MapGet("/", () => "Hello World!");
    });
  }
}
```

Note the following about the code:

- The `ConfigureServices` method is currently empty because we do not yet need any dependency services added.
- The `Configure` method sets up the HTTP request pipeline and enables the use of endpoint routing. It configures a routed endpoint to wait for requests using the same map for each HTTP `GET` request for the root path `/` that responds to those requests by returning the plain text `"Hello World!"`. We will learn about routed endpoints and their benefits at the end of this chapter.

Now we must specify that we want to use the `Startup` class in the application entry point.

3. Modify `Program.cs`, as shown in the following code:

```
using Northwind.Web; // Startup

Host.CreateDefaultBuilder(args)
  .ConfigureWebHostDefaults(webBuilder =>
  {
    webBuilder.UseStartup<Startup>();
  }).Build().Run();

Console.WriteLine("This executes after the web server has stopped!");
```

4. Start the website and note that it has the same behavior as before.
5. Shut down the web server.

In all the other website and service projects that we create in this book, we will use the single `Program.cs` file created by .NET 6 project templates. If you like the `Startup.cs` way of doing things, then you will see in this chapter how to use it.

Enabling a website to serve static content

A website that only ever returns a single plain text message isn't very useful!

At a minimum, it ought to return static HTML pages, CSS that the web pages will use for styling, and any other static resources, such as images and videos.

By convention, these files should be stored in a directory named `wwwroot` to keep them separate from the dynamically executing parts of your website project.

Creating a folder for static files and a web page

You will now create a folder for your static website resources and a basic index page that uses Bootstrap for styling:

1. In the `Northwind.Web` project/folder, create a folder named `wwwroot`.
2. Add a new HTML page file to the `wwwroot` folder named `index.html`.
3. Modify its content to link to CDN-hosted Bootstrap for styling, and use modern good practices such as setting the viewport, as shown in the following markup:

```
<!doctype html>
<html lang="en">

<head>
  <!-- Required meta tags -->
  <meta charset="utf-8" />
  <meta name="viewport" content=
    "width=device-width, initial-scale=1 " />

  <!-- Bootstrap CSS -->
  <link href=
"https://cdn.jsdelivr.net/npm/bootstrap@5.1.0/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-KyZXEAg3QhqLMpG8r+8fhAXLRk2vvoC2f3B09zV
XnSCA5QIVfZOJ3BCsw2P0p/We" crossorigin="anonymous">

  <title>Welcome ASP.NET Core!</title>
</head>
```



```
<body>
  <div class="container">
    <div class="jumbotron">
      <h1 class="display-3">Welcome to Northwind B2B</h1>
      <p class="lead">We supply products to our customers.</p>
      <hr />
      <h2>This is a static HTML page.</h2>
      <p>Our customers include restaurants, hotels, and cruise lines.</p>
      <p>
        <a class="btn btn-primary"
          href="https://www.asp.net/">Learn more</a>
      </p>
    </div>
  </div>
</body>

</html>
```

Good Practice: To get the latest `<link>` element for Bootstrap, copy and paste it from the documentation at the following link: `https://getbootstrap.com/docs/5.0/getting-started/introduction/#starter-template`.

Enabling static and default files

If you were to start the website now and enter `http://localhost:5000/index.html` in the address box, the website would return a `404 Not Found` error saying no web page was found. To enable the website to return static files such as `index.html`, we must explicitly configure that feature.

Even if we enable static files, if you were to start the website and enter `http://localhost:5000/` in the address box, the website will return a `404 Not Found` error because the web server does not know what to return by default if no named file is requested.

You will now enable static files, explicitly configure default files, and change the URL path registered that returns the plain text `Hello World!` response:

1. In `Startup.cs`, in the `Configure` method, add statements after enabling HTTPS redirection to enable static files and default files, and modify the statement that maps a `GET` request to return the `Hello World!` plain text response to only respond to the URL path `/hello`, as shown highlighted in the following code:

```
app.UseHttpsRedirection();

app.UseDefaultFiles(); // index.html, default.html, and so on
```

```
app.UseStaticFiles();

app.UseEndpoints(endpoints =>
{
  endpoints.MapGet("/hello", () => "Hello World!");
});
```

The call to `UseDefaultFiles` must come before the call to `UseStaticFiles`, or it will not work! You will learn more about the ordering of middleware and endpoint routing at the end of this chapter.

2. Start the website.
3. Start **Chrome** and show **Developer Tools**.
4. In Chrome, enter `http://localhost:5000/` and note that you are redirected to the HTTPS address on port `5001`, and the `index.html` file is now returned over that secure connection because it is one of the possible default files for this website.
5. In **Developer Tools**, note the request for the Bootstrap stylesheet.
6. In Chrome, enter `http://localhost:5000/hello` and note that it returns the plain text `Hello World!` as before.
7. Close Chrome and shut down the web server.

If all web pages are static, that is, they only get changed manually by a web editor, then our website programming work is complete. But almost all websites need dynamic content, which means a web page that is generated at runtime by executing code.

The easiest way to do that is to use a feature of ASP.NET Core named **Razor Pages**.

Exploring ASP.NET Core Razor Pages

ASP.NET Core Razor Pages allow a developer to easily mix C# code statements with HTML markup to make the generated web page dynamic. That is why they use the `.cshtml` file extension.

By convention, ASP.NET Core looks for Razor Pages in a folder named `Pages`.

Enabling Razor Pages

You will now copy and change the static HTML page into a dynamic Razor Page, and then add and enable the Razor Pages service:

1. In the `Northwind.Web` project folder, create a folder named `Pages`.
2. Copy the `index.html` file into the `Pages` folder.

3. For the file in the `Pages` folder, rename the file extension from `.html` to `.cshtml`.
4. Remove the `<h2>` element that says that this is a static HTML page.
5. In `Startup.cs`, in the `ConfigureServices` method, add a statement to add ASP.NET Core Razor Pages and its related services, such as model binding, authorization, anti-forgery, views, and tag helpers, to the builder, as shown in the following code:

```
services.AddRazorPages();
```

6. In `Startup.cs`, in the `Configure` method, in the configuration to use endpoints, add a statement to call the `MapRazorPages` method, as shown highlighted in the following code:

```
app.UseEndpoints(endpoints =>
{
  endpoints.MapRazorPages();

  endpoints.MapGet("/hello",  () => "Hello World!");
});
```

Adding code to a Razor Page

In the HTML markup of a web page, Razor syntax is indicated by the `@` symbol. Razor Pages can be described as follows:

- They require the `@page` directive at the top of the file.
- They can optionally have an `@functions` section that defines any of the following:
  - Properties for storing data values, like in a class definition. An instance of that class is automatically instantiated named `Model` that can have its properties set in special methods and you can get the property values in the HTML.
  - Methods named `OnGet`, `OnPost`, `OnDelete`, and so on that execute when HTTP requests are made, such as `GET`, `POST`, and `DELETE`.

Let's now convert the static HTML page into a Razor Page:

1. In the `Pages` folder, open `index.cshtml`.
2. Add the `@page` statement to the top of the file.
3. After the `@page` statement, add an `@functions` statement block.
4. Define a property to store the name of the current day as a `string` value.
5. Define a method to set `DayName` that executes when an HTTP `GET` request is made for the page, as shown in the following code:

```
@page

@functions
{
```

```
    public string? DayName { get; set; }

    public void OnGet()
    {
      Model.DayName = DateTime.Now.ToString("dddd");
    }
  }
```

6. Output the day name inside the second HTML paragraph, as shown highlighted in the following markup:

```
<p>It's @Model.DayName! Our customers include restaurants, hotels, and
cruise lines.</p>
```

7. Start the website.
8. In Chrome, enter `https://localhost:5001/` and note the current day name is output on the page, as shown in *Figure 14.10*:

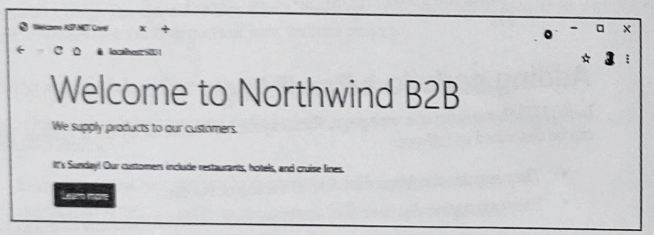

Figure 14.10: Welcome to Northwind page showing the current day

9. In Chrome, enter `https://localhost:5001/index.html`, which exactly matches the static filename, and note that it returns the static HTML page as before.
10. In Chrome, enter `https://localhost:5001/hello`, which exactly matches the endpoint route that returns plain text, and note that it returns `Hello World!` as before.
11. Close Chrome and shut down the web server.

Using shared layouts with Razor Pages

Most websites have more than one page. If every page had to contain all of the boilerplate markup that is currently in `index.cshtml`, that would become a pain to manage. So, ASP.NET Core has a feature named **layouts**.

To use layouts, we must create a Razor file to define the default layout for all Razor Pages (and all MVC views) and store it in a `Shared` folder so that it can be easily found by convention. The name of this file can be anything, because we will specify it, but `_Layout.cshtml` is good practice.

We must also create a specially named file to set the default layout file for all Razor Pages (and all MVC views). This file must be named `_ViewStart.cshtml`.

Let's see layouts in action:

1. In the `Pages` folder, add a file named `_ViewStart.cshtml`. (The Visual Studio item template is named **Razor View Start**.)
2. Modify its content, as shown in the following markup:

```
@{
  Layout = "_Layout";
}
```

3. In the `Pages` folder, create a folder named `Shared`.
4. In the `Shared` folder, create a file named `_Layout.cshtml`. (The Visual Studio item template is named **Razor Layout**.)
5. Modify the content of `_Layout.cshtml` (it is similar to `index.cshtml` so you can copy and paste the HTML markup from there), as shown in the following markup:

```
<!doctype html>
<html lang="en">

<head>
  <!-- Required meta tags -->
  <meta charset="utf-8" />
  <meta name="viewport" content=
    "width=device-width, initial-scale=1, shrink-to-fit=no" />

  <!-- Bootstrap CSS -->
  <link href=
"https://cdn.jsdelivr.net/npm/bootstrap@5.1.0/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-KyZXEAg3QhqLMpG8r+8fhAXLRk2vvoC2f3B09zV
Xn8CA5QIVfZOJ3BCsw2P0p/We" crossorigin="anonymous">

  <title>@ViewData["Title"]</title>
</head>

<body>
  <div class="container">
    @RenderBody()
    <hr />
    <footer>
      <p>Copyright &copy; 2021 - @ViewData["Title"]</p>
    </footer>
  </div>

  <!-- JavaScript to enable features like carousel -->
```

```
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.0/
dist/js/bootstrap.bundle.min.js" integrity="sha384-
U1DAWAznBHeqEIlVSCgzq+c9gqGAJn5c/t99JyeKa9xxaYpSvHU5awsuZVVFIhvj"
crossorigin="anonymous"></script>

    @RenderSection("Scripts", required: false)

</body>
</html>
```

While reviewing the preceding markup, note the following:

- `<title>` is set dynamically using server-side code from a dictionary named `ViewData`. This is a simple way to pass data between different parts of an ASP.NET Core website. In this case, the data will be set in a Razor Page class file and then output in the shared layout.
- `@RenderBody()` marks the insertion point for the view being requested.
- A horizontal rule and footer will appear at the bottom of each page.
- At the bottom of the layout is a script to implement some cool features of Bootstrap that we can use later, such as a carousel of images.
- After the `<script>` elements for Bootstrap, we have defined a section named `Scripts` so that a Razor Page can optionally inject additional scripts that it needs.

6. Modify `index.cshtml` to remove all HTML markup except `<div class="jumbotron">` and its contents, and leave the C# code in the `@functions` block that you added earlier.
7. Add a statement to the `OnGet` method to store a page title in the `ViewData` dictionary, and modify the button to navigate to a suppliers page (which we will create in the next section), as shown highlighted in the following markup:

```
@page

@functions
{
  public string? DayName { get; set; }

  public void OnGet()
  {
    ViewData["Title"] = "Northwind B2B";

    Model.DayName = DateTime.Now.ToString("dddd");
  }
}
<div class="jumbotron">
  <h1 class="display-3">Welcome to Northwind B2B</h1>
  <p class="lead">We supply products to our customers.</p>
```



```
  <hr />
  <p>It's @Model.DayName! Our customers include restaurants, hotels, and
cruise lines.</p>
  <p>
    <a class="btn btn-primary" href="suppliers">
      Learn more about our suppliers</a>
  </p>
</div>
```

8. Start the website, visit it with Chrome, and note that it has similar behavior as before, although clicking the button for suppliers will give a `404 Not Found` error because we have not created that page yet.

Using code-behind files with Razor Pages

Sometimes, it is better to separate the HTML markup from the data and executable code, so Razor Pages allows you to do this by putting the C# code in **code-behind** class files. They have the same name as the `.cshtml` file but end with `.cshtml.cs`.

You will now create a page that shows a list of suppliers. In this example, we are focusing on learning about code-behind files. In the next topic, we will load the list of suppliers from a database, but for now, we will simulate that with a hardcoded array of `string` values:

1. In the `Pages` folder, add two new files named `Suppliers.cshtml` and `Suppliers.cshtml.cs`. (The Visual Studio item template is named **Razor Page - Empty** and it creates both files.)
2. Add statements to the code-behind file named `Suppliers.cshtml.cs`, as shown in the following code:

```
using Microsoft.AspNetCore.Mvc.RazorPages; // PageModel

namespace Northwind.Web.Pages;

public class SuppliersModel : PageModel
{
  public IEnumerable<string>? Suppliers { get; set; }

  public void OnGet()
  {
    ViewData["Title"] = "Northwind B2B - Suppliers";

    Suppliers = new[]
    {
      "Alpha Co", "Beta Limited", "Gamma Corp"
    };
  }
}
```

While reviewing the preceding markup, note the following:

- `SuppliersModel` inherits from `PageModel`, so it has members such as the `ViewData` dictionary for sharing data. You can right-click on `PageModel` and select **Go To Definition** to see that it has lots more useful features, such as the entire `HttpContext` of the current request.
- `SuppliersModel` defines a property for storing a collection of `string` values named `Suppliers`.
- When an HTTP `GET` request is made for this Razor Page, the `Suppliers` property is populated with some example supplier names from an array of `string` values. Later, we will populate this from the Northwind database.

3. Modify the contents of `Suppliers.cshtml`, as shown in the following markup:

```
@page
@model Northwind.Web.Pages.SuppliersModel
<div class="row">
  <h1 class="display-2">Suppliers</h1>
  <table class="table">
    <thead class="thead-inverse">
      <tr><th>Company Name</th></tr>
    </thead>
    <tbody>
    @if (Model.Suppliers is not null)
    {
      @foreach(string name in Model.Suppliers)
      {
        <tr><td>@name</td></tr>
      }
    }
    </tbody>
  </table>
</div>
```

While reviewing the preceding markup, note the following:

- The model type for this Razor Page is set to `SuppliersModel`.
- The page outputs an HTML table with Bootstrap styles.
- The data rows in the table are generated by looping through the `Suppliers` property of `Model` if it is not `null`.

4. Start the website and visit it using Chrome.
5. Click on the button to learn more about suppliers, and note the table of suppliers, as shown in *Figure 14.11*:

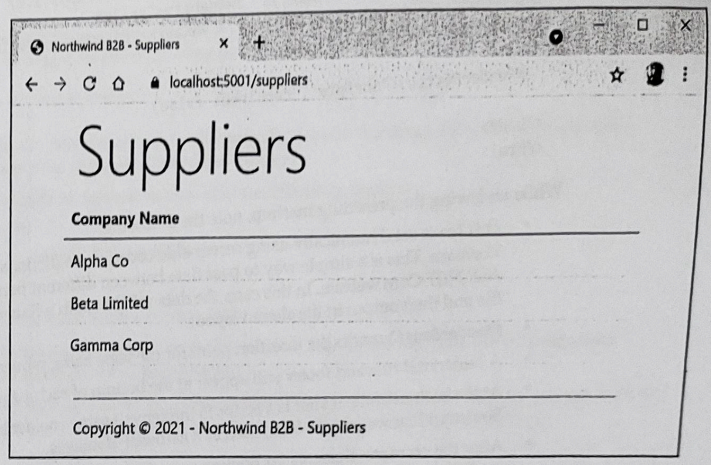

Figure 14.11: The table of suppliers loaded from an array of strings

Using Entity Framework Core with ASP.NET Core

Entity Framework Core is a natural way to get real data into a website. In *Chapter 13, Introducing Practical Applications of C# and .NET*, you created two pairs of class libraries: one for the entity models and one for the Northwind database context, for either SQL Server or SQLite or both. You will now use them in your website project.

Configure Entity Framework Core as a service

Functionality such as Entity Framework Core database contexts that are needed by ASP.NET Core must be registered as a service during website startup. The code in the GitHub repository solution and below uses SQLite, but you can easily use SQL Server if you prefer.

Let's see how:

1. In the `Northwind.Web` project, add a project reference to the `Northwind.Common.DataContext` project for either SQLite or SQL Server, as shown in the following markup:

```
<!-- change Sqlite to SqlServer if you prefer -->
<ItemGroup>
  <ProjectReference Include="..\Northwind.Common.DataContext.Sqlite\
Northwind.Common.DataContext.Sqlite.csproj" />
</ItemGroup>
```

The project reference must go all on one line with no line break.

2. Build the `Northwind.Web` project.
3. In `Startup.cs`, import namespaces to work with your entity model types, as shown in the following code:

```
using Packt.Shared; // AddNorthwindContext extension method
```

4. Add a statement to the `ConfigureServices` method to register the `Northwind` database context class, as shown in the following code:

```
services.AddNorthwindContext();
```

5. In the `Northwind.Web` project, in the `Pages` folder, open `Suppliers.cshtml.cs`, and import the namespace for our database context, as shown in the following code:

```
using Packt.Shared; // NorthwindContext
```

6. In the `SuppliersModel` class, add a private field to store the `Northwind` database context and a constructor to set it, as shown in the following code:

```
private NorthwindContext db;

public SuppliersModel(NorthwindContext injectedContext)
{
  db = injectedContext;
}
```

7. Change the `Suppliers` property to contain `Supplier` objects instead of `string` values.
8. In the `OnGet` method, modify the statements to set the `Suppliers` property from the `Suppliers` property of the database context, sorted by country and then company name, as shown highlighted in the following code:

```
public void OnGet()
{
  ViewData["Title"] = "Northwind B2B - Suppliers";

  Suppliers = db.Suppliers
    .OrderBy(c => c.Country).ThenBy(c => c.CompanyName);
}
```

9. Modify the contents of `Suppliers.cshtml` to import the `Packt.Shared` namespace and render multiple columns for each supplier, as shown highlighted in the following markup:

```
@page
@using Packt.Shared
@model Northwind.Web.Pages.SuppliersModel
<div class="row">
  <h1 class="display-2">Suppliers</h1>
  <table class="table">
    <thead class="thead-inverse">
      <tr>
        <th>Company Name</th>
        <th>Country</th>
        <th>Phone</th>
      </tr>
    </thead>
    <tbody>
    @if (Model.Suppliers is not null)
    {
      @foreach(Supplier s in Model.Suppliers)
      {
        <tr>
          <td>@s.CompanyName</td>
          <td>@s.Country</td>
          <td>@s.Phone</td>
        </tr>
      }
    }
    </tbody>
  </table>
</div>
```

10. Start the website.
11. In Chrome, enter `https://localhost:5001/`.
12. Click **Learn more about our suppliers** and note that the supplier table now loads from the database, as shown in *Figure 14.12*:

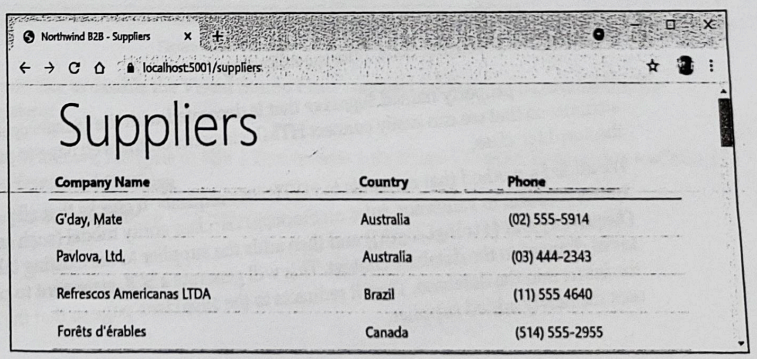

Figure 14.12: The suppliers table loaded from the Northwind database

Manipulating data using Razor Pages

You will now add functionality to insert a new supplier.

Enabling a model to insert entities

First, you will modify the supplier model so that it responds to HTTP `POST` requests when a visitor submits a form to insert a new supplier:

1. In the `Northwind.Web` project, in the `Pages` folder, open `Suppliers.cshtml.cs` and import the following namespace:

```
using Microsoft.AspNetCore.Mvc; // [BindProperty], IActionResult
```

2. In the `SuppliersModel` class, add a property to store a single supplier and a method named `OnPost` that adds the supplier to the `Suppliers` table in the Northwind database if its model is valid, as shown in the following code:

```
[BindProperty]
public Supplier? Supplier { get; set; }

public IActionResult OnPost()
{
  if ((Supplier is not null) && ModelState.IsValid)
  {
    db.Suppliers.Add(Supplier);
    db.SaveChanges();
    return RedirectToPage("/suppliers");
  }
  else
  {
    return Page(); // return to original page
  }
}
```

While reviewing the preceding code, note the following:

- We added a property named `Supplier` that is decorated with the `[BindProperty]` attribute so that we can easily connect HTML elements on the web page to properties in the `Supplier` class.
- We added a method that responds to HTTP `POST` requests. It checks that all property values conform to validation rules on the `Supplier` class entity model (such as `[Required]` and `[StringLength]`) and then adds the supplier to the existing table and saves changes to the database context. This will generate a SQL statement to perform the insert into the database. Then it redirects to the `Suppliers` page so that the visitor sees the newly added supplier.

Defining a form to insert a new supplier

Next, you will modify the Razor Page to define a form that a visitor can fill in and submit to insert a new supplier:

1. In `Suppliers.cshtml`, add tag helpers after the `@model` declaration so that we can use tag helpers such as `asp-for` on this Razor Page, as shown in the following markup:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

2. At the bottom of the file, add a form to insert a new supplier, and use the `asp-for` tag helper to bind the `CompanyName`, `Country`, and `Phone` properties of the `Supplier` class to the input box, as shown in the following markup:

```
<div class="row">
  <p>Enter details for a new supplier:</p>
  <form method="POST">
    <div><input asp-for="Supplier.CompanyName"
                placeholder="Company Name" /></div>
    <div><input asp-for="Supplier.Country"
                placeholder="Country" /></div>
    <div><input asp-for="Supplier.Phone"
                placeholder="Phone" /></div>
    <input type="submit" />
  </form>
</div>
```

While reviewing the preceding markup, note the following:

- The `<form>` element with a `POST` method is normal HTML, so an `<input type="submit" />` element inside it will make an HTTP `POST` request back to the current page with values of any other elements inside that form.
- An `<input>` element with a tag helper named `asp-for` enables data binding to the model behind the Razor Page.

3. Start the website.
4. Click **Learn more about our suppliers**, scroll down to the bottom of the page, enter `Bob's Burgers`, `USA`, and `(603) 555-4567`, and click **Submit**.
5. Note that you see a refreshed suppliers table with the new supplier added.
6. Close Chrome and shut down the web server.

Injecting a dependency service into a Razor Page

If you have a `.cshtml` Razor Page that does not have a code-behind file, then you can inject a dependency service using the `@inject` directive instead of constructor parameter injection, and then directly reference the injected database context using Razor syntax in the middle of the markup.

Let's create a simple example:

1. In the `Pages` folder, add a new file named `Orders.cshtml`. (The Visual Studio item template is named **Razor Page - Empty** and it creates two files. Delete the `.cs` file.)
2. In `Orders.cshtml`, write code to output the number of orders in the Northwind database, as shown in the following markup:

```
@page
@using Packt.Shared
@inject NorthwindContext db
@{
  string title = "Orders";
  ViewData["Title"] = $"Northwind B2B - {title}";
}
<div class="row">
  <h1 class="display-2">@title</h1>
  <p>
    There are @db.Orders.Count() orders in the Northwind database.
  </p>
</div>
```

3. Start the website.
4. Navigate to `/orders` and note that you see that there are 830 orders in the Northwind database.
5. Close Chrome and shut down the web server.

Using Razor class libraries

Everything related to a Razor Page can be compiled into a class library for easier reuse in multiple projects. With ASP.NET Core 3.0 and later, this can include static files such as HTML, CSS, JavaScript libraries, and media assets such as image files. A website can either use the Razor Page's view as defined in the class library or override it.

Creating a Razor class library

Let's create a new Razor class library:

Use your preferred code editor to add a new project, as defined in the following list:

1. Project template: **Razor Class Library** / `razorclasslib`
2. Checkbox/switch: **Support pages and views** / `-s`
3. Workspace/solution file and folder: `PracticalApps`
4. Project file and folder: `Northwind.Razor.Employees`

`-s` is short for the `--support-pages-and-views` switch that enables the class library to use Razor Pages and `.cshtml` file views.

Disabling compact folders for Visual Studio Code

Before we implement our Razor class library, I want to explain a Visual Studio Code feature that confused some readers of a previous edition because the feature was added after publishing.

The compact folders feature means that nested folders such as `/Areas/MyFeature/Pages/` are shown in a compact form if the intermediate folders in the hierarchy do not contain files, as shown in *Figure 14.13*:

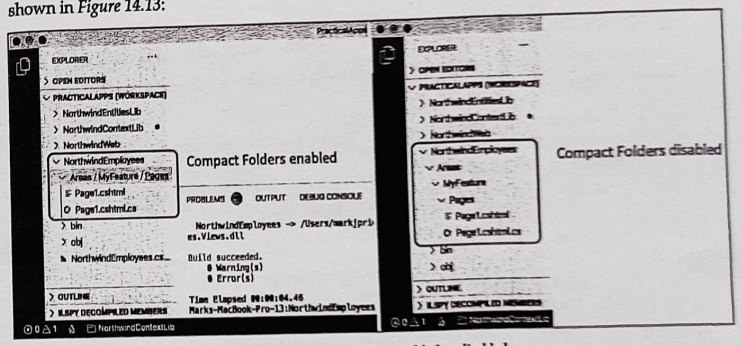

Figure 14.13: Compact folders enabled or disabled

If you would like to disable the Visual Studio Code compact folders feature, complete the following steps:

1. On Windows, navigate to **File | Preferences | Settings**. On macOS, navigate to **Code | Preferences | Settings**.
2. In the **Search settings** box, enter `compact`.

3. Clear the **Explorer: Compact Folders** check box, as shown in *Figure 14.14*:

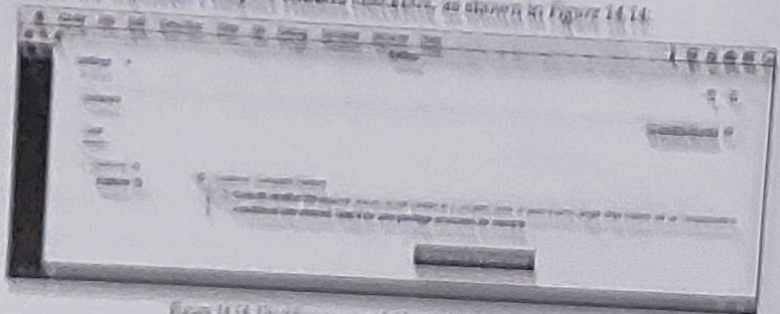

Figure 14.14: Disabling compact folders for Visual Studio Code

4. Close the **Settings** tab.

Implementing the employees feature using EF Core

Now we can add a reference to our entity models to get the employees to show in the Razor class library:

1. In the Northwind.Razor.Employees project, add a project reference to the Northwind.Common.DataContext project for either SQLite or SQL Server and note the SDK is Microsoft.NET.Sdk.Razor, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
[illegible]
```


> The project reference must go all on one line with no line break. Also, do not mix our SQLite and SQL Server projects or you will see compiler errors. If you used SQL Server in the Northwind.Web project, then you must use SQL Server in the Northwind.Razor.Employees project as well.

2. Build the Northwind.Razor.Employees project.
3. In the Areas folder, right-click the MyFeature folder, select **Rename**, enter the new name PacktFeatures, and press *Enter*.
4. In the PacktFeatures folder, in the Pages subfolder, add a new file named _ViewStart.cshtml. (The Visual Studio item template is named **Razor View Start**. Or just copy it from the Northwind.Web project.)
5. Modify its content to inform this class library that any Razor Pages should look for a layout with the same name as used in the Northwind.Web project, as shown in the following markup:

```
@{
  Layout = "_Layout";
}
```

> We do not need to create the _Layout.cshtml file in this project. It will use the one in its host project, for example, the one in the Northwind.Web project.

6. In the Pages subfolder, rename Page1.cshtml to Employees.cshtml, and rename Page1.cshtml.cs to Employees.cshtml.cs.
7. Modify Employees.cshtml.cs to define a page model with an array of Employee entity instances loaded from the Northwind database, as shown in the following code:

```
using Microsoft.AspNetCore.Mvc.RazorPages; // PageModel
using Packt.Shared; // Employee, NorthwindContext

namespace PacktFeatures.Pages;

public class EmployeesPageModel : PageModel
{
  [illegible]

  public EmployeesPageModel([illegible])
  {
    [illegible]
  }
```