

# Working with Databases Using Entity Framework Core

This section 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 section 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

# 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 data store such as Microsoft Azure Cosmos DB, Redis, MongoDB, and Apache Cassandra.

This section will focus on RDBMSes such as SQL Server and SQLite. If you wish to learn more about NoSQL databases, such as Cosmos DB and MongoDB, and how to use them with EF Core, then I recommend the following links, which will go over them in detail:

- **Welcome to Azure Cosmos DB:** <https://docs.microsoft.com/en-us/azure/cosmos-db/introduction>
- **Use NoSQL databases as a persistence infrastructure:** <https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/nosql-database-persistence-infrastructure>
- **Document Database Providers for Entity Framework Core:** <https://github.com/BlueshiftSoftware/EntityFrameworkCore>

## 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 old 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, including .NET 5. 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.

**More Information:** You can read more about Entity Framework 6.3 and its .NET Core 3.0 and later support at the following link: <https://devblogs.microsoft.com/dotnet/announcing-ef-core-3-0-and-ef-6-3-general-availability/>

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. This module is about modern cross-platform development so, in the rest of this section, 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 section.

## 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. For example, as well as traditional RDBMSes, EF Core also supports modern cloud-based, nonrelational, schema-less data stores, such as Microsoft Azure Cosmos DB and MongoDB, sometimes with third-party providers.

EF Core 5.0 runs on platforms that support .NET Standard 2.1, meaning .NET Core 3.0 and 3.1, as well as .NET 5. It will not run on .NET Standard 2.0 platforms like .NET Framework 4.8.

**More Information:** You can read more about the EF Core team's plans at the following link: <https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-5.0/plan>

EF Core 5.0 has so many improvements that this section cannot cover them all. I will focus on the fundamentals that all .NET developers should know and some of the cooler new features.

**More Information:** You can read the complete list of new features in EF Core 5 at the following link: <https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-5.0/whatsnew>

## 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 module:

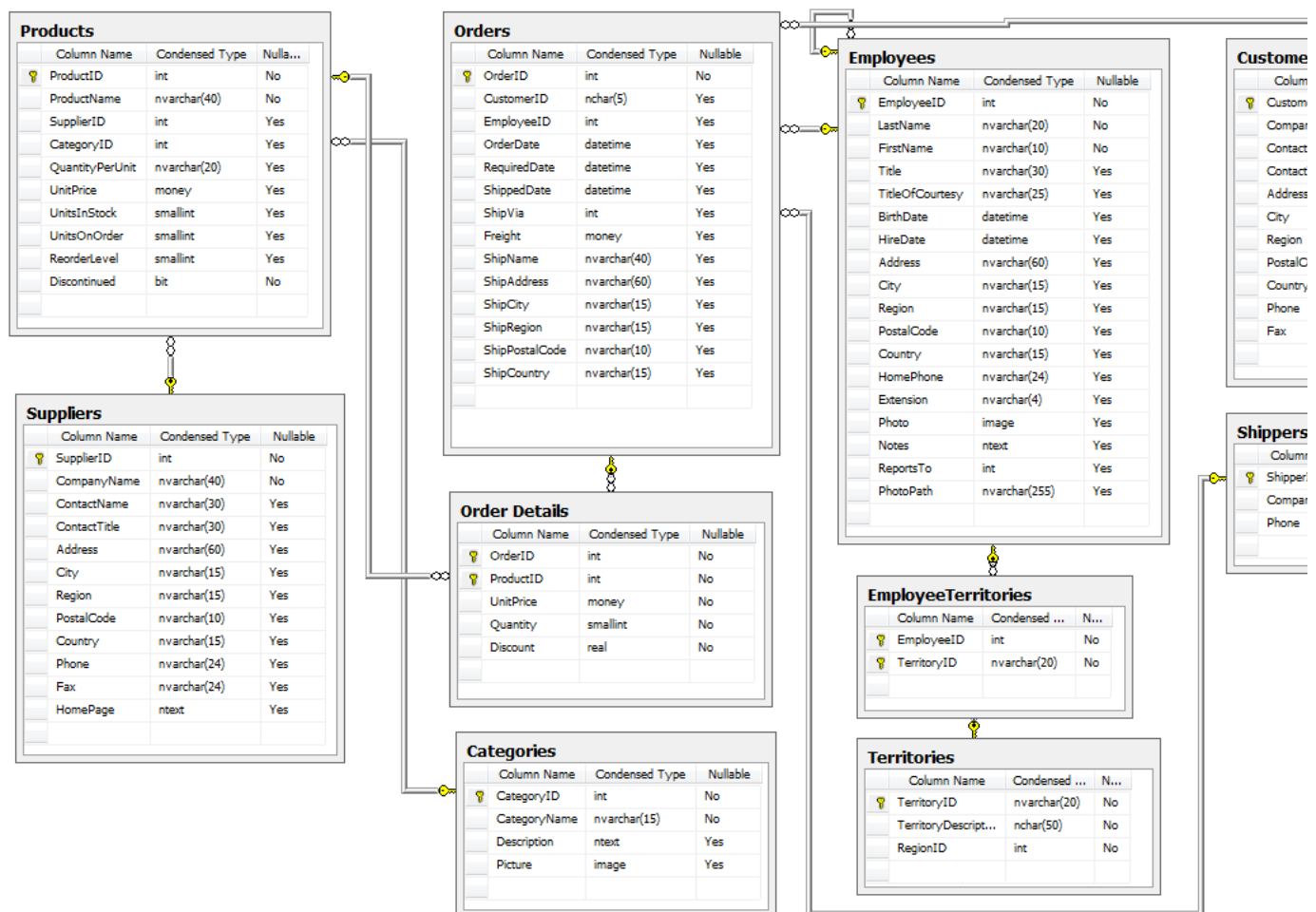


Figure 4.1: The Northwind database tables and relationships

You will write code to work with the `Categories` and `Products` tables later in this section and other tables in later sections. 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.

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.

## 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

SQLite can be downloaded and installed for other OSes. On Windows, we also need to add the folder for SQLite to the system path so it will be found when we enter commands in Command Prompt:

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-x86-3330000.zip`. Note the file might have a higher version number after this module is written.
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**.
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**.

## Creating the Northwind sample database for SQLite

Now we can create the Northwind sample database using a SQL script:

1. Create a folder named `Section04` with a subfolder named `WorkingWithEFCore`.
2. Download and copy the SQL script to create the Northwind database for SQLite to the `WorkingWithEFCore` folder.

3. In Visual Studio Code, open the `WorkingWithEFCore` folder.
4. Navigate to **TERMINAL** and execute the script using SQLite to create the `Northwind.db` database, as shown in the following command:

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

5. Be patient because this command might take a while to create the database structure, as shown in the following output:

```
-- Loading resources from Northwind.sql
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
sqlite>
```

6. Press `Ctrl + C` on Windows or `Ctrl + D` on macOS to exit SQLite command mode.

**More Information:** You can read about the SQL statements supported by SQLite at the following link: <https://sqlite.org/lang.html>

## 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, <http://sqlitestudio.pl>, and download and unpack the application.
2. Launch **SQLiteStudio**.
3. On the **Database** menu, choose **Add a database**.
4. In the **Database** dialog, click on the folder button to browse for an existing database file on the local computer, and 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 tables that were created by the script.
6. Right-click on the `Products` table and choose **Edit the table**.

In the table editor window, you will see the structure of the `Products` table, including column names, data types, keys, and constraints, as shown in the following screenshot:

	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Default value
1	ProductID	INTEGER	✓						NULL
2	ProductName	nvarchar (40)					☹		NULL
3	SupplierID	"int"		✓					NULL
4	CategoryID	"int"		✓					NULL
5	QuantityPerUnit	nvarchar (20)							NULL
6	UnitPrice	"money"				☹			0
7	UnitsInStock	"smallint"				☹			0
8	UnitsOnOrder	"smallint"				☹			0
9	ReorderLevel	"smallint"				☹			0
10	Discontinued	"bit"					☹		0

Figure 4.2: The table editor in SQLiteStudio showing the structure of the `Products` table

7. In the table editor window, click the **Data** tab. You will see 77 products, as shown in the following screenshot:

	ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder
1	1	Chai	1	1	10 boxes x 20 bags	18	39	0
2	2	Chang	1	1	24 - 12 oz bottles	19	17	40
3	3	Aniseed Syrup	1	2	12 - 550 ml bottles	10	13	70

Figure 4.3: The Data tab showing the rows in the `Products` table

# 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 is 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

**More Information:** You can see the full list of EF Core database providers at the following link: <https://docs.microsoft.com/en-us/ef/core/providers/>

Devart is a third party that offers EF Core database providers for a wide range of data stores.

**More Information:** Read more about Devart database providers at the following link: <https://www.devart.com/dotconnect/entityframework.html>

## 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 create and apply migrations from an older model to a newer model and generate code for a model from an existing database.

Since .NET Core 3.0, 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 the tool, you should uninstall any existing version:

1. In **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 the tool has been installed, 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 5.0.0
```

## Connecting to the database

To connect to SQLite, we just need to know the database filename. We specify this information in a connection string:

1. In Visual Studio Code, make sure that you have opened the `WorkingWithEFCore` folder, and then in **TERMINAL**, enter the `dotnet new console` command.
2. Edit `WorkingWithEFCore.csproj` to add a package reference to the EF Core data provider for SQLite, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Sqlite"
      Version="5.0.0" />
    </ItemGroup>
</Project>
```

3. In **TERMINAL**, build the project to restore packages, as shown in the following command:

```
dotnet build
```

**More Information:** You can check the most recent version at the following link:  
<https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Sqlite/>

# 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.

## EF Core conventions

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 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.
- A property that is named `ID`, or if the class is named `Product`, then the property can be named `ProductID`. That property is then assumed to be a primary key. If this property is an integer type or the `Guid` type, then it is also assumed to be `IDENTITY` (a column type that automatically assigns a value when inserting).

**More Information:** There are many other conventions, and you can even define your own, but that is beyond the scope of this module. You can read about them at the following link: <https://docs.microsoft.com/en-us/ef/core/modeling/>

## EF Core annotation attributes

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.

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:

```
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; }
```

## EF Core Fluent API

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, let's look at the following two attributes in a `Product` class, as shown in the following code:

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

The attributes could be removed from the class to keep it simpler, and replaced with an equivalent Fluent API statement in the `OnModelCreating` method of a database context class, as shown in the following code:

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

## Understanding data seeding

You can use the Fluent API to provide initial data to populate a database. EF Core automatically works out what insert, update, or delete operations must be executed. 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.

**More Information:** You can read more about data seeding at the following link: <https://docs.microsoft.com/en-us/ef/core/modeling/data-seeding>

## Building an EF Core model

Now that you've learned about model conventions, let's build a model to represent two tables and the `Northwind` database. To make the classes more reusable, we will define them in the `Packt.Shared` namespace. These three classes will refer to each other, so to avoid compiler errors, we will create the three classes without any members first:

1. Add three class files to the `WorkingWithEFCore` project named `Northwind.cs`, `Category.cs`, and `Product.cs`.
2. In the file named `Northwind.cs`, define a class named `Northwind`, as shown in the following code:

```
namespace Packt.Shared
{
    public class Northwind
    {
    }
}
```

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

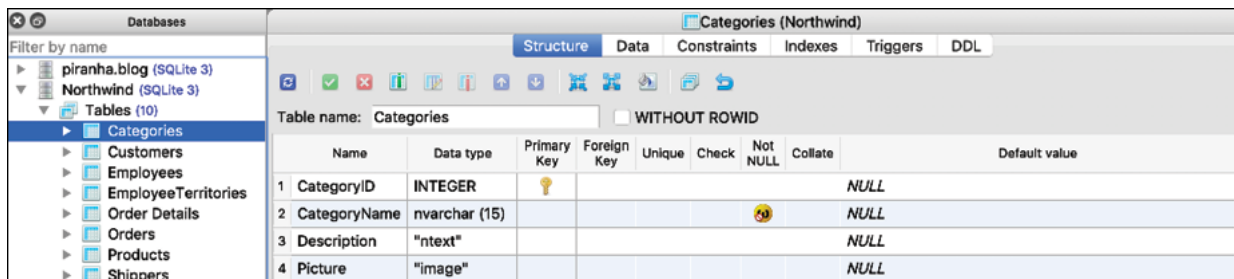
```
namespace Packt.Shared
{
    public class Category
    {
    }
}
```

4. In the file named `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

`Category` will be used to represent a row in the `Categories` table, which has four columns, as shown in the following screenshot:



Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Default value
1 CategoryID	INTEGER	Yes						NULL
2 CategoryName	nvarchar (15)					Yes		NULL
3 Description	"ntext"							NULL
4 Picture	"image"							NULL

Figure 4.4: The Categories table structure

We will use conventions to define three of the four properties (we will not map the `Picture` column), the primary key, and 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 section, we will use the Fluent API to define that `CategoryName` cannot be `null` and is limited to a maximum of 15 characters:



## 1. Modify Category.cs, as shown in the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;
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
            this.Products = new HashSet<Product>();
        }
    }
}
```

Product 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. In this scenario, the rows already have data values and I have decided that I do not need to read those values in the console 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 Product.cs, as shown in the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace Packt.Shared
{
    public class Product
    {
        public int ProductID { get; set; }
        [Required]
        [StringLength(40)]
        public string ProductName { get; set; }
        [Column("UnitPrice", TypeName = "money")]
        public decimal? Cost { get; set; } // property name != field 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; }
    }
}
```

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. Lazy loading is not available in EF Core 2.0 or earlier.

## 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.

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 properties use generics to specify a class that represents a row in the table, with properties that represent its columns.

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

Likewise, your 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:

## 1. Modify Northwind.cs, as shown in the following code:

```
using Microsoft.EntityFrameworkCore;
namespace Packt.Shared
{
    // this manages the connection to the database
    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)
        {
            string path = System.IO.Path.Combine(
                System.Environment.CurrentDirectory, "Northwind.db");
            optionsBuilder.UseSqlite($"Filename={path}");
        }
    }
}
```

```
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);
    // 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 for sorting and other operations. We can fix this by telling SQLite that `decimal` values can be converted to `double` values. 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.

## 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 a best approximation. Do not be afraid to overrule the tool.

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

1. In **TERMINAL**, add the EF Core design package to the `WorkingWithEFCore` project, as shown in the following command:

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

2. 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 AutoG
```

Note the following:

- The command to perform: `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 Packt.Shared.AutoGen`
  - To use data annotations as well as Fluent API: `--data-annotations`
  - To rename the context from `[database_name]Context`: `--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 con
/?LinkId=723263.
Could not scaffold the foreign key '0'. The referenced table could not be found. This most likely occurred because the referenced table was excl
```

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;
#nullable disable
namespace Packt.Shared.AutoGen
{
    [Index(nameof(CategoryName), Name = "CategoryName")]
    public partial class Category
    {
        public Category()
        {
            Products = new HashSet<Product>();
        }
        [Key]
        [Column("CategoryID")]
        public long CategoryId { get; set; }
        [Required]
        [Column(TypeName = "nvarchar (15)")]
        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:

- The `dotnet-ef` tool currently cannot use the nullable reference types language feature so it explicitly disables nullability.

- It decorates the entity class with the `[Index]` attribute introduced in EF Core 5.0 that indicates properties that match to fields that should have an index. In earlier versions, only the Fluent API was supported for defining indexes.
- The table name is `Categories` but the `dotnet-ef` tool uses the Humanizer third-party library to automatically singularize (or pluralize) the class name to `Category`, which is a more natural name when creating a single entity.

**More Information:** You can learn about the Humanizer library and how you might use it in your own apps at the following link: <http://humanizr.net>

- 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. It is also misnamed since Microsoft naming conventions say that two-letter abbreviations or acronyms should use all uppercase and not title case.
- 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;
#nullable disable
namespace Packt.Shared.AutoGen
{
    public partial class Northwind : DbContext
    {
        public Northwind()
        {
        }
        public Northwind(DbContextOptions<Northwind> options)
            : base(options)
        {
        }
        public virtual DbSet<Category> Categories { get; set; }
        public virtual DbSet<Product> Products { get; set; }
        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
                optionsBuilder.UseSqlite("Filename=Northwind.db");
            }
        }
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Category>(entity =>
            {
                entity.Property(e => e.CategoryId)
                    .ValueGeneratedNever()
                    .HasColumnName("CategoryID");
                entity.Property(e => e.CategoryName)
                    .HasAnnotation("Relational:ColumnType", "nvarchar (15)");
                entity.Property(e => e.Description)
                    .HasAnnotation("Relational:ColumnType", "ntext");
                entity.Property(e => e.Picture)
                    .HasAnnotation("Relational:ColumnType", "image");
            });
            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.
- 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, 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.

**More Information:** You can read more about scaffolding at the following link: <https://docs.microsoft.com/en-us/ef/core/managing-schemas/scaffolding?tabs=dotnet-core-cli>

In the rest of this section, 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 later. For now, just write the code and view the results:

1. Open `Program.cs` and import the following namespaces:

```
using static System.Console;
using Packt.Shared;
using Microsoft.EntityFrameworkCore;
using System.Linq;
```

2. In `Program`, 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 as soon as possible so we will wrap it in a `using` statement.
  - 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 (var db = new Northwind())
    {
        WriteLine("Categories and how many products they have:");
        // a query to get all categories and their related products
        IQueryable<Category> cats = db.Categories
            .Include(c => c.Products);
        foreach (Category c in cats)
        {
            WriteLine($"{c.CategoryName} has {c.Products.Count} products.");
        }
    }
}
```

3. In `Main`, call the `QueryingCategories` method, as shown in the following code:

```
static void Main(string[] args)
{
    QueryingCategories();
}
```

4. Run the application and view the result, as shown in the following output:

```
Categories and how many products they have:
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.
```

## 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. In `Program`, 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 (var db = new Northwind())
    {
        Write("Enter a minimum for units in stock: ");
        string unitsInStock = ReadLine();
        int stock = int.Parse(unitsInStock);
        IQueryable<Category> cats = db.Categories
            .Include(c => c.Products.Where(p => p.Stock >= stock));
        foreach (Category c in cats)
        {

```

```

        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 Main, comment out the `QueryingCategories` method and invoke the `FilteredIncludes` method, as shown in the following code:

```

static void Main(string[] args)
{
    // QueryingCategories();
    FilteredIncludes();
}

```

3. Run the application, 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.

```

**More Information:** You can read more about filtered include at the following link: <https://docs.microsoft.com/en-us/ef/core/querying/related-data/eager#filtered-include>

## Filtering and sorting products

Let's explore a more complex query that filters and sorts data:

1. In Program, 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.
  - Create a query for products that cost more than the price using LINQ.
  - Loop through the results, outputting the ID, name, cost (formatted with US dollars), and the number of units in stock:

```

static void QueryingProducts()
{
    using (var db = new Northwind())
    {
        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> prods = db.Products
            .Where(product => product.Cost > price)
            .OrderByDescending(product => product.Cost);
        foreach (Product item in prods)
        {
            WriteLine(
                $"{0}: {1} costs {2:$#,##0.00} and has {3} in stock.",
                item.ProductID, item.ProductName, item.Cost, item.Stock);
        }
    }
}

```

2. In Main, comment the previous method, and call the method, as shown in the following code:

```
static void Main(string[] args)
{
    // QueryingCategories();
    // FilteredIncludes();
    QueryingProducts();
}
```

- Run the application, 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.
```

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. 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
```

## 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 introduces a quick and easy way to see the SQL generated:

- In the `FilteredIncludes` method, after defining the query, add a statement to output the generated SQL, as shown highlighted in the following code:

```
IQueryable<Category> cats = db.Categories
    .Include(c => c.Products.Where(p => p.Stock >= stock));
WriteLine($"ToQueryString: {cats.ToQueryString()}");
```

- Modify the `Main` method to comment out the call to the `QueryingProducts` method and uncomment the call to the `FilteredIncludes` method.
- Run the application, enter a minimum for units in stock like 99, and view the result, as shown in the following output:

```
Enter a minimum for units in stock: 99
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.

## Logging EF Core

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 us see an example of this in action:

- Add a file to your project named `ConsoleLogger.cs`.
- 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;
using System;
using static System.Console;
namespace Packt.Shared
{
    public class ConsoleLoggerProvider : ILoggerProvider
    {
        public ILogger CreateLogger(string categoryName)
        {
            return new ConsoleLogger();
        }
        // if your logger uses unmanaged resources,
        // you can release the memory 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();
        }
    }
}
```

- At the top of the `Program.cs` file, 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;
```

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

- Run the console application 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/Code/Section04/WorkingWithEFCore/Northwind.db'.
Level: Debug, Event ID: 20001, State: Opened connection to database 'main' on
server '/Users/Code/Section04/WorkingWithEFCore/Northwind.db'.
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
PRAGMA foreign_keys=ON;
```

```
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT "product"."ProductID", "product"."CategoryID", "product"."UnitPrice", "product"."Discontinued",
       "product"."ProductName", "product"."UnitsInStock"
FROM "Products" AS "product"
ORDER BY "product"."UnitPrice" DESC
```

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 `EventId` to output has an `Id` value of 20100.

6. 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();
    }
}
```

7. In `Main`, uncomment the `QueryingCategories` method and comment the `FilteredIncludes` method so that we can monitor the SQL statements that are generated when joining two tables.
8. Run the console application, and note the following SQL statements that were logged, as shown in the following output that has been edited for space:

```
Categories and how many products they have:
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
PRAGMA foreign_keys=ON;
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT "c"."CategoryID", "c"."CategoryName", "c"."Description"
FROM "Categories" AS "c"
ORDER BY "c"."CategoryID"
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT "c.Products"."ProductID", "c.Products"."CategoryID", "c.Products"."UnitPrice", "c.Products"."Discontinued",
       "c.Products"."ProductName", "c.Products"."UnitsInStock"
FROM "Products" AS "c.Products"
INNER JOIN (
    SELECT "c0"."CategoryID"
    FROM "Categories" AS "c0"
) AS "t" ON "c.Products"."CategoryID" = "t"."CategoryID"
ORDER BY "t"."CategoryID"
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> prods = db.Products
    .TagWith("Products filtered by price and sorted.")
    .Where(product => product.Cost > price)
    .OrderByDescending(product => product.Cost);
```

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

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



**More Information:** You can read more about query tags at the following link: <https://docs.microsoft.com/en-us/ef/core/querying/tags>

## Pattern matching with Like

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

1. In `Program`, 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 (var db = new Northwind())
    {
        var loggerFactory = db.GetService<ILoggerFactory>();
        loggerFactory.AddProvider(new ConsoleLoggerProvider());
        Write("Enter part of a product name: ");
        string input = ReadLine();
        IQueryable<Product> prods = db.Products
            .Where(p => EF.Functions.Like(p.ProductName, $"%{input}%"));
        foreach (Product item in prods)
        {
            WriteLine("{0} has {1} units in stock. Discontinued? {2}",
                item.ProductName, item.Stock, item.Discontinued);
        }
    }
}
```

2. In `Main`, comment the existing methods, and call `QueryingWithLike`.
3. Run the console application, enter a partial product name such as `che`, and view the result, as shown in the following output:

```
Enter part of a product name: che
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
PRAGMA foreign_keys=ON;
Level: Debug, Event ID: 20100, State: Executing DbCommand [Parameters=[@__Format_1='?' (Size = 5)], 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
```

## Defining global filters

The `Northwind` products can be discontinued, so it might be useful to ensure that discontinued products are never returned in results, even if the programmer forgets to use `Where` to filter them out:

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

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // example of using Fluent API instead of attributes
    // to limit the length of a category name to under 15
    modelBuilder.Entity<Category>()
        .Property(category => category.CategoryName)
        .IsRequired() // NOT NULL
        .HasMaxLength(15);
    // added to "fix" the lack of decimal support in SQLite
    modelBuilder.Entity<Product>()
        .Property(product => product.Cost)
        .HasConversion<double>();
    // global filter to remove discontinued products
    modelBuilder.Entity<Product>()
        .HasQueryFilter(p => !p.Discontinued);
}
```

2. Run the console application, 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 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: **eager loading**, **lazy loading**, and **explicit loading**. 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 used the `Include` method to use eager loading (also known as **early loading**) for the related products:

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

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

2. In `Main`, comment all methods except `QueryingCategories`.
3. Run the console application 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 using a proxy.

Let us see this in action:

1. Open `WorkingWithEFCore.csproj` and add a package reference, as shown in the following markup:

```
<PackageReference  
  Include="Microsoft.EntityFrameworkCore.Proxies"  
  Version="5.0.0" />
```

2. In **TERMINAL**, build the project to restore packages, as shown in the following command:

```
dotnet build
```

3. Open `Northwind.cs`, import the `Microsoft.EntityFrameworkCore.Proxies` namespace, and call an extension method to use lazy loading proxies before using `SQLite`, as shown highlighted in the following code:

```
optionsBuilder.UseLazyLoadingProxies()  
    .UseSqlite($"Filename={path}");
```

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 would be returned to the output.

4. Run the console app and 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']  
PRAGMA foreign_keys=ON;  
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" = @__p_0)  
Beverages has 11 products.  
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)
Condiments has 11 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. In the `QueryingCategories` method, modify the statements to disable lazy loading and then prompt the user if they want to enable eager loading and explicit loading, as shown in the following code:

```
IQueryable<Category> cats;
// = 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)
{
    cats = db.Categories.Include(c => c.Products);
}
else
{
    cats = db.Categories;
    Write("Enable explicit loading? (Y/N): ");
    explicitloading = (ReadKey().Key == ConsoleKey.Y);
    WriteLine();
}
```

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

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

3. Run the console application; press `n` to disable eager loading, and press `y` to enable explicit loading. 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']
PRAGMA foreign_keys=ON;
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, create a new method named `AddProduct`, as shown in the following code:

```
static bool AddProduct(
    int categoryID, string productName, decimal? price)
{
    using (var db = new Northwind())
    {
        var newProduct = new Product
        {
            CategoryID = categoryID,
            ProductName = productName,
            Cost = price
        };
        // mark product as added in change tracking
        db.Products.Add(newProduct);
        // save tracked change to database
        int affected = db.SaveChanges();
        return (affected == 1);
    }
}
```

2. In Program, 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 (var db = new Northwind())
    {
        WriteLine("{0,-3} {1,-35} {2,8} {3,5} {4}",
            "ID", "Product Name", "Cost", "Stock", "Disc.");
        foreach (var item in db.Products.OrderByDescending(p => p.Cost))
        {
            WriteLine("{0:000} {1,-35} {2,8:$#,##0.00} {3,5} {4}",
                item.ProductID, item.ProductName, item.Cost,
                item.Stock, item.Discontinued);
        }
    }
}
```

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

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

```
static void Main(string[] args)
{
```

```
// QueryingCategories();
// FilteredIncludes();
// QueryingProducts();
// QueryingWithLike();
if (AddProduct(6, "Bob's Burgers", 500M))
{
    WriteLine("Add product successful.");
}
ListProducts();
}
```

4. Run the application, 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 $81.00      40 False
...
```

## Updating entities

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

1. In `Program`, 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 name, decimal amount)
{
    using (var db = new Northwind())
    {
        // get first product whose name starts with name
        Product updateProduct = db.Products.First(
            p => p.ProductName.StartsWith(name));
        updateProduct.Cost += amount;
        int affected = db.SaveChanges();
        return (affected == 1);
    }
}
```

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

```
if (IncreaseProductPrice("Bob", 20M))
{
    // ...
    WriteLine("Update product price successful.");
}
ListProducts();
```

3. Run the console application, 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
...
```

## Deleting entities

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

1. In Program, import `System.Collections.Generic`.
2. 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 name)
{
    using (var db = new Northwind())
    {
        IEnumerable<Product> products = db.Products.Where(
            p => p.ProductName.StartsWith(name));
        db.Products.RemoveRange(products);
        int affected = db.SaveChanges();
        return affected;
    }
}
```

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

3. In Main, comment the whole `if` statement block that calls `IncreaseProductPrice`, and add a call to `DeleteProducts`, as shown highlighted in the following code:

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

4. Run the console application and view the result, as shown in the following output:

```
1 product(s) were deleted.
ID Product Name Cost Stock Disc.
038 Côte de Blaye $263.50 17 False
020 Sir Rodney's Marmalade $81.00 40 False
```

If multiple product names started with Bob, then they are all deleted. As an optional challenge, uncomment 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.

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 web applications and web services.

This allows you to create and dispose of as many `DbContext`-derived objects as you want, knowing your code is still very efficient.

**More Information:** You can read more about pooling database contexts at the following link: <https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-2.0#dbcontext-pooling>

## Transactions

Every time you call the `SaveChanges` method, an **implicit transaction** is started so that if something goes wrong, it would 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. The opposite of durable is volatile:

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. Import the following namespace in `Program.cs` to use the `IDbContextTransaction` interface:

```
using Microsoft.EntityFrameworkCore.Storage;
```

2. In the `DeleteProducts` method, after the instantiation of the `db` variable, add the following highlighted 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 in the following code:

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

```
using (var db = new Northwind())
{
    using (IDbContextTransaction t = db.Database.BeginTransaction())
    {
        WriteLine("Transaction isolation level: {0}",
            t.GetDbTransaction().IsolationLevel);
        var products = db.Products.Where(
            p => p.ProductName.StartsWith(name));
        db.Products.RemoveRange(products);
        int affected = db.SaveChanges();
        t.Commit();
        return affected;
    }
}
```

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

```
Transaction isolation level: Serializable
```

# Explore the EF Core documentation

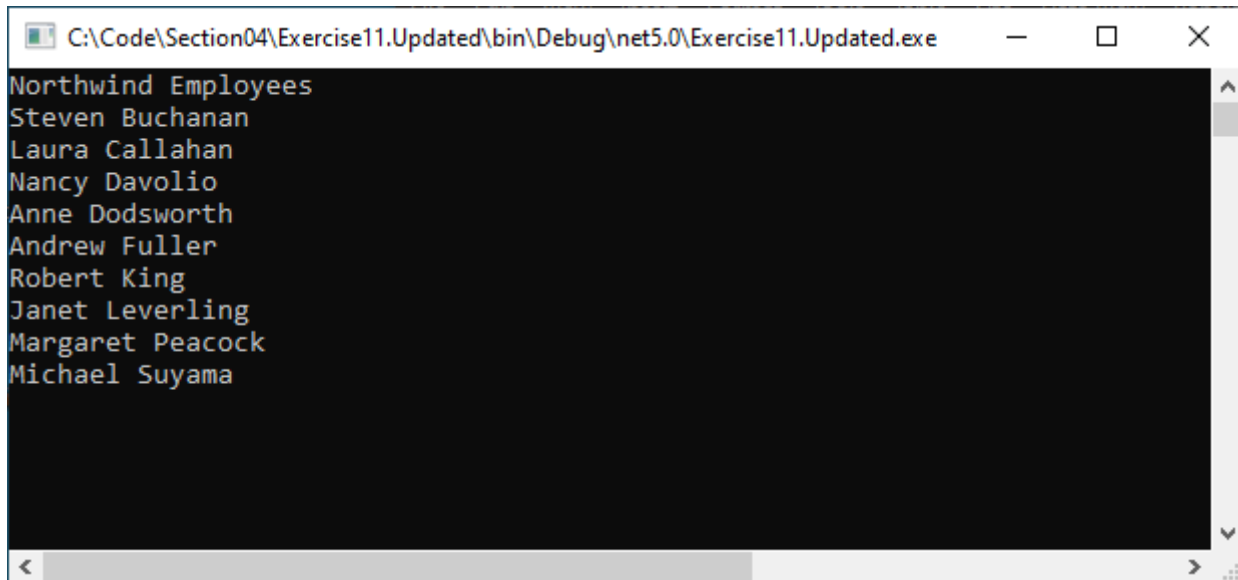
Use the following link to read more about the topics covered in this section:

<https://docs.microsoft.com/en-us/ef/core/>

# Exercises

## Exercise 4.1.

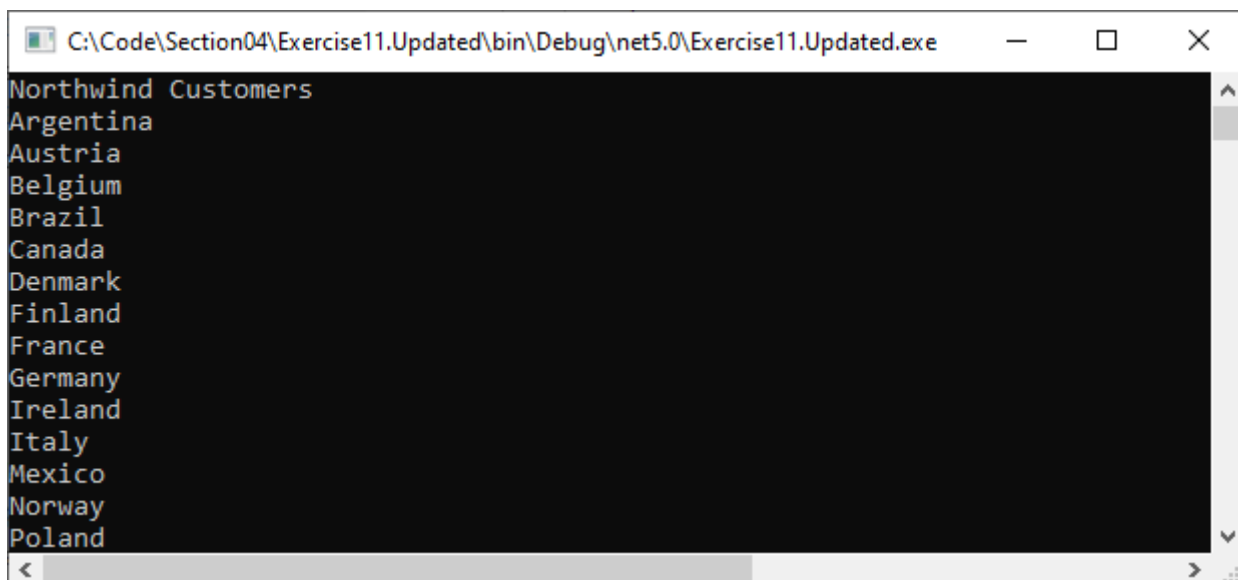
- create a console application that displays the full names of all the Northwind Employees.



```
C:\Code\Section04\Exercise11.Updated\bin\Debug\net5.0\Exercise11.Updated.exe
Northwind Employees
Steven Buchanan
Laura Callahan
Nancy Davolio
Anne Dodsworth
Andrew Fuller
Robert King
Janet Leverling
Margaret Peacock
Michael Suyama
```

## Exercise 4.2.

- create a console application that displays all the countries where Northwind Customers are based



```
C:\Code\Section04\Exercise11.Updated\bin\Debug\net5.0\Exercise11.Updated.exe
Northwind Customers
Argentina
Austria
Belgium
Brazil
Canada
Denmark
Finland
France
Germany
Ireland
Italy
Mexico
Norway
Poland
```

## Exercise 4.3.

- create a console application that lets the end user add a new record to the Northwind Shippers' table

```
C:\Code\Section04\Exercise11.Updated\bin\Debug\net5.0\Exercise11.Updated.exe
Northwind Shippers
Add a new shipper to the database...
What is the company's name? Homer's Haulage
What is the company's phone number? (503) 555-1899
A new shipper has been added to the database
```

- aside, create another console application that displays the entire contents of the Northwind Shippers table

```
C:\Code\Section04\Exercise11.Updated\bin\Debug\net5.0\Exercise11.Updated.exe
Northwind Shippers
Shipper Id | Company Name      | Phone
1          | Speedy Express    | (503) 555-9831
2          | United Package    | (503) 555-3199
3          | Federal Shipping  | (503) 555-9931
4          | Homer's Haulage   | (503) 555-1899
```