|  |
| --- |
| **EASJ Notes** |
| Object-Oriented Pro-gramming with C# |
| Data Persistency |

|  |
| --- |
| By Per Laursen  08-10-2018 |

[Introduction 2](#_Toc526425799)

[File-based Persistency 3](#_Toc526425800)

[Accessing data stored in a relational database 10](#_Toc526425801)

[Creating a local database with Visual Studio 10](#_Toc526425802)

[Accessing data from an UWP application - overview 11](#_Toc526425803)

[The Entity Framework 11](#_Toc526425804)

[RESTful Web Service 12](#_Toc526425805)

[Accessing a database using Entity Framework Core 14](#_Toc526425806)

[Installing the EF Core Power Tools extension 14](#_Toc526425807)

[Using the EF Core Power Tools 16](#_Toc526425808)

[Using the generated classes and properties 24](#_Toc526425809)

[Deploying a database to the Cloud (Azure) 26](#_Toc526425810)

[Exercises 35](#_Toc526425811)

[PERSIST.1 35](#_Toc526425812)

[PERSIST.2 36](#_Toc526425813)

[PERSIST.3 37](#_Toc526425814)

[PERSIST.4 38](#_Toc526425815)

# Introduction

We have until this point not cared much about what happens to our data, once the appli­cation containing the data terminates. Real-life applications will usually contain functionality for saving and loading certain data into the application from so-called **persistent storage**. The term persistent storage just covers all those media for data storage, where data is retained even after the power for the media is shut off. Exam­ples of locations for persistent storage could be your own local hard drive, or simply “the cloud”, where the exact format and location of the data is not known.

Management of persistent storage is a large topic, and more advanced applications will often use some sort of **database** for structured storage of data. For smaller appli­cations producing small amounts of data, simpler media may suffice. Storing data in a text file on your own hard drive is a fairly easy solution to use for such situations.

# File-based Persistency

Even for file-based persistency, there are several possible approaches. We will here present a fairly simple approach, specifically aimed at storing a collection of domain data objects in a text file. Our approach will also rely on using the **JSON** (**JavaScript Object Notation**) text format. JSON can be considered a sort of alternative to XML, since it is just a structured way of storing data on text form. We will not explain JSON further here, but there are plenty of sources about JSON online[[1]](#footnote-1).

The reason for using JSON at all is because it provides a general and convenient way of transform­ing C# objects to a text format, which can then easily be written to a file. Likewise, the text (on JSON format) can be read from the file again, and transformed back into C# objects. There are several third-party class libraries available for JSON conversion, the most popular (at the time of writing) being the **NewtonSoft.JSON** package. This library can be installed as a so-called NuGet package: In Visual Studio, go to **Tools | NuGet Package Manager | Manage NuGet Packages for Solution**, and choose **Browse**. The **NewtonSoft.JSON** package is usually found on the first page; if that is not the case, simply search for it using the search field. Once the package is found, simply click **Install**.

With the package in place, it is possible to create a small C# helper class targeted for our needs. We need to be able to save a collection of domain objects into a file, and load it back into the application. An example of such a class called **FilePersistency** follows below:

**public class FilePersistency<T> where T : class**

**{**

**private const string FileName = "data.json";**

**private CreationCollisionOption \_options;**

**private StorageFolder \_folder;**

**public FilePersistency()**

**{**

**\_options = CreationCollisionOption.OpenIfExists;**

**\_folder = ApplicationData.Current.LocalFolder;**

**}**

**public async Task SaveAsync(List<T> data)**

**{**

**var dataFile = await \_folder.CreateFileAsync(FileName, \_options);**

**string dataJSON = JsonConvert.SerializeObject(data);**

**await FileIO.WriteTextAsync(dataFile, dataJSON);**

**}**

**public async Task<List<T>> LoadAsync()**

**{**

**try**

**{**

**StorageFile dataFile = await \_folder.GetFileAsync(FileName);**

**string dataJSON = await FileIO.ReadTextAsync(dataFile);**

**return (dataJSON != null) ?**

**JsonConvert.DeserializeObject<List<T>>(dataJSON)**

**: new List<T>();**

**}**

**catch (FileNotFoundException)**

**{**

**await SaveAsync(new List<T>());**

**return new List<T>();**

**}**

**}**

**}**

The file operation methods **Create**- and **GetFileAsync** are both part of the .NET class library. They are as such not particularly remarkable, but they both end with the suffix **Async**, indicating that they are both **asynchronous** methods, as discussed in a previ­ous section. Asynchronousmethods offer the possibility to continue execution of the application, even though the method has not returned a result yet! However, we must not let this alternative model for execution confuse us too much right now; the essential point is that these methods offer the functionality we need at this point.

The code above performs a two-step conversion between the file **json.data** and the list of C# objects of type **T** (**T** is a **type parameter**, so it can be any class when the **File­Per­sis­tency** class is actually used). Consider first the **SaveAsync** method. The first line of code creates (or opens, if the file already exists) a file, and returns a variable **data­File** which now refers to the file. The next line converts the incoming data (i.e. a **List** of objects of type **T**) to JSON format. Since this is a text format, the result of the con­ver­­sion is of type **string**. This kind of conversion between an in-memory object and a sequ­en­ce of cha­racters is also known as **serializing** the object, which is why the called method is named **SerializeObject**. Finally, the **JSON** string is written to the file.

JSON string

List of C# objects

File

The **LoadAsync** method is essentially the same deal, just in reversed order. A **string** is read from the file, and then converted (an operation also known as **deserializing**) into a **List** of objects of type **T**

List of C# objects

File

JSON string

The **try-catch** block in the **LoadAsync** method is added to handle the case where the caller attempt­s to load data from a file that has not been created yet, e.g. the first time the appli­ca­tion is executed. In that case, the **LoadAsync** method simply calls the **SaveAsync** method with an empty list, to invoke creation of the file.

How can we then use **FilePersistency** in an MVVM setup? Sticking to the **Car** domain example, we could imagine a fairly simple class e.g. called **CarSource**, containing two methods **LoadAsync** and **SaveAsync**:

**public class CarSource**

**{**

**private FilePersistency<Car> \_fileSource;**

**public CarSource()**

**{**

**\_fileSource = new FilePersistency<Car>();**

**}**

**public async Task<List<Car>> LoadAsync()**

**{**

**await \_fileSource.LoadAsync();**

**}**

**public async Task SaveAsync(List<Car> cars)**

**{**

**await \_filePersistor.SaveAsync(cars);**

**}**

**}**

The **CarSource** class is thus just a very thin type-specific wrapper around the general **FilePersistency<T>** class. Still, it does hide the specific source of the data from the user of the class. The **CarSource** class effectively adds another element to the chain of conversions shown before:

JSON string

List<Car>

CarModel

File

The syntax **async Task** in the method declarations may look a bit strange, but you should not be too intimidated by it. It is just a bit of lingo needed when working with these asynchronous methods, and is thus related to the **await** keyword mentioned before. With the **CarSource** class in place, we can now add Load- and Save-function­ality to an MVVM application. If we define a **CarModel** class for containing the collec­tion of **Car** objects, we can add **LoadAsync** and **SaveAsync** methods to it:

**private async void LoadAsync()**

**{**

**List<Car> cars = await \_source.LoadAsync();**

**...**

**}**

**private async void SaveAsync()**

**{**

**await \_source.SaveAsync(All);**

**...**

**}**

The (...) just indicates that we might need to do a bit more in each method; we could imagine that we would need to refresh the application GUIafter having loaded data from the file. We have also assumed that the **CarModel** class contains a property **All**, which returns all **Car** objects stored in the model.

With the **LoadAsync** and **SaveAsync** methods in place in the model class, we can now tie the functionality together with a corresponding view model class. In a **Car­Page­View­­Model** class, we can now define **Load** and **Save** methods:

**private void Load()**

**{**

**\_model.LoadAsync();**

**OnPropertyChanged(…);**

**}**

**private void Save()**

**{**

**\_model.SaveAsync();**

**}**

Wrapping these methods up as commands is also pretty simple:

**private RelayCommand \_loadCommand;**

**private RelayCommand \_saveCommand;**

**...**

**\_loadCommand = new RelayCommand(Load, CanLoad);**

**\_saveCommand = new RelayCommand(Save, CanSave);**

**...**

**public ICommand LoadCommand**

**{**

**get { return \_loadCommand; }**

**}**

**public ICommand SaveCommand**

**{**

**get { return \_saveCommand; }**

**}**

**...**

The **async** syntax has ebbed out at this point, so the command-handling code looks just as before. These commands could then be bound to e.g. buttons or maybe menu items in the application. The exact conditions for when to allow **Load** and **Save­** to be exe­­cutable – i.e. the exact code for **CanLoad** and **CanSave** – will of course depend on the requirements for the application.

We have now completed the system of classes needed to add Load/Save function­ality to an MVVM application. Our system looks like this (with a somewhat informal nota­tion):

FilePersistency

<Car>

CarSource

CarModel

CarPage

ViewModel

Reflecting a bit on this system of classes, we could ask ourselves: how much different would this look for loading/saving of e.g. **Customer** objects? Pro­ba­bly not that much different. So, maybe we could create a more generally applicable system of classes? The first step could be to consider the public methods of the **FilePersistency** class as being an **interface** for loading and saving, without even considering it as being based on files:

**public interface IDataSource<T>**

**{**

**Task SaveAsync(List<T> objects);**

**Task<List<T>> LoadAsync();**

**}**

The **FilePersistency<T>** class is then simply a file-specific implementation of this inter­­­face. Following this idea, we could also create a more general data source class **Data­Source<T>**, which would be a very thin wrapper around a **FilePersistency<T>** object. We might even drop the idea of having a separate data source class, which would change the **CarModel** class to something like:

**public class CarModel**

**{**

**private IDataSource<Car> \_dataSource;**

**public CarModel()**

**{**

**\_dataSource = new FilePersistency<Car>();**

**}**

**// LoadAsync and SaveAsync are not changed**

**}**

The price paid for this simplification is that the model class is now tightly coupled to a file-based implementation of the **DataSource<T>** interface. If this is an unacceptable drawback, we could let the creator of the **CarModel** object decide what implementa­tion of the interface to use:

**public class CarModel**

**{**

**private IDataSource<Car> \_dataSource;**

**public CarModel(IDataSource<Car> dataSource)**

**{**

**\_dataSource = dataSource;**

**}**

**…**

**}**

Who creates the **CarModel** object? The **CarPageViewModel** does have an instance field of this type, but it should probably not be a view model class that decides which specific data source to use. A natural next step in this class refactoring exercise would be to define a (type-parameterised) base class for model classes, e.g. called **Data­Model<T>**. An outline of such a base class could be:

**public class DataModel<T>**

**{**

**private IDataSource<T> \_dataSource;**

**public DataModel()**

**{**

**\_dataSource = new FilePersistency<T>();**

**}**

**…**

**}**

In this way, we have isolated the choice of data source to a single place; the base class constructor. A type-specific model class – which might contain type-specific methods – can then simply inherit from this base class. This will hide knowledge about specific data sources from the view model class, even if it is this class that creates a model object. As discussed earlier, we can then also consider to create type-parameterised base classes for view model classes as well.

# Accessing data stored in a relational database

As mentioned above, we will often store data in a relational database, as soon as we move past the simplest of applications. Relational databases is a very large topic in its own right, and we will not attempt to cover it here. Instead, we from now on assume that the reader is familiar with the essential terms and techniques relating to rela­tio­nal databases.

## Creating a local database with Visual Studio

Since databases are an integral part of many applications, Visual Studio offers sub­stan­tial support for working with databases. Microsoft also offers various data­base products as part of its product suite, the primary product being **Microsoft SQL Server** (MSSQL). It is quite easy to create a relational database from within Visual Studio; for use in this chapter, we have created a database called **CarRetailDB**. The database is a local database, i.e. it simply resides on the computer we are using for this example. We will later describe how a database can be deployed to a cloud hosting service.



Again, we will not go into the details about how to create such a database. For com­pleteness, we will however point out that you need to install a couple of additional Visual Studio **workloads**, in order to perform the steps described in this chapter. More specifically, you need to install the workloads

* Data Storage and Processing
* Azure Development

After installing these workloads, you should be able to create a database direct­­ly in Visual Studio. Once a database has been created, a number of tables can be added to it. This can be done either by using a database script, or creating the tables directly in Visual Studio. We have created a table called **Car** in the **CarRetailDB** database.



Once the table has been created, it can be populated with some sample data, again either through a script or by manually entering the data in Visual Studio.

## Accessing data from an UWP application - overview

Creating and populating a database in this way is fairly straightforward; the much more important question is obviously: *how do we access and manipulate data from a relational database in an application written in C#?*

The short answer is: it depends. Numerous technologies for accessing data from a data­base have emerged over the years, but at the time of writing, two main techno­lo­­gies are in play, named the **Entity Framework** and **RESTful Web Services**.

### The Entity Framework

A general problem when working with relational data in an object-oriented langu­age, is to “map” data properly between these two data representation para­digms: relatio­nal and object-oriented. This problem generally is known as the **Object-Relational Mapping** (**ORM**) problem, and several frameworks exist for performing this mapping. In a Microsoft context, a framework known as the **Entity Frame­work** (**EF**) has been the predominant framework for this in recent years. We will not dive into all the details of the Entity Framework, but we do need a bit of history in order to under­stand the current status of the Entity Framework technology.

The first version of EF dates back to around 2008, and was part of the .NET frame­work in general (and thus targeted exclusively for the Windows platform). After a somewhat rocky start, it became the most widely used ORM framework for Windows applications. Currently, EF version 6 is the newest version of the original EF.

As part of of Microsoft’s shift towards a cross-platform .NET version (also known as **.NET Core**), Microsoft also started the process of porting EF functionality to .NET Core, in the form of the **Entity Framework Core** (EF Core) library. Version 1.0 of this library was released in 2016. Along with this push towards cross-platform compati­bi­lity, Micro­soft have also been pushing the **Universal Windows Platform** (**UWP**) appli­ca­tion model, which we are also using in these notes. Here was, however, a very large *cat-in-a-bag*, due to a couple of problems:

* UWP applications are only compatible with EF Core
* EF Core version 1.0 did not support use of SQL Server databases

So, you were essentially at at dead end, if you wished to interact with an SQL Server database from a UWP application through either version of EF… The only feasible alternative was to insert a **web service** (see later) in between the UWP application and the SQL Server database, which is a significant complication.

This was the state of affairs until early 2018, where EF Core version 2.1 was released. This version does support access to SQL Server databases, so it is now indeed possi­ble to interact with an SQL Server database without the web service layer. This is the setup we will pursue in the remaining part of the chater: using EF Core version 2.1+ to interact with an SQL Server database from an UWP application. For completeness, we do provide a brief description of the web service approch as well.

### RESTful Web Service

The Entity Framework thus makes it possible to manipulate data from a relational database with relative ease in C#. This is as such independent of the physi­cal loca­tion of the database, since the location is essentially just a configuration detail. Either way, we will to some extent have tied the application to using a database as the source of data. This may be perfectly fine in many scenarios, but it is possible to ima­gine scenarios where you wish to hide the choice of specific data source behind a more general layer, which could indeed take the form of a web service. The applica­tion will then interact with the web service in order to manipulate application data, and the web service itself may then use a database for data storage, but may in prin­ciple use any media for data storage, without exposing this to the application.

So, what is a web service more specifically. A web service can be thought of as an application running somewhere on the Web – perhaps on a remote computer in a cloud hosting service – and exposing a set of methods which other applications can invoke for for useful purpose. For a web service, the term **Web API** (Application Pro­gramming Interface) is usually used for describing such a set of methods.

How does an application then make use of the methods in such a Web API? Calling such a method is not done in the same way as calling an ordinary C# method in a self-contained application, even though there are similaries. Since the web service and the application (often called the **client**) interact via the Web, they need to follow a Web-oriented “protocol” for comminication. This protocol will usually be the **HTTP**[[2]](#footnote-2)protocol.

Calling methods using the HTTP protocol does as mentioned work significantly diffe­rent than ordinary method calls. A method call is invoked by sending a so-called **HTTP request** (the format of which is part of the HTTP protocol definition) to the web ser­vice. The web service will reply by returning an **HTTP response** to the client. An HTTP request will always contain one of the HTTP “verbs” GET, POST, PUT and DELETE. Such a verb states the intention of the request. The request may also include actual data, which can be thought of as parameters to the method call. The returned HTTP response can also include data, which can likewise be thought of as the return value for the method call. The data itself must also follow a well-defined format, which could e.g. be JSON or XML. A Web API of this kind is often denoted a **RESTful** Web API, since it follows the so-called **REST**[[3]](#footnote-3) paradigm.

An application relying on web services will thus need to perform such method calls using the HTTP protocol. Fortunately, a set of C# classes exist in the .NET class library to assist with this. The library **System.Net.Http** contains classes like **HttpClient**, **Http­RequestMessage**, **HttpResponseMessage** and a few other, which encapsulate some of the technicalities of communicating with a web service. On top of this, it might be beneficial to create your own “wrapper class” around the use of these HTTP-oriented classes, such that the rest of the application interacts with a fairly generic CRUD-like interface, decoupled from any web service-specific implementation details.

## Accessing a database using Entity Framework Core

In the remainder of this chapter, we will focus on how to access a database from a **UWP** application using the **Entity Framework Core** only, and thus not consider the inclusion of web services further.

The first question to answer when embarking on such an implementation is: *where does the definition of domain data structures come from?* We can imagine two sour­ces:

* The object-oriented class definitions
* The database table definitions

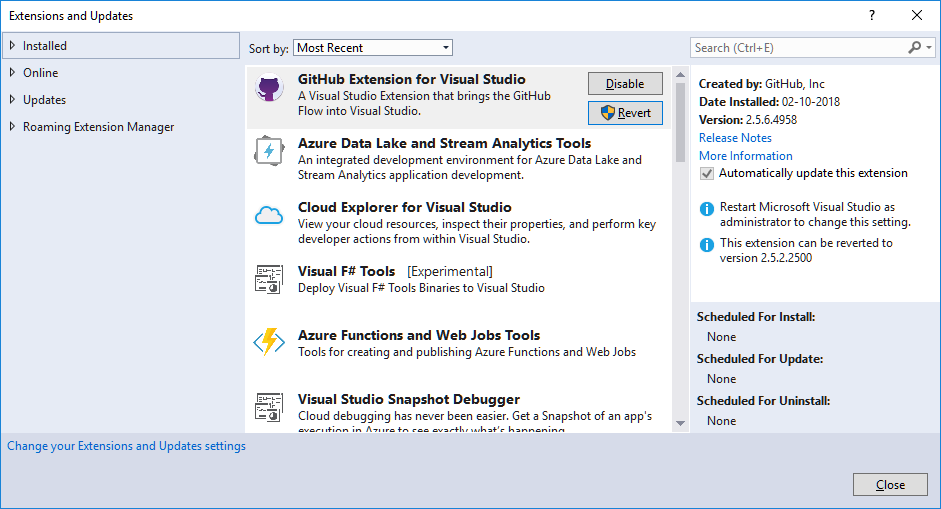
Obviously, these two representations of our data model must be consistent with each other. Making sure that this happens is the responsibility of the ORM frame­work, i.e. the Entity Framework. So, the question is more specifically: *Do we derive the class definitions from the database definition, or do we derive the database definition from the class definitions?*

Approaches of the first category are called **database-first**, while approaches of the second category are called **model-first**. What you choose will of course depend on your development process; we will focus on the database-first approach here.

### Installing the EF Core Power Tools extension

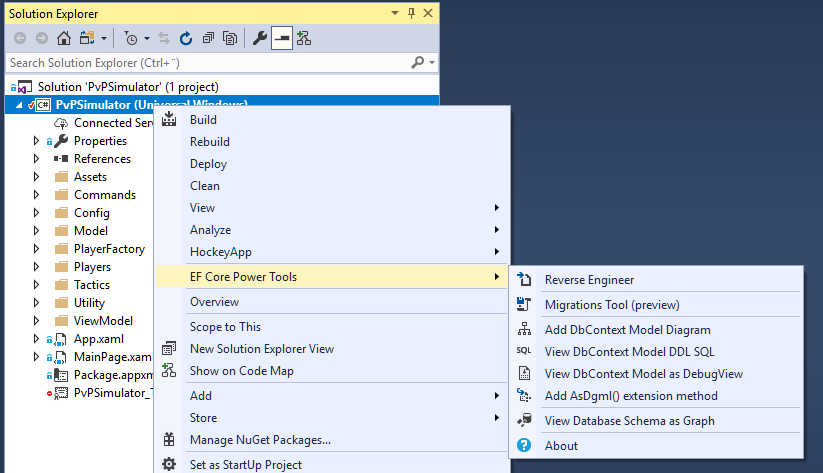
The “traditional” version of the Entity Framework has as mentioned been around for some years, and there is substantial support for it in Visual Studio, most impor­tantly in the form of an **Entity Framework Wizard**, which simplifies the process of genera­ting C# classes needed for connecting to and using a database. At the time of wri­ting (late 2018), no such wizard exists for EF Core, however. Fortunately, a non-Micro­­soft tool does exists which provides wizard-like functionality for EF Core as well. This tool comes in the form of a Visual Studio extension, named **EF Core Power Tools**[[4]](#footnote-4).

The **EF Core Power Tools** extension is installed in Visual Studio just as any other ex­ten­sion. In Visual Studio, choose the menu item **Tools | Extensions and Updates…**, which will open the below dialog:



Expand the **Online** entry in the left side, and type in “EF Core Power Tools” in the search box in the top right corner. This should bring up the **EF Core Power Tools** entry in the extension list in the center of the dialog. Click on **Download**, and install the extension. You may need to restart Visual Studio in order to complete the instal­lation of the extension.

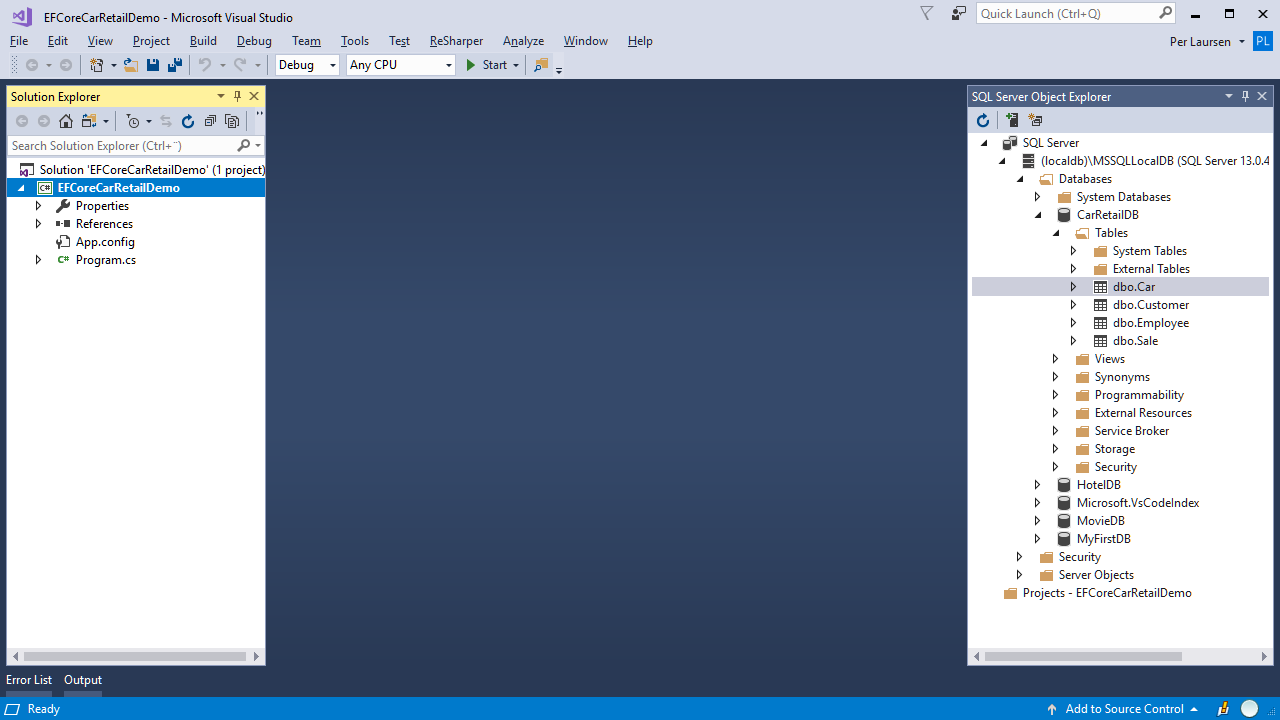
Once the extension has been installed, open up a C# project in Visual Studio, or cre­ate a new project. Open the **Solution Explorer** window, and right-click on the project (not the solution). You should now see an entry in the menu named **EF Core Power Tools**. The menu item has a submenu with several items in it:



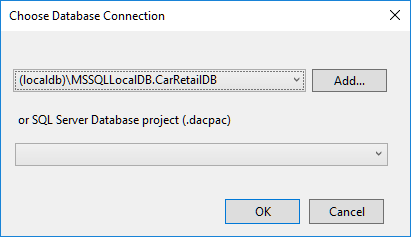
Fortunately, we will only use the functionality available under the **Reverse Engineer** menu item. This name may seem a bit cryptic; it stems from the perception that the usual order of creation is to create classes first, then tables based on classes. Doing it in the reverse order (creating classes based on table definitions) is thus perceived as **reverse engineering** the classes. Still, this is what we want to do ☺.

### Using the EF Core Power Tools

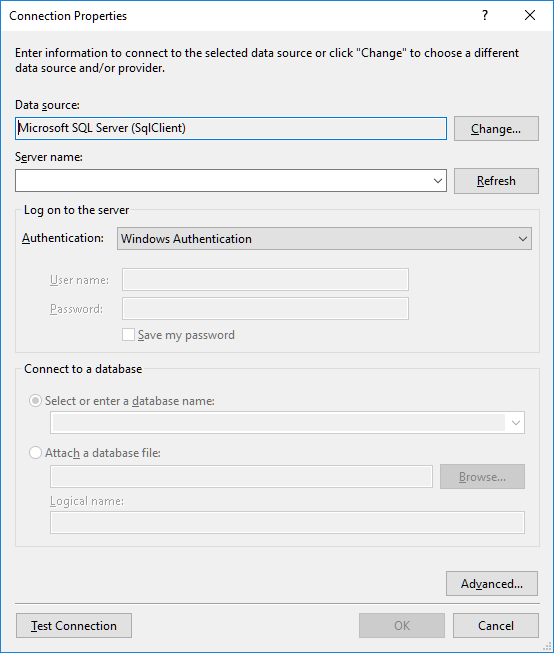
Let’s try out this reverse engineering functionality. In order to keep things simple, we will initially access a database directly from a non-UWP application, e.g. a simple Con­sole application. We use an extended version of the **CarRetailDB** database defined above as an example, where we have added three more tables **Customer**, **Employee** and **Sale**. The starting point is illustrated below:



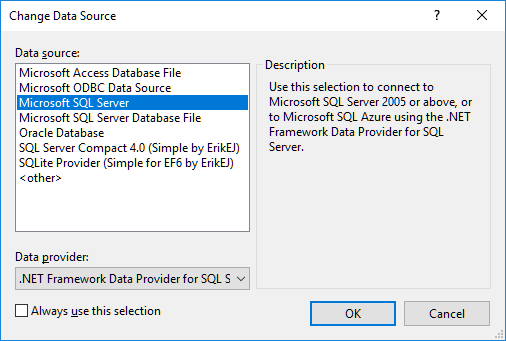
We have a local database **CarRetailDB** as described above, plus a new Console appli­ca­tion **EFCoreCarRetailDemo**. We now right-click on the project, and choose the menu item **EF Core Power Tools | Reverse Engineer**. This brings up the below dialog:



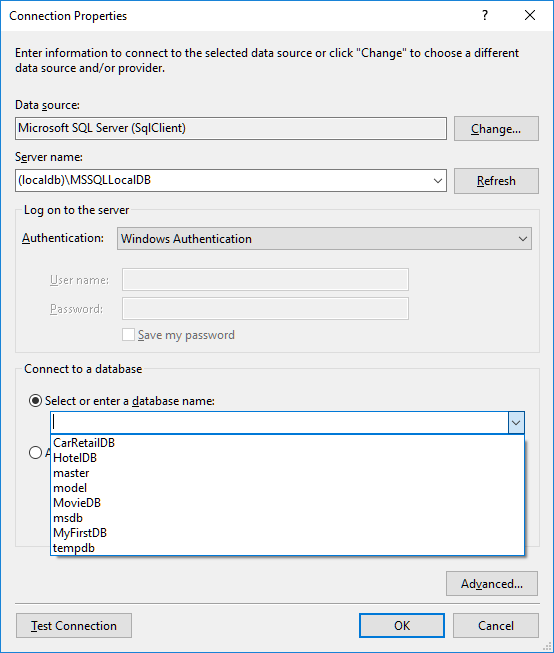
What you actually see in the list box next to the **Add** button will depend a bit on your setup. If you have never used this functionality before, the list box will probably be empty. In that case, we then need to create a new database connection, and do this by clicking on the **Add…** button. This bring up a somewhat larger dialog:



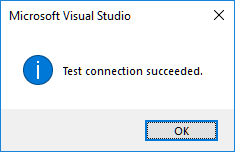
It looks a bit formidable, but once we are done here, the job is almost complete. The first part is to confirm that **Data source** is set to **Microsoft SQL Server (SqlClient)**, as it is in this case. If another data source has been chosen, you just click **Change…**, and choose the correct data source from a list of available data sources:



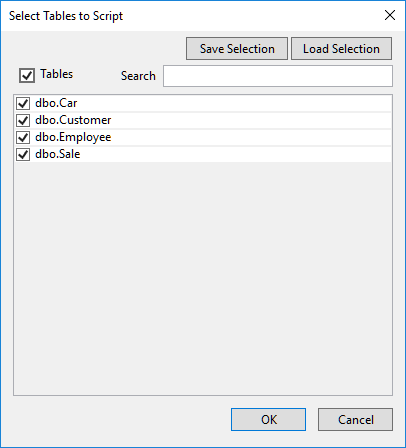
Next, we need to specify the **Server name**. We are (for now) using a local database, which is specified by typing in this exact text (including parentheses, backslash and correct use of upper- and lower-case letters): **(localdb)\MSSQLLocalDB**. Once you have typed in this text, click on the arrow on the right-hand side of the list box under the **Select or enter a database name** radio button. After a couple of seconds, the list box should contain the names of all your local databases:



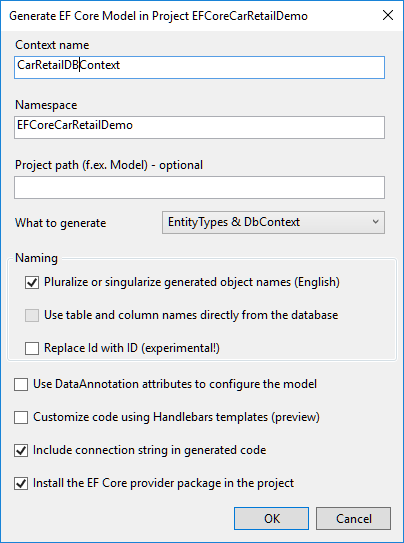
If the list remains empty, check that you have typed the server name correctly, and try again. In this example, we do indeed see the **CarRetailDB** database in the list, so we select it, and click on the **Test Connection** button to verify that we are actually able to connect to the database. This produces – if all goes well – a small congratu­lary dialog:



Having verified the connection, we can – finally – click on the **OK** button, since the connection specification is now complete. This brings up the **Select Tables to Script** dialog:



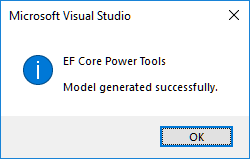
This name is perhaps a bit misleading, since we are actually asked which tables we wish to generate classes for. That aside, the process is usually quite simple; we just select all tables in the database, and click **OK**. This brings us to the final dialog:



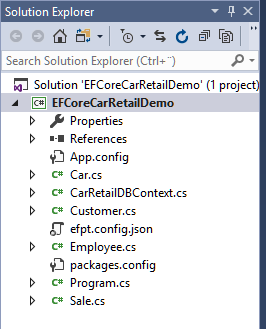
Also a fairly complex dialog, but we usually only need to consider two issues.

* **Choosing a context name**: As we will see in a moment, the wizard will gene­rate several classes for us. More specifically, one class corresponding to each table, and a class corresponding to the entire database. This last class is often referred to as the **database context**, and the naming convention is to give it the same name as the database, plus the word **Context**. In our example, this then becomes **CarRetailDBContext**.
* **Pluralize/Singularize generated object names**: This is more a convenience with regards to the names given to some of the properties in the context class. If e.g. a **Car** class is generated, the context class will contain a collection property containing all **Car** objects. If the **Pluralize…** option is not chosen, this property will be named **Car**, which can easily be confused with the **Car** class itself. If the the **Pluralize…** option is chosen, such properties are pluralised like **Cars**, **Sales**, etc., which is – in the author’s opinion – a better naming convention.

Having considered these two issues, we can click **OK**. This will initiate the class gene­ra­tion process. Note, however, that the first time you execute this class generation process for a project, it may take quite a while to complete, perhaps up to a couple of minutes! The class gene­ration makes use of a couple of EF Core packages, and these packages will need to be fetched and installed as part of the process (you might have noticed the **Install the EF Core provider package in the project** option at the bottom of the dialog). Even more deceptively, it only takes a couple of seconds before it ap­pe­ars like the process is done… Visual Studio looks idle, but it is in fact busy instal­ling the packages. So **BE PATIENT**, and wait until this dialog appears:



The class generation process has now finally completed, and a number of new classes have been added to the project. Try to rebuild the entire project, to check that all is in order. In this example, the project now looks like this:



Compared to the original project, five classes have been added:

* **Car**, **Customer**, **Employee** and **Sale**. Each of these classes correspond to a single table in the database with an identical name.
* **CarRetailDBContext**. This is the class that represents the entire database.

Let’s have a look inside the **CarRetailDBContext** class. It contains quite a lot of code, so this is an abbreviated version:

**public partial class CarRetailDBContext : DbContext**

**{**

**public CarRetailDBContext()**

**{**

**}**

**public CarRetailDBContext(DbContextOptions<CarRetailDBContext> options)**

**: base(options)**

**{**

**}**

**public virtual DbSet<Car> Cars { get; set; }**

**public virtual DbSet<Customer> Customers { get; set; }**

**public virtual DbSet<Employee> Employees { get; set; }**

**public virtual DbSet<Sale> Sales { get; set; }**

**protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)**

**{**

**if (!optionsBuilder.IsConfigured)**

**{**

**optionsBuilder.UseSqlServer(**

**Data Source=(localdb)\\MSSQLLocalDB;" +**

**"Initial Catalog=CarRetailDB;" +**

**"Integrated Security=True");**

**}**

**}**

**protected override void OnModelCreating(ModelBuilder modelBuilder)**

**{**

**// Code for building the object model**

**OnModelCreatingPartial(modelBuilder);**

**}**

**partial void OnModelCreatingPartial(ModelBuilder modelBuilder);**

**}**

We do not need a complete understanding of the code in **CarRetailDBContext** in order to use it, but it is useful to know the essential parts.

The four properties of the type **DbSet<…>** each represent a single table in the data­base, and will contain objects corresponding to the rows in the tables themselves. So, if the **Car** table contains 15 rows, the **Cars** property will contain 15 **Car** objects. This is really the essence of what the Entity Framework does for us; it transforms the con­tent of a table into a collection of C# objects, which we can access and even change (some exam­ples of this follow below) in a way we are familiar with. The Entity Frame­work ensures that the content of the database is changed accordingly!

The method **OnConfiguring** is from our perspective the place where we can specify which database to connect to. The specification comes as a **connection string**, as we have seen before. We usually do not need to change anything here.

The method **OnModelCreating** contains the bulk of the code. It falls in four distinct parts; one part for each table. This is the code for the **Car** table:

**modelBuilder.Entity<Car>(entity =>**

**{**

**entity.ToTable("Car");**

**entity.Property(e => e.Id).ValueGeneratedNever();**

**entity.Property(e => e.Brand)**

**.IsRequired()**

**.HasMaxLength(30);**

**entity.Property(e => e.LicensePlate)**

**.IsRequired()**

**.HasMaxLength(20);**

**entity.Property(e => e.Model)**

**.IsRequired()**

**.HasMaxLength(30);**

**});**

When the Entity Framework builds a class corresponding to a specific table, it will essentially create a property for each column in the table. This is often very straight­forward; if a column e.g. has type **int**, the corresponding property will also have the type **int**. However, some properties need a bit more “decoration”, if the column in the table has cer­tain constraints on it. The **LicensePlate** column has for instance been defined as having the type **nvarchar(20)**, and has the constraint **not null**. This implies that even though the corresponding **LicensePlate** property in the **Car** class has type **string**, not all string values are legal (e.g. a string longer than 20 characters)! This is stated in the **OnModelCreating** code, as a sort of “meta-property” on the **License­Plate** property. These “meta-properties” are added by the method calls **IsRequired** and **HasMaxLength**. With this knowledge, you can probably see that the rest of the code in **OnModelCreating** has a similar nature.

Will you ever need to change anything in the **OnModelCreating** method? It depends a bit on the exact setup of your database, and also whether or not you “manually” (see later) add properties to the generated classes. If this is the case, it might be neces­­sary to e.g. specify that certain properties should be ignored by the model builder. If your setup requires such fine-tuning of the model builder, you should seek out more detailed documentation on the Entity Framework Core online.

Let’s also have a look inside one of the generated domain classes. The **Car** class looks like this:

**public partial class Car**

**{**

**public int Id { get; set; }**

**public string LicensePlate { get; set; }**

**public string Brand { get; set; }**

**public string Model { get; set; }**

**// ... plus some additional properties**

**}**

A fairly simple class, which is essentially just a number of properties of simple types. Note, however, that the **partial** modifier is used in the class definition. The **partial** modifier implies that this is only a part of the **Car** class definition, i.e. that we may define another part of the **Car** class somewhere else. This is actually quite useful!

One of the fundamental ideas in Object-Oriented programming is that a class defini­tion should contain both state (i.e. data) and behavior. What if we during our design have decided that the **Car** class should contain some business logic, in the form of a num­ber of methods. Where should we define these methods? We could add them to the just generated **Car** class (and perhaps remove the **partial** modifier), but this could cause the code to be overwritten if we decided to re-run the class generation wizard! This can be avoided by keeping that part of the **Car** class definition in a separate file. As long as we use the **partial** modifier, this is perfectly legal. It will require a bit of file naming discipline, and you should probably decide upon a naming convention for files containing these partial definitions. One approach could be to name the file containing the manu­ally created part of the class definition for e.g. **CarLogic.cs** (again, remember that this will not imply that **CarLogic.cs** will contain a class called **CarLogic**, but rather a partial defi­nition of the **Car** class), but there are no well-esta­blished naming conventions for how to name such files. Choose a naming convention that makes sense to you, and use it consistently.

### Using the generated classes and properties

All the pieces should now be in place for actually accessing the data from the appli­cation. So, how is it done? The code needed for a simple reading of all rows in a table is quite simple (an override of **ToString** has been added to the **Car** class):

**using (var db = new CarRetailDBContext())**

**{**

**Console.WriteLine("All records in Car table:");**

**foreach (Car c in db.Cars)**

**{**

**Console.WriteLine(c);**

**}**

**Console.WriteLine();**

**}**

First, note the use of the **using** keyword, which we have not seen before. In gene­ral, whenever you create an object which uses some sort of expensive resource (in this case a database connection), it is considered good practice to define it inside a **using** statement as above. This ensures that no matter the outcome of the code inside the code block, the resources claimed by the object will be properly released.

Next, we focus on the code inside the code block. The code written for printing out the **Car** objects is next-to-identical to code we would have written if the **Car** objects were just created directly in the code, and inserted into e.g. a **List** data structure. Instead, we simply refer to the property **Cars**, which was defined in the class defini­tion for **CarRetailDBContext**. That is the power of the Entity Framework; once the connection to the database has been properly specified, all of the work is done behind the scenes. The data itself is then readily available as demonstrated above.

The above code snippet illustrates that reading data from the database is quite easy. Changing the data is also relatively easy. The below code illustrates how to add a new row to a table, i.e. a **Create** operation (note that a constructor has been added to the **Car** class):

**db.Cars.Add(new Car(4, "MN 1234", 80000, "Skoda", "Octavia", 2013));**

**db.Cars.Add(new Car(5, "QR 3456", 30000, "VW", "Polo", 2009));**

**db.SaveChanges();**

That is it! Data is created simply by creating a new object of the relevant type, and ad­ding it to the corresponding **DBSet** property, in this case the **Cars** property. This is very similar to inserting the object into e.g. a **List**. Note, however, that the data is not entered into the database before the call of **db.SaveChanges()**. Also, be aware that when you attempt to insert a new object/row into the database, an excep­tion may be thrown if the data is not in accordance with the rules defined for the table (e.g. that a value may not be **null**, etc.). If such an exception is thrown, the application should of course handle it in an appropriate manner.

Deleting data (i.e. a **Delete** operation) is done using the **Remove** method:

**Car c = db.Cars.Find(2);**

**if (c != null)**

**{**

**db.Cars.Remove(c);**

**}**

Note that the **Remove** method takes an object as parameter, not e.g. a key identi­fy­ing an object. However, the **Find** method can retrieve an object given a (primary) key.

How about updating an object? There is not as such an explicit method for upda­ting an object; you simply update the object itself:

**Car c = db.Cars.Find(2);**

**if (c != null)**

**{**

**c.Price = c.Price + 10000;**

**}**

Again, remember that these changes are not entered into the database before a call of **db.SaveChanges()** is made.

It is thus rather simple to perform the four **CRUD** (Create, Read, Update, Delete) operations on data, once the connection be­tween the database and the generated data structures has been established via the Entity Framework. More sophisticated queries on the data can also be performed, e.g. using LINQ:

**var queryResult = from c in db.Cars**

**where c.Price > 70000 && c.Brand == "BMW"**

**select new {c.LicensePlate, c.Price};**

Seen in an MVVM perspective, the Entity Framework can thus almost provide us with a ready-made Model layer, or at least some data structures which it will be fairly easy to map to a Model layer.

## Deploying a database to the Cloud (Azure)

The databases we have worked with so far have been located on our local computer. This is fine for doing experiments, but usually not sufficient in a real-world scenario, where several clients may need to be able to access a specific database. How can we then “publish” a database in a way that makes the database generally available? This can be achi­eved in various ways, and the specific choice will of course depend on the cir­cum­stances of a specific usage scenario.

One solution could be to move the database into the “cloud”, and provide the client applications with the information needed to connect to the database. All mod­ern cloud hosting service providers offer hosting of SQL Server databases, including all sorts of database security and maintenance services.

One such hosting provider is **Microsoft Azure**, or just **Azure**. Azure is a cloud hosting service offered by Microsoft (other significant providers of such services are Google and Amazon), and is the service we choose to work with here. Azure offers certain limited services free-of-charge, but these services are sufficient for our purposes here. Azure is also – not surprisingly – quite well integrated with Visual Studio.

We will not in these notes make any attempt at describing Azure in depth, and also not detail any specifics about how to obtain an Azure account. Details about account types available can vary from person to person, and also over time. We will therefore in the following assume that the reader has managed to create an account at Azure.

Once you have obtained an account and log into the Azure portal, you will see an ini­tial screen looking something like this[[5]](#footnote-5):



This is your “dashboard”, to which you can “pin” the services you wish to focus on. Our focus will be on creating an SQL database. The general principles for Azure SQL databases are:

* An SQL database runs on a (virtual) SQL Server.
* Many SQL databases can run on the same SQL Server.
* The number of virtual SQL Server instances you can create varies with your subscription – for our purpose, a single server will suffice.

The reason for this two-layering is that you can specify certain settings on the server level, while other settings are specified for the individual database. Settings set on the server level will then apply to all databases running on that particular server.

We will as mentioned just create a single SQL Server instance, on which a single data­base will run. Click on **SQL databases**, and choose **Add** (if this is the first database you ever create, you might see a creation option named in a slightly different way). You should then see something like this:



Here you just enter a suitable name for the database. If you have not created a data­base on Azure before, you will probably not have created an SQL server either. In the **Server** section, click on the right arrow, which opens this view:

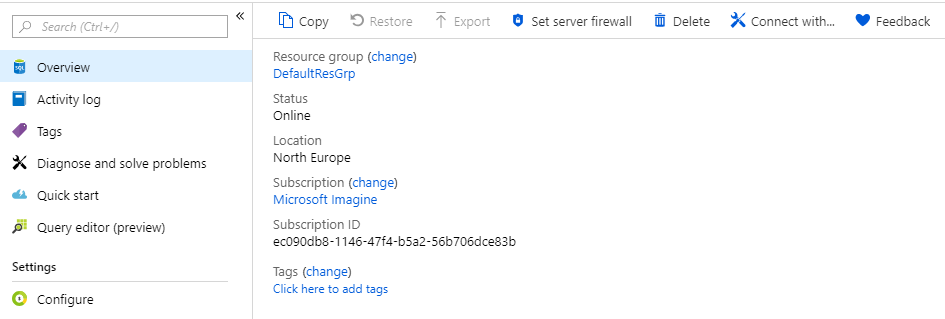


Here we have clicked on **Create a new server** in the middle pane, which opens the right-hand pane. Here you simply enter a suitable name for your database server, and choose an administrator login name and password (NB! **Remember** this login name and password, since you will need it later! Also, do **not** use an email address as user name, since the @-symbol has a special meaning in this context). Once this is done, you click on **Select**, which brings you back to the database specification pane. Finish the specification, click on **Create**, and wait for a while. The database and server crea­tion process can take a few minutes to complete.

When the database is created, it should show up when you choose **SQL databases** from the Azure main menu:



You can click on the database to see more detailed information about it. Note that the database is initially completely empty.

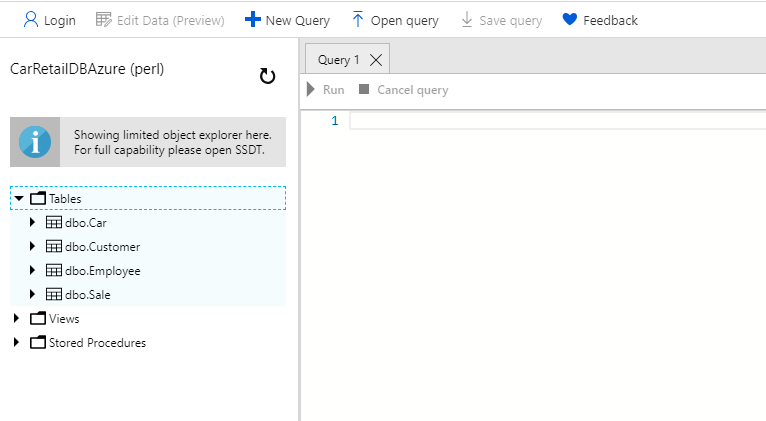


The first item of interest is the **Set server firewall** option. A database is per default not accessible from the outside; you have to specify who can access the data­base. If you choose **Set server firewall**, you will see this dialog:



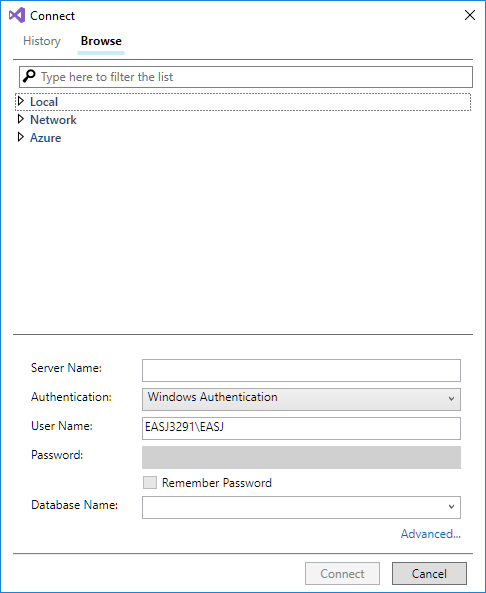
Here you can specify IP addresses which are allowed to access the database server. For convenience, you can add your own computer (or rather, the IP address on which your computer is currently connected) very easily, by clicking on **Add client IP**, and then on **Save**.

Having this out of the way, we now click on **Query editor**. This opens an Explorer-like tool from which queries can be executed on the database:



This is by itself quite useful, since you can e.g. copy-paste a script into the query win­dow. If the script contains queries for table generation and population, it is a conve­ni­ent way to get the data­base up-and-running. Once the database contains tables and rows, you can view and edit the data in a table simply by clicking on one of the tables (this will enable the **Edit Data** button in the toolbar), and click **Edit Data**. After a short detour to a dialog where you must accept a disclaimer saying that this feature is only a preview feature, you can view and edit the data in the selected table.

If you prefer to work with your Azure databases in this manner – which is perfectly fine – there is not much more to know. It is, however, also possible to open a data­base in Visual Studio, if you feel more comfortable in that environment. In the **SQL Server Object Explorer** window, select the entry at the very top (named **SQL Server**), right-click, and select **Add SQL Server…** This brings up the below dialog:



First thing to do here is to change **Windows Authentication** to **SQL Server Authenti­ca­tion** (this will enable the **Password** field). We now need to type in three bits of infor­mation:

* **Server name**: This is the name of the server the database is deployed to. If you go back to Azure and open the page for the database, you can find the name of the database server on the **Overview** page (there is even a neat little option to copy the server name to the clipboard).
* **User name and password**: This is why you had to remember the administrator name and password you speci­fied when creating the database server! Type in the user name and password; if correct, you should now be able to select the database itself in the list box at the bottom on the dialog. Select the database and click **Connect**. The database should now become visible in the **SQL Server Object Explorer** window.



We are now back in our comfort zone, and can work with our Azure database just as if the database was a local database. However, be aware that many operations take con­siderably longer to complete. Just expanding the **Tables** node for a database can take several seconds, as can simple queries.

Once you have created a database on Azure – including table definitions and some data as well – you can now connect to the database from an application just as if it was a local database… almost. There are a couple of minor differences concerning the database connection setup. When we applied the EF Core Wizard previously, we encountered the **Connection Properties** dialog, where a new database connection must be specified:



Compared to our previous encounter with this dialog – where we connected to a local database – the main differences are now:

* Server name must now be the name of your (virtual) SQL server on Azure.
* Authentication must now be set to **SQL Server Authentication**.
* As user name and password, specify the administrator name and pass­word you chose when creating the database server.

As before, you can test if everything is in order by clicking **Test Connection**. When every­thing is in order, click **OK**. You will now see a new dialog, which we have not seen before:



Here we are asked to decide how the password string should be handled. For this small demonstration, we assume that it is okay to save the password in plain text in the connection string. This may not be acceptable in other scenarios. Once we have chosen, the **Next** button becomes enabled, and we can continue the process. The re­main­ing steps are identical to what we have seen before.

When the wizard has been applied successfully, we should indeed be able to connect to the specified database on Azure. Testing this should be fairly simple; we can simply reuse the test code we saw earlier, when we tested the ability to connect to a local database. This is of course only true if the databases are structurally identical (i.e. con­taining identical table definitions), but if that is indeed the case, the difference between connecting to the local database and the Azure-deployed database should boil down to one single line in the source code: the **database connection string**.

The database connection string is found in the **…DBContext** class (perhaps called **Car­RetailDBContext**, or whatever you have chosen to call it), as part of the body of the method **OnConfiguring** (the connection string is quite long, so it has been split into several lines):

**protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)**

**{**

**if (!optionsBuilder.IsConfigured)**

**{**

**optionsBuilder.UseSqlServer(**

**"Data Source=(localdb)\\MSSQLLocalDB;" +**

**"Initial Catalog=CarRetailDB;" +**

**"Integrated Security=True");**

**}**

**}**

This is a connection string for connecting to a local database. Connecting to a data­base on Azure could look like this:

**protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)**

**{**

**if (!optionsBuilder.IsConfigured)**

**{**

**optionsBuilder.UseSqlServer(**

**"Server=tcp:perldbserver.database.windows.net,1433;" +**

**"Initial Catalog=CarRetailDBAzure;" +**

**"Persist Security Info=False;" +**

**"User ID=perl;" +**

**"Password=Th1sIsN0tTh9R9alP4ssw0rd;" +**

**"MultipleActiveResultSets=False;" +**

**"Encrypt=True;" +**

**"TrustServerCertificate=False;" +**

**"Connection Timeout=30;");**

**}**

**}**

A bit more information is needed when connecting to a non-local database, but the point remains valid: shifting from a local version of a database to a remote version only requires a different connection string! The connection string could even be cho­sen at run-time. This also makes testing somewhat easier; until you are done testing the functionality of the application, you can just connect to a local database, and then finally switch to a hosted database once testing has been completed.

# Exercises

|  |  |
| --- | --- |
| **Exercise** | PERSIST.1 |
| **Project** | NoteBookV10 |
| **Purpose** | Add Load- and Save-functionality to an MVVM application |
| **Description** | The given application contains a very simple system for creating notes. A note consists of a title and some content. It is not allowed to have two notes with the same title. However, the application does not support saving and loading of notes yet. |
| **Steps** | In the **NotePageViewModel** class:   1. Add a new instance field **\_loadCommand**, of type **Relay­Command** 2. Add a new property **LoadCommand**, of type **ICommand**. It should just return the instance field **\_loadCommand**, in the same style as e.g. the **AddCommand** property. 3. Add a new method **Load**, which calls **LoadAsync** on the **\_model** instance field, and also calls **OnPropertyChanged** (with what as parameter?). 4. In the constructor, initialise **\_loadCommand** in the same style as in the notes, i.e. with **Load** and **CanLoad** as parameters to **RelayCommand**. 5. In the method **NotifyCommands**, add a call of **RaiseCanExecuteChanged** on the **\_loadCommand** instance field 6. Repeat steps 1-5 for the Save functionality   In the **MainPage.xaml** file:   1. Add two new buttons **Load** and **Save** to the view, and bind them to the **LoadCommand** and **SaveCommand** property, respectively. 2. Rebuild the application, and see if you can now load and save notes. Create some notes, click **Save**, close the application, start it again, click **Load**, and see if the saved notes reappear. |

|  |  |
| --- | --- |
| **Exercise** | PERSIST.2 |
| **Project** | NoteBookV20 |
| **Purpose** | Use exceptions for error-handling in an MVVM application |
| **Description** | The given application checks that notes cannot have the same title (try it!), but the implementation is quite a mess… The handling is all done in the **set**-part of the **Title** property in **NoteDataViewModel**, with several calls to the model and the master-details view model |
| **Steps** | Our aim is to clean up the error handling. This involves using exceptions for error signaling and handling, and also to distribute various responsibilities to the proper classes. An exception class **TitleExistsException** is included in the project.   1. In the **NoteModel** class, add checks to the methods **Add** and **UpdateTitle**, such that a **TitleExistsException** is thrown if the new title exists 2. In the **NotePageViewModel** class, take a look at the method **UpdateTitle** (which is not used yet). See if you understand why the method is struc­tured in this particular way. 3. In the **NoteDataViewModel** class, go to the **set**-part of the **Title** property. Remove ALL the code in the **set**-part, and replace it with a single line of code:   \_viewModel.UpdateTitle(value);   1. Clean up the **NoteDataViewModel** class a bit, since it no longer needs a reference to the model (remove the instance field, and remove the para­ meter from the constructor) 2. Rebuild the application and run it. See if the validation of titles still works as before. 3. See if you can answer the below questions:    1. Which class **detects and signals** the error?    2. Which class **assumes responsibility** for handling the error?    3. Which class **reports** the error to the user? |

|  |  |
| --- | --- |
| **Exercise** | PERSIST.3 |
| **Project** | EFCoreMovie |
| **Purpose** | Use the Entity Framework Core to establish a connection to a local rela­tional database, and to retrieve and alter data in the database. |
| **Description** | In this exercise, we create a local relational database, and create tables and data in the database, by using a given database script. We then try to connect to the database through the Entity Framework Core, and access the data in the tables. |
| **Steps** | The project contains a text file called **MovieDBScript.txt**. This script can be used to generate tables in a database, and insert some sample data into the tables.   1. Create a new local database called **MovieDB**. 2. Run the given script file on the database (Right-click on the database in the **SQL Server Object Explorer** window, choose **New Query**, copy the content of the script file into the query window, and execute the query). The data­base should now contain two tables **Movie** and **Studio**, and both tables should contain some sample data (seven records in **Movie**, three records in **Studio**). 3. In the **EFCoreMovie** project, use the Entity Framework Core to create a class model corresponding to the **MovieDB** database, by using the Entity Framework Core Power Tools as described in the notes. This process should add three new classes named **MovieDBContext**, **Movie** and **Studio** to the project. 4. In order to make it easy to print out **Movie** and **Studio** objects, the files **MovieLogic.cs** and **StudioLogic.cs** (which contain partial definitions of the classes **Movie** and **Studio**) contain an override of **ToString** for each class. Uncomment the code for **ToString** in both classes. 5. In the **MovieDBTest** class, uncomment the code in all the places indicated in the code, and try to run the application. This tests various operations on the database. Make sure you understand what happens in each of the methods. 6. Experiment with creating, updating and deleting some **Movie** and **Studio** objects (you can always recreate the original data by running the database script again). Remember that any changes you make will not be reflected in the database before calling **SaveChanges** on a **MovieDBContext** object. |

|  |  |
| --- | --- |
| **Exercise** | PERSIST.4 |
| **Project** | EFCoreHotel |
| **Purpose** | Use the Entity Framework Core to establish a connection to a local rela­tional database, and to retrieve data in the database. |
| **Description** | In this exercise, we create a local relational database, and create tables and data in the database, by using a given database script. We then try to connect to the database through the Entity Framework Core, and access the data in the tables. |
| **Steps** | The project contains a text file called **HotelDBScript.txt**. This script can be used to generate tables in a database, and insert some sample data into the tables.   1. Create a new local database called **HotelDB**. 2. Run the given script file on the database (Right-click on the database in the **SQL Server Object Explorer** window, choose **New Query**, copy the content of the script file into the query window, and execute the query). The data­base should now contain four tables **Booking, Guest, Hotel** and **Room**, and all tables should contain some sample data. 3. In the **EFCoreHotel** project, use the Entity Framework Core to create a class model corresponding to the **HotelDB** database, by using the Entity Framework Core Power Tools as described in the notes. This process should add five new classes named **HotelDBContext**, **Booking, Guest, Hotel** and **Room** to the project. 4. Take a closer look at the generated classes. Some of them contain proper­ties of class types and collection types, like e.g. the **Rooms** property in the **Hotel** class, of type **ICollection<Room>**. What do you suppose such a pro­perty represents? 5. In order to make it easy to print out objects from the tables, the four files named **…Logic.cs** each contain an override of **ToString** for one of the four domain classes. Uncomment the code for **ToString** in all classes. 6. In **Program.cs**, uncomment the **using** statement under the text *“Printing content of all tables, first attempt”*, and run the application. Study the output. It seems like the references between the objects are broken (what indicates that this is the case?). 7. In **Program.cs**, uncomment the **using** statement under the text *“Printing content of all tables, second attempt”* (and comment the first attempt away again), and run the application. Study the output. What is different? |

1. https://en.wikipedia.org/wiki/JSON [↑](#footnote-ref-1)
2. See e.g <https://www.tutorialspoint.com/http/index.htm> for a brief introduction to HTTP [↑](#footnote-ref-2)
3. See e.g. <https://en.wikipedia.org/wiki/Representational_state_transfer>, or one of the many online REST tutorials [↑](#footnote-ref-3)
4. https://marketplace.visualstudio.com/items?itemName=ErikEJ.EFCorePowerTools [↑](#footnote-ref-4)
5. Azure is an ever-changing enviroment, with tools and services being adjusted constantly. Screenshots will therefore quickly become less than 100 % accurate. [↑](#footnote-ref-5)