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

|  |
| --- |
| By Per Laursen  01-08-2018 |

[Introduction 2](#_Toc521929127)

[File-based Persistency 3](#_Toc521929128)

[Accessing data stored in a relational database 10](#_Toc521929129)

[Creating a local database with Visual Studio 10](#_Toc521929130)

[Accessing data from an application - overview 11](#_Toc521929131)

[Accessing a database using the Entity Framework 13](#_Toc521929132)

[Creating a Web Service for database access 22](#_Toc521929133)

[Creating a basic RESTful web service 28](#_Toc521929134)

[Using a RESTful web service 33](#_Toc521929135)

[Towards a general client-side implementation 35](#_Toc521929136)

[Deploying a database and web service to the Cloud (Azure) 38](#_Toc521929137)

[Exercises 49](#_Toc521929138)

[PERSIST.1 49](#_Toc521929139)

[PERSIST.2 50](#_Toc521929140)

[PERSIST.3 51](#_Toc521929141)

[PERSIST.4 52](#_Toc521929142)

[PERSIST.5 53](#_Toc521929143)

[PERSIST.6 54](#_Toc521929144)

[PERSIST.7 55](#_Toc521929145)

[PERSIST.8 56](#_Toc521929146)

[PERSIST.9 57](#_Toc521929147)

# 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 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 our 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 our 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 **ListView** after having loaded data from the file, by calling **OnPropertyChanged** for a collection property bound to by the **ListView**. We have also assumed that the **CarModel** class contains a property **All**, which returns all of the **Car** objects stored in the model.

With the **LoadAsync** and **SaveAsync** methods in place in the model class, we can now finally tie the functionality together with the 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 exe­cute – 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-based) 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 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:

The **Entity Framework**: A general challenge when working with relational data in an object-oriented language, is to “translate” data properly between these two data representation paradigms: relational and object-oriented. This problem is known as the **Object-Relational Mapping** (ORM) problem, and several frameworks exist for performing this mapping. In a Microsoft context, the most widely used framework is known as the **Entity Framework** (EF). We will investigate the Entity Framework a bit further in this chapter.

**RESTful Web Service**: The Entity Framework 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. Still, the Entity Framework is not always enough. If you wish to access a rela­tional database from a UWP (Universal Windows Platform) application – which has been our platform-of-choice since starting to discuss applications with GUI – it has until very recently (early 2018) not been possible to do this directly! This somewhat surprising situation has been due to some intentio­nal restrict­ions put on UWP appli­cations by Microsoft, and has imposed an extra layer of complexity onto the data­base access problem. Even though the restriction has to some extent been lifted now, we will still discuss how you can alleviate the problem in general, since the solution can also be applied to other scenarios where the exact data source needs to be abstracted away.

This extra layer comes in the form of a **web service**. The web service – which can be hosted locally on in a cloud hosting service, see later – can access a database using e.g. the Entity Framework (the web service itself is not a UWP application), and then make that data available for an external application, through a so-called **Web API**. API in general stands for **Application Programming Interface**, i.e. a set of methods avail­able for other programmers, to invoke functionality offered by a “server”. A Web API thus also consists of a set of “method calls”, which in practice are **HTTP[[2]](#footnote-2) requests** using the standard HTTP “verbs” GET, POST, PUT and DELETE. These calls will in turn generate **HTTP responses**, including the request­ed data in a well-defined format like XML or JSON. A Web API of this kind is denoted a **RESTful** Web API, since it follows the so-called **REST**[[3]](#footnote-3) paradigm.

The complete setup for a scenario involving a UWP application accessing data from a MSSQL database then becomes:



This is a quite significant increase in complexity over the file-based approach to per­sis­tency. Fortunately, some rather powerful tools in Visual Studio exist to help us with the creation of each layer.

## Accessing a database using the Entity Framework

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 model
* 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 model from the database definition, or do we derive the database definition from the class model?*

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.

In order to keep things simple, we will initially access a database directly from a non-UWP application, e.g. a simple Console-based application. We use the **CarRetailDB** database defined above as an example, augmented with a **Customer** table.

Once a project has been generated, the next step is to generate a class model corre­sponding to the database definition. This is done by right-clicking on the project in the **Solution Explorer** window, and then choosing **Add | New Item**. This brings up the **Add New Item** dialog:



In this window, choose **Visual C# Items | Data**



Now choose **ADO.NET Entity Data Model**, provide a name in the **Name** text box at the bottom (the usual naming convention is to use the name of the database, suffixed by the word **Context**, i.e. in this case **CarRetailDBContext**), and finally click **Add**. This will bring up the **Entity Data Model Wizard** dialog:



Initially, it can be a bit hard to see how these four options map to the model-first and database-first categories described before. The two options which are database-first are the **EF Designer from database** and the **Code First from database**. The main dif­fe­­rence is that the former provides a graphical tool for model mapping, while the lat­ter is code-centric. We choose the latter option (**Code First from Database**), even if the name is a bit misleading; it is indeed a database-first approach. This choice brings up the **Data Connection** section of the wizard:



In this window, choose **New Connection**:



A bit of work needs to be done in this dialog. First, make sure that the **Data Source** is set to **Microsoft SQL Server (SqlClient)**; if not, change the selection to that option. Next, the server name for the MSSQL data­base must be specified in the **Server Nam**e text box. You can find the server name in the **SQL Server Object Explorer** window:



In our case, the server name is thus **(localdb)\MSSQLLocalDB**, including the brackets! If you enter the name of your server into the **Server Nam**e text box and click **Refresh**, the drop-down listbox at the **Select or enter a database name** option should become populated with database names (note that the listbox doesn’t actually get populated before you expand it!):



In our case, we do indeed see the **CarRetailDB** in the list. Choose your database in the list, and click **Test Connection** to see if the connection works (a small congratu­latory dialog should appear…). If not, you may need to check if you have typed in the server name, etc. correctly.

When all is in order, click **OK**. This brings you back to the **New Connection** window in the wizard. This should now look something like this:



Now click **Next**. This brings up the **Choose Your Database…** section of the wizard:



In this window, set a checkmark at **Tables**, and also at the **Pluralize…** option. Then click **Finish**. After all little while, the **Solution Explorer** window should look like this:



Note that a number of new classes have been generated:

* **CarRetailDBContext**
* **Car**
* **Customer**

These classes represent the mapping of the selected database (**CarRetailDB**) to a set of C# classes. The **CarRetailDBContext** class represent the database as a whole, while the classes **Car** and **Customer** correspond to the tables **Car** and **Customer** in the data­base. Let us have a look inside these classes.

The class **CarRetailDBContext** should look similar to this:

**public partial class CarRetailDBContext : DbContext**

**{**

**public CarRetailDBContext() : base("name=CarRetailDBContext")**

**{**

**}**

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

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

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

**// Body of OnModelCreating omitted for brevity**

**}**

The most noteworthy elements of this class are the two properties **Customers** and **Cars**. They both have the type **DBSet**, and represent the records in the database tables **Customer** and **Car**, respectively (note that if you see error messages appear in these new classes, it might be because the Entity Framework NuGet Package has not been installed/restored yet. Right-click on the project, choose **Manage NuGet Pack­ages**, and check to see if you need to install/restore the package).

We have not entered any sample data into our database yet. Let us now enter this data into the tables:





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 a table is quite short (a simple 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. 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.

A valid question at this point is: how does the application know which database to con­nect to? This information is not found in the source code as such, but rather in the application configuration file **app.config**. This file contains various configuration infor­mation in XML format, including an element called **connectionStrings**:

**<connectionStrings>**

**<add name="CarRetailDBContext"**

**connectionString="**

**data source=(localdb)\MSSQLLocalDB;**

**initial catalog=CarRetailDB;**

**integrated security=True;**

**MultipleActiveResultSets=True;**

**App=EntityFramework"**

**providerName="System.Data.SqlClient" />**

**</connectionStrings>**

This is where the connection information is maintained.

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 record to the data, 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. 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, you should be aware that when you attempt to insert a new object/record 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 the **Update** operation? There is no 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 the call of **db.SaveChanges()**.

It is thus not particularly complicated to perform **CRUD** (Create, Read, Update, Delete) operations on data, once the connection be­tween the database and the exposed 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.

## Creating a Web Service for database access

If we are able to utilise the Entity Framework directly in a UWP application, our job is almost done now. If this is not possible – or if we for some other reason do not want to access the database directly – we need to insert an extra layer of indirection, in the form of a **web service**.

The container for the web service is a separate application, of the type **ASP.Net Web Application**. In a new or existing Visual Studio solution, choose **Add | New Project**. Among the available of project types, choose **Web | ASP.NET Web Applica­tion**.



Choose a suitable name for the web service, and click **OK**. This brings up the below dialog:



Make sure that **Web API** is highlighted, and that **MVC** and **Web API** are ticked. Before proceeding, click on **Change Authen­tication**, to bring up the below dialog:



In this dialog, confirm that **No Authentication** has been selected (if not, then select it), and click **OK**. Also click **OK** in the previous dialog – this will set the creation of the web service project in motion. After a little while, the project should become visible in the **Solution Explorer** window:



This may look a bit overwhelming, but fortunately we do not need to add that much in order to actually implement the web service. The next step is to go through the same steps as before, concerning how to add a “data item” of the type **ADO.NET Entity Data Model** to the project[[4]](#footnote-4). This is hopefully not so surprising, since we want the web service to be able to connect to the database using the Entity Frame­work, just as we did in the previous example. Once these steps have been repeated, you should have the same classes added to the project as before:



We do, however, need to add one small addition to the auto-generated code. In the constructor of **CarRetailDBContext** class, we add these two lines:

**public MovieDBContext() : base("name=MovieDBContext")**

**{**

**base.Configuration.LazyLoadingEnabled = false;**

**base.Configuration.ProxyCreationEnabled = false;**

**}**

The reasons for this addition are a bit technical, but it is essentially about preventing the service from trying to be too “smart”, with regards to how data is fetched by the service. With this addition, we should avoid unexpected behavior in that regard.

Staying on the topic of slightly obscure actions to perform before proceeding, you should also open the **Properties** window for the web service, and choose the **Web** pane. This should look something like this:



Two things are of interest here: First, we can see the specific URL for this project, i.e. the URL the client will need to specify when using the web service. Second, you must click on the **Create Virtual Directory** button, for reasons that are not yet entirely clear to me…

This completes the “back-end” of the web service. The remaining part is to make the data available to users of the web service. We said earlier that a web service essen­tially works by having a client send **HTTP requests** to the web service, which will then respond by sending **HTTP responses** to the client, including data in a suitable format, e.g. XML or JSON. This can be achieved in many different ways; one way – which has become somewhat of a current standard for web services – is to implement a so-called **RESTful** web service. **REST** (**RE**presentational **S**tate **T**ransfer) is in itself not a specific standard, but rather a set of recommendations about how a web service should be implemented. A RESTful web service essentially boils down to this:

1. A client sends a **HTTP request** – which has a standardised format – to the web service. The request contains three items of interest:
   1. An **HTTP Method Identifier**. Such an identifier is just a word, and will for most RESTful web services be either GET, PUT, POST or DELETE.
   2. A **URI** (Uniform Resource Identifier). This will also just be a string, and could e.g. be the string “api/Cars” or “api/Cars/2”
   3. **Data**. Depending on the action the web service should perform, the request may or may not include data.
2. When the web service receives the HTTP request, it will initiate actions corre­sponding to the content of the request (see below). These actions may involve:
   1. Retrieving or updating ( i.e. performing a CRUD operation) on the data source, which can e.g. be a database.
   2. Constructing an **HTTP response**, which may or may not contain data, depending on the operation performed by the server.
   3. Sending the HTTP response back to the client.

This may seem somewhat obscure, and it is definitely hard to see how this scheme can provide the data and actions we need. The crucial point is this: certain combi­nations of HTTP method identifiers and URI strings are defined as having a specific meaning, and will cause the web service to invoke specific actions. The mapping of such combinations to specific actions is what constitutes a so-called **REST API**.

The first stage of defining such an API, is to agree upon the combinations of HTTP method identifiers and URI strings which have specific meanings. A basic REST API typically includes these combinations (we use **Car** as an example below):

|  |  |  |  |
| --- | --- | --- | --- |
| **HTTP Request** | | | **HTTP Response** |
| **HTTP method** | **URI** | **Data** | **Action (by web service)**  ***Response*** |
| GET | /api/Cars | none | **Read** all Car objects in data source  *Return data to client* |
| GET | /api/Cars/[id] | none | **Read** Car object with this [id]  *Return data to client* |
| PUT | /api/Cars/[id] | Car data | **Update** Car object with this [id]  *Response code indicating ok/error* |
| POST | /api/Cars | Car data | **Create** Car object  *Response code indicating ok/error* |
| DELETE | /api/Cars/[id] | None | **Delete** Car object with this [id]  *Response code indicating ok/error* |

As you can hopefully see from the response column, we are now getter much closer to something resembling database operations (CRUD operations). In fact, the last four operations correspond exactly to single-record CRUD operations, while the first ope­ra­tion is a Read-All (Load) operation. Whether or not the request/response carries any data with it will obviously depend on the nature of the operation: A Read operation will of course return data, while an Update operation may just return an “OK” code.

The above description has hopefully convinced you that such an API can be defined in a meaningful way. Still, there are several practical questions to consider, like:

* How does a web service know what method to activate, when a specific request is received?
* How do we specifically construct such HTTP requests and responses?
* How is data “packaged” to be part of a request/response?

Fortunately, we are again aided by the tools available in Visual Studio ☺

## Creating a basic RESTful web service

The strategy for implementing a RESTful web service within an ASP.Net Web Applica­tion is to add so-called **controllers** to the application. You may have noticed that the web service project contains a folder named **Controllers**. This is where these control­lers will reside. If you right-click on the **Controllers** folder, and then choose **Add | New Scaffolded Item**, the below dialog will appear:



What does “scaffolding” mean in this context? The idea is that since we have already created model classes through the Entity Framework – which are thus connected to an underlying database – it is possible to auto-generate code that implements a basic RESTful web service for a specific model class. You can perceive the generated code as a “scaffold”, to which you can additional code if needed.

In the dialog, select the **Web API 2 Controller with actions…** option, and click **Add**. A new dialog now appears:



In the **Model Class** listbox, we can choose the **Car** class, and choose the “data con­text” class as well (this is the **CarRetailDBContext** class, which represents the data­base as a whole). When this is done, we click **Add**. After a short while, a class named **CarController** will be added to the **Controllers** folder (if you get an error during this process, you may need to rebuild your entire solution, and try again). Once this is done, you can repeat the procedure for the **Customer** class, which should generate a **CustomerController** class as well.

Before taking a closer look at one of these controller classes, let us briefly review the web service project as such. With the addition of the two controller classes, the pro­ject is essentially done. So…can we run it? What does it even mean to “run” a web service? A web service is not an application that a user interacts directly with; it is an application that is started at a “server” computer somewhere on the Internet, and other applications (often called “clients”) can then interact with the web service, by sending HTTP requests to it as described above. If you right-click on the web service project and choose **Set as Startup Project**, you will see that the familiar **Start** button (with the small green tri­angle), has now been replaced with a button labelled **Google Chrome** (or whatever browser you have chosen as your standard browser). If we click on the button, Google Chrome (or another browse) will be launched, and after a few seconds, a window will appear:



This is the “GUI” of our web service. However, the GUI is not intended as something a regular user will interact with. The GUI is more of a “help page”, intended for those who wish to create an application which interacts with the web service. You may also notice that the **Address** field in the browser reads **localhost:30833** (the number after the : may vary from project to project). The web service currently runs as a local web service, i.e. it runs on your own computer. We will later see how to deploy a web ser­vice to the Cloud. For development purposes, it is however fully adequate to run the web service locally.

Another interesting feature is the menu entry called **API**. If we click on this menu entry, the below window appears:



The window is divided into three sections: **Customers**, **Values** and **Cars**. The **Values** part is only included because the **Controllers** folder initially contains a sample class called **ValuesController**. If you find this class is disturbing the overview of the web service, you can safely delete it.

The **Cars** and **Customers** sections are thus the interesting sections. Each section con­tains a description of the web service API for that particular class. You can hope­fully see that this matches the earlier description of a basic RESTful Web API. We can actu­ally try out the API already. If we type **localhost:30833/api/Cars** in the **Address** field of the browser, you should see something like the below appear:



This is the data we originally stored in the **Car** table in our database! The data is here shown in XML format, but that is not in itself particularly important. The important point is that we can now pull data out of a database, transform it to objects using the Entity Framework, and make the data available to the outside world through a REST­ful web service! You can also try to write **localhost:30833/api/Cars/2** in the **Address** field, which should produce this result:



This is the parameterised version of the Read operation, which only reads a single object. Feel free to try out similar operations with the **Customers** API.

We now have a basic RESTful web service up-and-running for our two classes **Car** and **Customer**. Notice how little code we had to write ourselves, in order to make these ends meet. This again illustrates the power of the tools; once we have defin­ed the domain classes themselves (here by means of table definitions in a database), the remaining infrastructure can almost be generated automatically. Before proceed­ing to the client side, let’s have a look inside a controller class. A “collapsed” version of the **CarsController** class looks like this:

**public class CarsController : ApiController**

**{**

**private CarRetailDBContext db = new CarRetailDBContext();**

**// GET: api/Cars**

**public IQueryable<Car> GetCars()**

**// GET: api/Cars/5**

**[ResponseType(typeof(Car))]**

**public IHttpActionResult GetCar(int id)**

**// PUT: api/Cars/5**

**[ResponseType(typeof(void))]**

**public IHttpActionResult PutCar(int id, Car car)**

**// POST: api/Cars**

**[ResponseType(typeof(Car))]**

**public IHttpActionResult PostCar(Car car)**

**// DELETE: api/Cars/5**

**[ResponseType(typeof(Car))]**

**public IHttpActionResult DeleteCar(int id)**

**}**

First of all, we see that the **CarsController** class inherits from the **ApiController** class, which comes from the **System.Web.Http** class library. The **ApiController** base class contains a lot of the generic code for implementing a RESTful web services, so the only respon­sibility left to the **CarsController** class is to implement the five **Car**-speci­fic API methods. The issue of “routing” web service calls (in the form of an HTTP request) to the correct method is also handled by the auto-generated code.

With all this auto-generated goodness available, we do as such not need to change any of the code in a controller class. Still, it may be interesting to peek inside a couple of the method definitions. The **GetCars** method – which returns all cars in our **Car** table – is as simple as it gets:

**// GET: api/Cars**

**public IQueryable<Car> GetCars() { return db.Cars; }**

The **GetCar** method has a little bit more meat on its bones:

**// GET: api/Cars/5**

**[ResponseType(typeof(Car))]**

**public IHttpActionResult GetCar(int id)**

**{**

**Car car = db.Cars.Find(id);**

**if (car == null) { return NotFound(); }**

**return Ok(car);**

**}**

This is not so surprising, since we have two possible outcomes of this call: either a **Car** object with the given **id** was found, or it wasn’t… The **NotFound** and **OK** methods – which are defined in the **ApiController** base class – generate the appropriate HTTP responses, relieving us from dealing with the details of how to do this.

The **DeleteCar** method is structurally similar to **GetCar**, while the methods **PutCar** and **PostCar** are a bit more complex, and require a bit more effort to fully understand. Still, the main point is that the web service is now up-and-running, and we can turn our attention to how we then use the web service.

## Using a RESTful web service

To keep things simple, we illustrate use of a web service with a simple console appli­cation first. The main principles for use are of course independent of this choice.

An application using a web service is usually referred to as a **client**. We use that terminology as well. In order to make use of a web service, a client needs to know:

* The **URL** by which the web service is available
* The **API** offered by the web service

For our web service, the API follows the standard for a basic RESTful web ser­vice, and the service is available on a local URL (**localhost:****30833** in our example). With that infor­mation, the first parts of the client code can be written. Again, we make heavy use of classes from the .NET class library[[5]](#footnote-5):

**const string serverUrl = "http://localhost:30833";**

**HttpClientHandler handler = new HttpClientHandler();**

**handler.UseDefaultCredentials = true;**

**using (var client = new HttpClient(handler))**

**{**

**client.BaseAddress = new Uri(serverUrl);**

**client.DefaultRequestHeaders.Clear();**

**client.DefaultRequestHeaders.Accept.Add(**

**new MediaTypeWithQualityHeaderValue("application/json"));**

**// api-specific code follows…**

**}**

The above code can be considered client “boilerplate code”, and we will not discuss the details of it. The essense of the code is to make the URL of the web service known to the code, and to prepare an **HTTP­Client** object for use. It is through this **HTTP­Client** object we invoke the web service. We can now write code to perform specific API calls. A very simple operation is to delete a **Car** object, given its unique identifier **id**:

**await client.DeleteAsync($"api/Cars/{id}");**

The return type of **DeleteAsync** is **Task<HttpResponseMessage>**, making it possible to use the **await** operator here. This code style is admittedly a *call-and-forget* style; a more robust implementation should do some error handling by examining the nature of the **HttpResponseMessage**. The point here is to illustrate how simple it is to get up-and-running with a web service, from a client perspective. Client code for reading all **Car** objects is slightly more complex:

**Task<HttpResponseMessage> rTask = client.GetAsync("api/Cars");**

**await rTask;**

**if (rTask.Result.IsSuccessStatusCode)**

**{**

**var cTask = rTask.Result.Content.ReadAsAsync<IEnumerable<Car>>();**

**await cTask;**

**foreach (var car in cTask.Result)**

**{**

**Console.WriteLine(car);**

**}**

**}**

## Towards a general client-side implementation

The next step forward from here could be to create some sort of suitable “wrapper” or base class around the specific web service calls. Since the boilerplate code is iden­tical for all the domain-specific web service calls, it should be straightforward to sepa­rate it into a base class method. Also, the domain-specific web service calls only differ on a few places – e.g. the specific name of the API methods – meaning that it should also be possible to create parameterised versions of wrappers around these calls, and place them in a base class as well.

These considerations are similar to those considerations we discussed for the file-based approach to persistency. During that discussion, we suggested a simple, but general interface for persistency functionality:

**public interface IPersistentSource<T>**

**{**

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

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

**}**

Would it make sense to implement a version of this interface, using a RESTful web service as the persistency provider? Concerning the **LoadAsync** method, it would make good sense. Assuming we have created a class **WebAPISource** which imple­ments **IPersistentSource**, and equipped it with some suitable instance fields and helper methods, an implementation of **LoadAsync** could be something like:

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

**{**

**using (var client = new HttpClient(\_httpClientHandler))**

**{**

**InitHTTPClient(client);**

**Task<HttpResponseMessage> responseTask =**

**client.GetAsync($"api/{\_apiID}");**

**await responseTask;**

**if (responseTask.Result.IsSuccessStatusCode)**

**{**

**Task<IEnumerable<T>> readTask =**

**responseTask.Result.Content.**

**ReadAsAsync<IEnumerable<T>>();**

**await readTask;**

**return readTask.Result.ToList();**

**}**

**return null;**

**}**

**}**

The **SaveAsync** method is more problematic. What is the actual meaning of “saving”? The assumption has so far been that “saving” means:

* All existing data is deleted from persistent storage
* The data provided as parameter to **SaveAsync** is saved to persistent storage

This is a rather crude strategy for saving data, which might be sufficient for file-based persistency, where a small amount of data is saved in a local file. For a REST­­ful web service-based implementation, the first problem is that there is no method in the API that directly matches the intended functionality for **SaveAsync**. If we insist on imple­menting **SaveAsync**, we have two options:

**Create a custom API Save method**. This is possible, since you are free to add custom methods to the basic RESTful API. The five “standard” API methods we have seen so far are implemented by auto-generated code, but you can supplement this code with hand-written implementations. A **Save** method could e.g. be implemented by send­­ing an HTTP Request consisting of (using **Car** as an example)

* The **DELETE** verb
* The URI **/api/Cars**
* A set of **Car** object, on JSON format

The meaning of this call should then be *“delete all* ***Car*** *objects from persistent storage, and then write all of the provided* ***Car*** *objects to persistent storage”*. This is identical to how **SaveAsync** works when using files.

**Implement SaveAsync on the client side**. The standard API only provides single-object creation and deletion. A client-side implementation would then require:

* Get all identifiers for all objects currently stored. This can be achieved by calling the Read-All API method, and then filter out all identifiers
* Delete all currently stored objects. This requires calling the Delete method for each of the identifiers from the first step.
* Insert all of the provided objects. This requires calling the Create method for each object in the provided collection

Both options are feasible, but the latter sounds very inefficient. The amount of API calls across the network would be excessive; at least twice the number of objects in the collection at hand. This leaves the first option as the only viable option.

Should we then pursue this approach? An alternative could be to consider if the origi­nal **IPersistentSource** interface was too simplistic from the start. An alternative definition of the interface could be (we have omitted the **Async** suffix for brevity):

**public interface IPersistentSource<T>**

**{**

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

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

**void Create(T obj);**

**T Read(int id);**

**void Update(int id, T obj);**

**void Delete(int id);**

**}**

Can we implement this interface with the file-based and Web API-based persistency approach? Not without effort, at least. The starting point with regards to interface compliance is this:

|  |  |  |
| --- | --- | --- |
| ***Method*** | ***File-based*** | ***Web API-based*** |
| **Save** | OK | No |
| **Load** | OK | OK |
| **Create** | No | OK |
| **Read** | No | OK |
| **Update** | No | OK |
| **Delete** | No | OK |

There is no clear-cut solution to this situation: one approach could be to go with this “broad” interface, and then let the implementations throw exceptions in case a non-compliant method is called. This sort of breaks the idea of client transparency with regards to data source, but could be a pragmatic solution. The client-side code can as such be written at interface level (and thus unaware of the specific choice of persis­ten­cy medi­um), but a conscious decision must be made at a higher level, concerning which stra­tegy to go with. That is, if a file-based approach is used, it will make sense to create an application with load/save functionality available (e.g having **Load** and **Save** entries in a main menu), while a Web API-based application should not have an explicit save functionality available.

We will not at this point deem one approach better than another, but you should be aware that some thought needs to go into how persistency manifests itself in a speci­fic applica­tion.

## Deploying a database and web service to the Cloud (Azure)

The web services we have looked at so far have been deployed (i.e. been located and invoked) on our local computer. This is fine for doing experiments, but sort of defies the purpose of a web service, which is to make the web service available through the World Wide Web. How can we then “publish” a service, so it becomes available? This can be done in different ways, but it essentially comes down to these steps:

* Find a web service hosting provider, to which our web service can be pub­lished. The web service will then run “in the cloud” at this provider.
* Create a data source – e.g. a database – for the web service, and set up the web service to use the data source. This will often mean publishing the data source itself to the web service provider as well.

One such hosting provider is **Microsoft Azure**, or just **Azure**. Azure is a 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 of 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:



This is your “dashboard”, to which you can “pin” the services you wish to focus on. Our first task will be to create an SQL database, to which we will later connect our web service. 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.
* You can create many (virtual) SQL Server instances.

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 here just create a single SQL Server, 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.



Two options are of interest initially: **Tools** and **Set server firewall**. A database is per default not accessible from the outside; you have to specify who can access the data­base (note, however, that databases per default can be accessed by your own Azure web services). If you choose **Set server firewall**, you will see this:



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 **Tools**. This brings up a rather small set of available tools:



All we can do with a database here is either to execute raw SQL queries against the database, or open the database in Visual Studio. The former option is useful enough, 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. We do however choose to open the database in Visual Studio, to leverage our experience with working with databases in Visual Studio. If that option is chosen, you will be prompted about opening Visual Studio. Accept this, and you will be transferred to Visual Studio. Here you will be met with a login dialog:



This is why you had to remember the administrator name and password you specified when creating the database server! Type in the password; if correct, the database will open in Visual Studio (Note that if you are now met with a dialog about not having set firewall settings, you must go back to Azure and set the firewall settings for the SQL server, as described previously):

 

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

A convenient consequence of opening the database in Visual Studio, is that we can now create a web service connected to this database, going through the same steps as we have described previously. The only difference appears during the steps of the Entity Framework Wizard, where a new database connection must be specified:



As compared to our previous encounters with this dialog, the main differences are:

* Server name must now be the name of your (virtual) SQL server on Azure. If you open the database in the Azure portal, you can find the full name of the server in the **Overview** pane.
* 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 (yes, it really was important to re­mem­­ber it…)

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 readable form 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.

Once we have completed the Entity Framework Wizard, we continue – just as before – with the creation of Controller classes; one for each domain class. This brings us quite close to the finish line. Assuming that we have executed a suitable script on our Azure database (if not, we couldn’t generate the domain classes with the Entity Fra­me­work…), we now have:

* An up-and-running SQL database on Azure
* A web service capable of accessing said database through the Entity Frame­work, and exposing the data by its REST API methods.

The only step left is how to “publish” the created web service to Azure. This is actually fairly easy. Once the web service project can build properly, you highlight the project in the **Solution Explorer**, right-click and choose **Publish** in the context menu:



This bring up a simple project page, where the specific kind of publishing can be selected:



Here we choose **Create New**, click on the Azure option, and then **Publish**. This brings up a slightly compli­cated dialog:



Depending on whether or not you have already logged on to Azure from Visual Stu­dio, the first hurdle may be to actually log on. Once that is done, you have to decide on a couple of properties for your new web service. First, the web service will get a name on Azure. This must be a unique name, which is why the auto-suggested name looks rather odd… Feel free to choose a more de­scrip­tive name for your service, as long as it is unique. You also have to decide what subscription and resource group the web service should belong to; the default values are probably fine. Finally, the web service must be associated with a web service “plan”. A web service may in real life require few or many resources in terms of memory and CPU power, which are not free at Azure (the smallest plans are actually free, like the plans we are using here). The choice of plan thus decides how many resources Azure assigns to the web service, and how much you are then billed for this service… Again, don’t worry; if you have chosen a free subscription, you cannot suddenly be billed ☺. In short; choose an existing plan if you have one, otherwise create a new one.

When all properties have been specified, we are ready to publish the web service. Click on **Create**, and the publishing process will start. This process may take a few minutes to complete, and will hopefully culminate with your browser displaying something like this:



Success! A web service available on the specified not-so-descriptive URL is now up-and-running. A first small test could be to click on the **API** entry in the menu, and see if the REST API methods for the domain classes show up. If that is the case, we can try to actually use the REST API methods directly from the browser:



This is indeed data matching the data present in the SQL database running on Azure. We have thus completed our mission of deploying both an SQL database and a web service to Azure.

What if you want to update your web service later on? This is also quite easy. If we return to Visual Studio, we should see something like this after a successful publish:



If we now change something in the Web Service code, we can publish the change by choosing **Publish** from the project context menu (just as before), which brings up the **Publish** window above. Simply click on the **Publish** button; this will start a new pub­lishing process, again ending with the browser showing the updated web service in action.

This concludes the creation of the web service, now running on Azure. We can now create a client which uses the web service, just as we did earlier. In fact, if we already have a client which uses a local version of the same web service (i.e. a web service ex­po­sing the same REST API), the only thing we need to change is the URL for the web service! The URL could even be chosen at run-time. The deciding factor is the REST API, while the two web services can be considered two implementations of the same interface.

# 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** | DBandEFMovie |
| **Purpose** | Use the Entity Framework 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, 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 **DBandEFMovie** project, use the Entity Framework to create a class model corresponding to the **MovieDB** database, by following the guide­lines in the notes closely. 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, add an override of **ToString** to each class. (tip: call **TrimEnd(' ')** on string proper­ties to get rid of trailing spaces for strings). 5. In **MovieDBTester**, uncomment the code in the methods **CreateMovies**, **DeleteMovies** and **UpdateMovies** (and remove the **throw** statement!) 6. Implement the methods **PrintAllMovies** and **PrintAllStudios** in the class **MovieDBTester**, using the style suggested in the notes. The methods are already being called from **Main** in **Program.cs** - test that the methods produce the expected result. 7. 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 the **MovieDBContext** object. |

|  |  |
| --- | --- |
| **Exercise** | PERSIST.4 |
| **Project** | DBandEFHotel |
| **Purpose** | Use the Entity Framework 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, 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 **DBandEFHotel** project, use the Entity Framework to create a class model corresponding to the **HotelDB** database, by following the guide­lines in the notes closely. This process should add five new classes named **HotelDBContext**, **Booking, Guest, Hotel** and **Room** to the project. 4. Verify that you are now capable of retrieving and altering the data from the database, e.g. by writing some methods similar to the methods used in the previous exercise. 5. Take a closer look at the classes generated by the Entity Framework. Note how some of the classes have relations to other classes (for instance **Hotel** and **Room**). Compare the relations between classes with the information in the database script, more specifically the information about foreign key relationships. How does a foreign key relationship “translate” into a class relationship? |

|  |  |
| --- | --- |
| **Exercise** | PERSIST.5 |
| **Solution** | DBEFandWSMovie (contains two projects) |
| **Purpose** | Use the Entity Framework and a RESTful Web Service to enable a client to access data in a local 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, and expose the data by using a RESTful Web Service. |
| **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** (if you don’t have it already). 2. Run the given script file on the database. The data­base should now con­tain two tables **Movie** and **Studio**, and both tables should contain some sample data. 3. In the **DBEFandWSMovieServer** project, use the Entity Framework to create a class model corresponding to the **MovieDB** database. Make sure that the classes **Movie** and **Studio** are generated. Also remember to update the **MovieDBContext** constructor with the two calls to the base class described in the notes. 4. Still in the **DBEFandWSMovieServer** project, now create a controller for both the **Movie** and **Studio** classes (one controller for each class), by fol­lowing the guidelines in the notes. This should generate a **MoviesControl­ler** and a **StudiosController** class in the **Controllers** folder. 5. Once the controllers are generated, the web service is ready to run. You can test it by right-clicking on the **DBEFandWSMovieServer** project, and choose **View | View in browser**. You should see the standard ASP.Net start page. If you choose **API** in the menu, you should see the web service API for **Studio** and **Movie**, both containing five methods. 6. Once the web service is up-and-running, you should be able to compile and run the **DBEFandWSMovieClient** project. That project makes use of the web service server, by calling some of the methods in the web API. The test is performed by using the class **WebAPITest**, which in turn uses the class **WebAPIAsync**, which contains a general implementation of a wrap­per around the web API method calls. 7. Feel free to explore the classes **WebAPITest** and **WebAPIAsync** in more detail, and use them to conduct further tests of the web service. |

|  |  |
| --- | --- |
| **Exercise** | PERSIST.6 |
| **Solution** | DBEFandWSHotel (contains two projects) |
| **Purpose** | Use the Entity Framework and a RESTful Web Service to enable a client to access data in a local 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, and expose the data by using a RESTful Web Service. |
| **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** (if you don’t have it already). 2. Run the given script file on the database. The data­base should now con­tain four tables **Booking**, **Guest**, **Hotel** and **Room**, and all tables should contain some sample data. 3. In the **DBEFandWSHotelServer** project, use the Entity Framework to create a class model corresponding to the **HotelDB** database. Make sure that the classes **Booking**, **Guest**, **Hotel** and **Room** are generated. 4. Still in the **DBEFandWSHotelServer** project, now create a controller for each of the classes mentioned above (one controller for each class), by fol­lowing the guidelines in the notes. Make sure that all controller classes are generated in the **Controllers** folder. 5. Once the controllers are generated, the web service is ready to run. You can test it by right-clicking on the **DBEFandWSHotelServer** project, and choose **View | View in browser**. You should see the standard ASP.Net start page. If you choose **API** in the menu, you should see the web service API for all of the classes. 6. Once the web service is up-and-running, you should be able to compile and run the **DBEFandWSHotelClient** project. Using a style similar to the style used in the previous exercise, add some code that tests the web service APIs. You should of course try to run the tests, and verify that the methods return the expected data. 7. Take a closer look at the content of the returned objects. It seems that the relations between objects are “broken”. Why do you think this happens? |

|  |  |
| --- | --- |
| **Exercise** | PERSIST.7 |
| **Solution** | AzureWebServiceA (contains two projects) |
| **Purpose** | Deploy a RESTful web service to Azure, and connect to it from a console application |
| **Description** | The solution contains the project **AzureCarRetailDBWebService**, which is a RESTful web service application (or rather; we will turn it into a web service). The project **ConsoleClient** contains code for a simple test of the web service. |
| **Steps** | 1. Go to your Azure account, and create a new database called **CarRetailDB**. Part of that process may also involve creating a new SQL Server, if you do not have one running already. 2. Open the database in Visual Studio (note that you may need to configure the firewall settings for your database server – see the notes), and run the script **CarRetailDBScript.txt** on it. This will create four tables **Car**, **Custo-mer**, **Employee** and **Sale**, and insert some sample data into the tables. 3. Go back to the **AzureCarRetailDBWebService** project, and turn it into a RESTful web service. That is: First perform the steps which connects the new database to the application using the Entity Framework (follow the steps described in the notes), and then add controllers for each table to the application (follow the steps described in the notes). 4. Now publish the web service to Azure. Again, do this by following the steps in the notes. Once the service is published, you can test it a bit by trying to call the API in a browser. 5. Open **Program.cs** in the **ConsoleClient** project. Change the value of the variable **serverURL** in **Main**, to refer to the URL of your own web service. 6. Run the **ConsoleClient** project (Note that you may be prompted by a virus checker to allow the application to make outgoing calls) . It contains a simple test of the web service: The data from all four tables is loaded and printed on the screen. Note that it may take several seconds before data is returned by the web service, so be a bit patient . |

|  |  |
| --- | --- |
| **Exercise** | PERSIST.8 |
| **Solutions** | MVVMStarterLibrary2018 and MVVMStarterDemoB (**NOTE**: These solutions are found as separate repositories on GitHub). |
| **Purpose** | See a fairly complete example of a UWP application using a RESTful web service as a data source |
| **Description** | The **MVVMStarterLibrary2018** solution contains a number of small, general-purpose class libraries for developing UWP applications based on the MVVM architecture. The **MVVMStarterDemoB** solution is an example of how the class libraries can be used, when using a web service as data source |
| **Steps** | 1. Download the **MVVMStarterLibrary2018** solution, and rebuild it. This may involve restoring some NuGet packages for the solution. Do not proceed until the solution can rebuild entirely. 2. Download the **MVVMStarterDemoB** solution, and try to rebuild it. This will probably result in several compilation errors. In the **Solution Explorer**, open the **References** node. For each “broken” reference, re-establish it by removing the reference, and browse to the .dll file on your own computer. All .dll files can be found in the folder **MVVMStarterLibrary2018DLLs** in the **MVVMStarterLibrary2018** solution. Do not proceed until all references are re-established, and the entire solution can compile without errors. 3. In the **MVVMStarterDemoB** solution, open the file **AppConfig.cs** in the folder **Configuration/App**. Change the variable **ServerURL** to refer to your own web service for the CarRetailDB on Azure. If you did not create such a web service, you can leave the URL as it is, and hope there is a web service at the other end . 4. Run the **MVVMStarterDemoB** application, choose **Load**, and see if the data from the Azure database is loaded, by opening the views related to each of the four data types (**Car**, **Customer**, **Employee** and **Sale**) 5. Explore the structure of the solution further. See if you can figure out the responsibilities for each class. Specifically, see if you can pin-point where the application is “locked” to using a web service as its data source. |

|  |  |
| --- | --- |
| **Exercise** | PERSIST.9 |
| **Solution** | DataSourceExample (contains two projects) |
| **Purpose** | Illustrate the benefits of using interface-based programming, using the problem of accessing different data source as an example |
| **Description** | This exercise focuses on the client side of a client-server setup, where the client needs to interact with the server in order to work with domain data. The exercise does not require that you do much pro­gramming, or even run the program   The **Client** project contains several files, organised in three folders.   * The **EFClasses** folder contains classes which were auto-generated by the Entity Framework, corresponding to the **MovieDB** database used in previous exercises (containing a **Movie** and **Studio** table). They are not as such important for this exercise. * The **ModelClasses** contains interfaces and classes defining a very simple framework for a domain data model. * The **DataSourceClasses** folder contains interfaces and classes for accessing data sources. |
| **Steps** | 1. Study the classes/interfaces in the **ModelClasses** folder. Make sure you understand the responsibilities for each class. 2. Study the classes/interfaces in the **DataSourceClasses** folder. Make sure you understand the responsibilities for each class. Why is it useful to de­fine an interface **IDataSourceAsync** for data sources? What is the interface used for? 3. Study the code in **Program.cs**. What happens in the **SelectSource** method? What parts of the client application as a whole knows the specific data source we are currently using? Do the **Model** and **Catalog** classes know? 4. Suppose we have to support a third way of accessing data, e.g. from a file. How would you implement that functionality in the client application? What needs to be added/changed? 5. Suppose we for some reason forbid to use interface (and base classes in general) in the client application. How would that impact the structure of the code? |

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. Note that you may need to install the **EntityFramework** NuGet package as part of this process. Right-click on the project in the **Solution Explorer**, choose **Manage NuGet Packages**, browse for **EntityFramework**, and install it. At the time of writing, the version to install was version 6.2 [↑](#footnote-ref-4)
5. In addition to classes from the .NET class library, the client will also make use of functionality from (yet another) NuGet package called **Microsoft.AspNet.WebAPI.Client**. If the package is not installed automatically, you may need to install it manually, in the usual fashion for installing these packages. [↑](#footnote-ref-5)