|  |
| --- |
|  |
| Object-Oriented Pro-gramming with C# |
| Database access with Entity Framework Core (EFCore) |

|  |
| --- |
| By Per Laursen  10-04-2023 |

[Introduction 2](#_Toc132039729)

[Exercises 3](#_Toc132039730)

[EFCore.0 3](#_Toc132039731)

[EFCore.1 5](#_Toc132039732)

[EFCore.2 8](#_Toc132039733)

[EFCore.3 10](#_Toc132039734)

[EFCore.4 12](#_Toc132039735)

# Introduction

This document contains an exercise set for database access via **Entity Framework Core**, aka **EFCore** (version 7). The exercises are based on console applications.

The overall progression of the exercises is as follows:

* **EFCore.0**: Minimal example of reading data from a single table.
* **EFCore.1**: Use *EFCore Power Tools* to create a database context and a single domain class.
* **EFCore.2**: Use **EFCore** and *EFCore Power Tools* to create and access two related tables. See **EFCore** manage object references.
* **EFCore.3**: Use **EFCore** to access two related tables. Use a “data service” abstraction layer.
* **EFCore.4**: Use **EFCore** to access three related tables. Use a “data service” abstraction layer.

All exercises are based on the *database-first* principle, i.e. the domain classes are generated based on a given set of table definitions. The Visual Studio extension *EFCore Power Tools* is used for generating domain classes.

All exercises rely on table definitions which all include a column **Id** of type **int**, which will always serve as the primary key. Furthermore, the **Id** column will always be de­cla­red with the IDENTITY(1,1) definition, i.e. new keys are chosen by the database.

# Exercises

|  |  |
| --- | --- |
| **Exercise** | EFCore.0 |
| **Project** | EFCBarDBv0 |
| **Purpose** | Use *EFCore* to access a single table in a local database |
| **Description** | The project contains:   * A class **DrinkFlat**, which is a simple modeling of a “drink”, defined as con­sisting of an alcoholic part and a non-alcoholic part. * A class **DrinkDBContext**, which can be thought of as an *in-memory* repre­sen­tation of a database (see below). * A file **DBScript.sql**, which contains an SQL script for generating the table *DrinkFlat* in the database (see below), and populating it with some data. * Code in **Program.cs** for reading data from the *DrinkFlat* table. |
| **Steps** | 1. Create a local database named **EFCDrinkDB** (in Visual Studio, open the **SQL Server Object Explorer** view, open **SQL Server\(localdb)\MSSQLLocalDB\ Data­bases**, right-click and choose **Add New Database**). It is important to choose the name as **EFCDrinkDB**, since the project relies on using a data­base with that name. 2. Once the **EFCDrinkDB** database has been created, run the SQL script found in **DBScript.sql** (right-click on the database, and choose **New Query**. Copy-paste the script to the query window and run it). This creates the table *DrinkFlat* in the database. You should check that the table has in­deed been created and populated before proceeding. 3. Open the **DrinkFlat** class. As mentioned above, the class itself is simple: six properties of simple types and a **ToString** method. There are, however, a couple of unusual elements in the class. First, the comment in the top line claims that this class has been *auto-generated by EF Core Power Tools*. This is true – even though some small modifications have been made after the generation – and we will work with *EF Core Power Tools* in the next exer­cise. Also note the “annotations” in square brackets above some of the properties; these are also auto-generated. Finally, also note that the class has been declared as **partial**. If you don’t know the meaning of a par­tial class, try to find information about it. Why could that be useful when we are working with auto-generated code? 4. Open the **DrinkDBContext** class. This class has also been auto-generated – and subsequently slightly modified – by *EF Core Power Tools*. For now, you can think of this class – which inherits from the **EFCore** base class **DbCon­text** – as a kind of *in-memory* representation of a database. The database is identified by the connection string in the **OnConfiguring** method (**NB**: If you for some reason need to use a different database, you need to update the database connection string accordingly), and the class will typically con­tain a property similar to the **Drinks** property for each table in the data­base (i.e. a property of type **DbSet**<(name of domain class)>). The **Drinks** property is thus an *in-memory* representation of the content of the *DrinkFlat* table (notice the **[Table("DrinkFlat")]** annotation of the **Drink­Flat** class). That is, we can use this property for data access and data alter­ation, and **EFCore** will then manage the all details about how to keep the database itself updated. 5. Now open **Program.cs**. In the first line of code (under ***// 1*** ***Create a con­text…***), the class **DrinkDBContext** is used for creating a database con­text. In step 2 in **Program.cs**, we read and print the data in the *DrinkFlat* table, simply by accessing the **Drinks** property on the context object. This is where *EFCore* shows its force w.r.t. making the client code very simple! For now, simply try to run the application, and see if the output printed on the screen is as ex­pected (see the comments in **Program.cs** for details). If you get any errors w.r.t. the connection to the database, you need to re-check if it has been set up properly (see steps 1 and 2 above). |

|  |  |
| --- | --- |
| **Exercise** | EFCore.1 |
| **Project** | EFCBarDBv1 |
| **Purpose** | Use *EFCore Power Tools* to create a database context and a single domain class. Use **EFCore** to access a single table in a local database. |
| **Description** | The project initially contains:   * A file **DBScript.sql**, which contains an SQL script for generating the table *DrinkFlat* in the database and populating it with some data (you have probably already done this in the previous exercise). * A class **Helpers**, with a static method **PrintList** for printing a collection of objects. * **Program.cs**, with several snippets of code in comments. |
| **Steps** | 1. Since we will now be using the *Entity Framework Core Power Tools* exten­sion, the first step is to download the extension from the Visual Studio Marketplace: <https://marketplace.visualstudio.com/items?itemName=ErikEJ.EFCorePowerTools> 2. Once you have downloaded the extension, simply install it by following the *on-screen* instructions. You need to close any running instances of Visual Studio first. 3. When the extension has been installed, open the **EFCBarDBv1** solution. Right-click on the **EFCBarDBv1** project (not the solution!). The context menu should now contain a new menu entry named *EF Core Power Tools*. In that menu, choose the menu item *Reverse Engineer*. 4. A window with the title **Choose Your Data Connection** should appear (also see the screenshots at the end of the exercise). Here you must add a reference to the **EFCDrinkDB** database (it’s probably called *(localdb)\MSSQLLocalDB.EFCDrinkDB*, but if you for some reason are not running on a local database, you will need to specify your specific database instead). Click **OK** to proceed. 5. A window with the title **Choose Your Database Objects** should appear (also see the screenshots). Under **Tables/dbo**, you should see a list of tables in the **EFC­DrinkDB** database (you pro­bably only have the **DrinkFlat** table). Check off the **DrinkFlat** table and click **OK**. 6. A window with the title **Choose Your Settings for Project EFCBarDBv1** should appear (also see the screenshots). Delete the pre-filled *Models* from the *Entity­Type path* text box, and check off the options (click **OK** afterwards to proceed)    1. *Pluralize or singularize…*    2. *Use DataAnnotation…*    3. *Include connection string…* 7. After clicking OK, the classes **DrinkFlat** and **EFCDrinkDBContext** should be auto-generated. Note that they are generated using the “old-fashioned” syntax with an explicit namespace definition surrounding the class definition. If you wish, you can simply remove the surrounding namespace definitions. 8. Note that the naming of classes and properties is a bit different than in the previ­ous exercise. The *DrinkFlat* table is mapped to a **DrinkFlat** class. Also, the collect­ion property for **DrinkFlat** objects in **EFCDrinkDBContext** is named **DrinkFlats**. For convenience, rename that property to **Drinks**. 9. Open **Program.cs** and uncomment the code for steps 0+1+2. Run the application. It will probably just print “DrinkFlat” a number of times. Why doesn’t it print each **DrinkFlat** object more “neatly” as in the previous exercise? What is missing (Hint: What method defines how an object is transformed into a string…)? 10. Assuming that you now get the expected output, we proceed to how to create new objects. In **Program.cs**, uncomment the code for steps 3 + 4 +5. Step 3 just creates two new **DrinkFlat** objects (note that we use the syntax where we assign values directly to properties, since the **DrinkFlat** class does not contain any con­structor. If you wish, you can add one yourself and use that instead, BUT remem­ber to obey the parameter naming rules for such constructors, as explained in the **EFCore** presentation!!). Step 4 is interesting, since we here see how to actually add new data to the data­base using **EFCore**. The **DbSet** class has an **Add** method, which we then simply call on the **Drinks** property. The new data is, however, not saved to the physical data­base until we call the **SaveChanges** method. Note that **SaveChanges** is called on the context object, not the collection property. This means that we can change data in multiple ways before actually “committing” it to the database. If you run the application now, you should see that the data is indeed added. When the ap­pli­­ca­tion has terminated, check that the data was actually saved to the **EFCDrink­DB** database as well. 11. Finally, we try to delete data as well. Before proceeding, make sure to delete the newly added data – i.e. the **DrinkFlat** objects just created – from the database, such that it only contains the original data. Now uncomment steps 6 + 7, and run the application. The printouts should confirm that the two new drinks were added to the database, and the deleted again (also check the database itself). Note the use of the method **Remove**, which is called on the collection property. |

|  |  |
| --- | --- |
| **Exercise** | EFCore.2 |
| **Project** | EFCBarDBv2 |
| **Purpose** | Use **EFCore** and *EFCore Power Tools* to create and access two related tables. See **EFCore** manage object references. |
| **Description** | The project initially contains:   * A file **DBScript.sql**, which contains an SQL script for generating the tables *Drink* and *Ingredient* in the database and populating it with some data. * A class **Helpers**, with a static method **PrintList** for printing a collection of objects. * A folder **ModelsCustom**, which contains partial definitions for the classes **Drink** and **Ingredient** (note that the content of the classes is initially com­mented out). * A (for now) empty folder **ModelsGenerated**. * **Program.cs**, with several snippets of code in comments.   As compared to the previous exercise, the setup is more complex here. The class **Drink** now contains references to **Ingredient**, such that a **Drink** object may refer to two **Ingredient** objects. However, the *Drink* table in the database refers to the *Ingredient* table through foreign keys, i.e. the ids for the *Ingredient* rows in quest­ion. These foreign keys need to be mapped to actual object references, which is exactly what **EFCore** can do for us. |
| **Steps** | 1. Run the SQL script found in **DBScript.sql** (right-click on the database, and choose **New Query**. Copy-paste the script to the query window, and run it). This creates the tables *Drink* and *Ingredient* in the database and popu­lates them with data. Your should check that the tables have in­deed been created and populated before proceeding. 2. Use *EFCore Power Tools* to create the class definitions for **Drink** and **Ingre­dient**. Write “ModelsGenerated” in the “EntityTypes path” text box, such that the generated files end up in the **ModelsGenerated** folder. 3. For all three generated classes: remove the name­space definition specified at the top of the file, such that the classes appear in the same style as the existing classes. 4. Uncomment the code in the two class definitions in the **Models­Custom** folder. Can you see the idea of having the class definition of the domain classes split across two files? Why is it actually a bit “risky” that we do any manual modifications of the auto-generated files? 5. In **Program.cs**, uncomment the steps 1) and 2a), and run the application. Does the output look correct (probably not…)? It looks like we are missing something… Accessing the **Drink** collection in this way does not cause the object references to be resolved. Try placing a breakpoint in the line that prints out **drinksNoInclude**. Run the application and drill down into the collection (you need to click on the *Results View* in order to see the actual collection). What can you see on the individual **Drink** objects? Are the **…Id** properties correctly set? Are the **…Part** object references set? 6. Now uncomment step 2b) and do a similar investigation. In the **drinks­With­Include** collection, are the object references set correctly? The use of the **Include**-statement apparently makes all the difference! 7. Given that we can now read **Drink** objects correctly, what remains is to try out creation and deletion of **Drink** objects as well. Go ahead and imple­ment these steps: (you can use the solution to the previous exercise as inspiration):    1. Create a new **Drink** named *Gin and Tonic* (3 cl. Gin, and 15 cl. Tonic), and add it to the **Drinks** property on the context object.    2. Create a new **Drink** named *Elefanta* (3 cl. Rum, and 20 cl. Fanta), and add it to the **Drinks** property on the context object.    3. Save the new data by calling **SaveChanges** on the context object.    4. Print out the content of the **Drinks** property again, to con­firm that the **Drinks** were added.    5. Delete the two **Drinks** you just created (remember to save the changes).    6. Print out the content of the **Drinks** property again, to con­firm that the **Drinks** were deleted. 8. How did you create the new **Drink** objects, w.r.t. the references to the two **Ingredient** objects? Did you set the properties **AlcoholicPart** and **NonAlco­holic­Part** to refer to **Ingredient** objects? This is actually not the preferred style when using **EFCore**. Instead, try to set the identifier properties **Alco­holicPartId** and **NonAlcoholicPartId** to their proper value, and try again. You should hopefully observe that **EFCore** will resolve the identifiers into proper object references. How could we make it more obvious that a client should not set object references directly, but only object identifiers (hint: what method is called when we create new objects)? 9. How did you find the **Ingredient** objects needed for the new drinks? It might be convenient to have a method which – given an ingredient name – can return the corresponding **Ingredient** object. Try to define and use such a method. Where should we define this method…? |

|  |  |
| --- | --- |
| **Exercise** | EFCore.3 |
| **Project** | EFCBarDBv3 |
| **Purpose** | Use **EFCore** to access two related tables.  Use a “data service” abstraction layer. |
| **Description** | The project initially contains elements corresponding to the completed version of the previous exercise, plus some new elements:   * A file **DBScript.sql**, which contains an SQL script for generating the tables *Drink* and *Ingredient* in the database and populating it with some data (you have probably already used this script in the previous exercise). * A class **Helpers**, with a static method **PrintList** for printing a collection of objects. * A folder **ModelsCustom**, which contains partial definitions for the classes **Drink** and **Ingredient** (note that the content of the classes is initially com­mented out). * A folder **ModelsGenerated,** which contains the code generated – and subsequently slightly modified – when using *EFCore Power Tools*. * A new folder **Services**, which has quite a bit of content (see below). * **Program.cs**, which is empty. |
| **Steps** | 1. The folder **Services** contains a first shot at implementing a “data service” layer using **EFCore** as the database access technology. Start out by explo­ring the **Interfaces** subfolder, in particular the **IDataService** interface. Does it make any assumptions about how the methods are implemented? What are the purposes of the other interfaces in the folder? 2. In the **EFCore** subfolder, explore the **EFCoreDataServiceBase** class. See if you can answer these questions:    1. What does the method **Set** do (in the **DbContext** base class)?    2. What is the role of the method **GetAllWithIncludes**? Why is it de­cla­red as **virtual**?    3. Why don’t we call **SaveChanges** in **Read** and **GetAll**? 3. Next, explore the type-specific data service classes (they are quite small), to see what responsibilities they have. Could we have placed these imple­men­tations anywhere else? 4. Given this new data service layer, your job is now to implement the same functionality as in step 7 in the previous exercise (if you find this to be too repetitive, you are welcome to try out the data service layer functionality in a different way):    1. Create a new **Drink** named *Gin and Tonic* (3 cl. Gin, and 15 cl. Tonic), and add it to the **Drinks** property on the context object.    2. Create a new **Drink** named *Elefanta* (3 cl. Rum, and 20 cl. Fanta), and add it to the **Drinks** property on the context object.    3. Save the new data by calling **SaveChanges** on the context object.    4. Print out the content of the **Drinks** property again, to con­firm that the **Drinks** were added.    5. Delete the two **Drinks** you just created (remember to save the changes).    6. Print out the content of the **Drinks** property again, to con­firm that the **Drinks** were deleted. 5. If you are up for the challenge, try to add one or more new classes to the application, and see what it takes to get it “up and running”. You will (at least) need to define new tables, use *EFCore Power Tools* to generate new classes, imple­ment new interfaces and classes in the data service layer, and try it out in **Program.cs**. 6. [Slightly harder] All the operations on the domain objects are now done through the interfaces **IDrinkDataService** and **IIngredientDataService**. They are currently implemented by the corresponding **EFCCore…DataSer­vice** classes, which in turn rely on the **EFCoreDataServiceBase** base class. In the base class, the implementations are obviously very dependent on the context class. Consider whether it would be possible to implement the two data service interfaces directly on the context class. Maybe you could create a new partial class definition for **EFCDrinkDBContext**, and see what it would take to directly implement **IDrinkDataService** and **IIngredient­Data­­Service**. This is slightly tricky, since both interfaces contain a **Delete** method (why is that method of particular interest?). See how far you get. 7. If you succeed with step 6, you should in principle be able to choose your implementation of the interfaces at the top of **Program.cs**, since the rest of the code in **Program.cs** should only rely on interfaces . |

|  |  |
| --- | --- |
| **Exercise** | EFCore.4 |
| **Project** | EFCBarDBv4 |
| **Purpose** | Use **EFCore** to access three related tables.  Use a “data service” abstraction layer. |
| **Description** | The project contains the same elements as **EFCBarDBv3**, but now a new, more complex domain class **Cocktail** is added. A **Cocktail** has a *many-to-many* relation with **Ingredient**: a cocktail can contain many ingredients (and for each ingredient in a specific cocktail, a specific amount of the ingredient will be present), and an ingredient can be part of many cocktails.  W.r.t. the database, this gives rise to creation of two new tables *Cocktail* and *Cock­tail­Ingredient*, the latter being the consequence of the *many-to-many* rela­tion­ship mentioned before. The script file **DBScript.sql** has been updated to contain definitions and data for these two new tables.  In addition to the **Cocktail** domain class itself, a new class **Cock­tail­Ingredient** has there­fore also been introduced, to model the amount of a specific ingredient in a specific cocktail. A **Cocktail** object therefore contains a collection of **Cock­tail­Ingredient** objects. |
| **Steps** | 1. First, make sure to run the script in the **DBScript.sql** file, which creates and populates four tables. If you have already created the tables *Drink* and *Ingredient*, you can selectively just create and populate the tables *Cocktail* and *CocktailIngredient*. 2. The overall structure of the **EFCBarDBv4** project is the same as for the **EFCBarDBv3** project, but several new classes have been added in the various folders. Take particular note of:    1. **Cocktail** and **CocktailIngredient** in the **ModelsGenerated** folder. Note the property **CocktailIngredients** in **Cocktail**, with the *Inverse­Property* annotation. Why has this property been generated? Why do you suppose it has this annotation?    2. **Cocktail** and **CocktailIngredient** in the **ModelsCustom** folder. What is the purpose of the constructors? Do they obey the parameter naming rules for such constructors, as explained in the **EFCore** presentation?    3. The **EFCoreCocktailDataService** class in the **Services/EFCore** folder. Can you explain what is going on in **GetAllWithIncludes**? If not, try hovering the mouse cursor over **Include** and **ThenInclude**, and see if the explanations make sense.    4. Why don’t we have a **CocktailIngredient** service?    5. The **EFCDrinkDBContext** class. How has it changed from the previ­ous exercise? 3. The introduction of the **Cocktail** class definitely makes our domain model more complex, but **EFCore** makes it relatively easy to manage this com­plex­ity. Experiment a bit in **Program.cs**, by creating one or more new **Cock­tail** objects, read them and perhaps delete them again. Also make sure to look into the database as well, to see what gets created/deleted when we create/delete a **Cocktail** object. 4. A notoriously tricky issue when having *many-to-many* relations is how to handle deletion. What should happen to instances of **CocktailIngredient** when the parent **Cocktail** is deleted? What about the parent **Ingredient**? At first attempt has been implemented in **EFCDrinkDBContext** (how?), but it might not be exactly what we need… Try to find more information about how to handle deletion and form your own opinion. 5. Could we still go the alternative route of letting **EFCDrinkDBContext** itself implement all of the type-specific data service interfaces (see steps 6 and 7 in the previous exercise)? If you did those steps in the previous exercise, then try to extend it to also include **ICocktailDataService**. It is probably not that hard, actually… 6. Finally, feel free to experiment with adding more tables/classes to the pro­ject, and/or perhaps add a **Setup** class that can encapsulate the details of how the implementation of the type-specific data service interfaces is cho­sen. Ideally, you can then switch between using the “**EFCDrinkDB­Context**-based” or the “**EFCoreDataServiceBas**e-based” strategy with just a single line of code. |