|  |
| --- |
|  |
| Object-Oriented Pro-gramming with C# |
| Database access with ADO.Net |

|  |
| --- |
| By Per Laursen  26-03-2023 |

[Introduction 2](#_Toc130743316)

[Exercises 3](#_Toc130743317)

[ADONet.0 3](#_Toc130743318)

[ADONet.1 5](#_Toc130743319)

[ADONet.2 6](#_Toc130743320)

[ADONet.3 8](#_Toc130743321)

[ADONet.4 9](#_Toc130743322)

[ADONet.5 11](#_Toc130743323)

# Introduction

This document contains an exercise set for database access via ADO.Net. The exercises are based on console applications.

The overall progression of the exercises is as follows:

* **ADONet.0**: Minimal example of reading data from a single table.
* **ADONet.1**: Reading, writing and deleting data from/to a single table. Dedicated class for database operations.
* **ADONet.2**: Reading, writing and deleting data from/to two related tables.
* **ADONet.3**: Reading, writing and deleting data from/to two related tables, using a service layer abstraction.
* **ADONet.4**: Reading, writing and deleting data from/to two related tables, using a service layer and a repository layer abstraction.
* **ADONet.5**: Reading, writing and deleting data from/to three related tables, using a service layer and a repository layer abstraction.

An general principle in these exercise is to let the application handle all “business rules”, like assigning new ids to new data instances, and to perform e.g. cascade deletion. The database is seen as a “passive” data storage unit, and all tables are defined without keys and constraints (except non-null constraints). This is not the same as stating that this is the most appropriate strategy in all cases, but more an attempt to keep focus on the data access logic in the applications.

# Exercises

|  |  |
| --- | --- |
| **Exercise** | ADONet.0 |
| **Project** | ADOBarDBv0 |
| **Purpose** | Use **ADO.Net** to access a single table in a local database |
| **Description** | The project contains:   * A class **Drink**, which is a simple modeling of a “drink”, defined as consisting of an alcoholic part and a non-alcoholic part. * 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.   **NB:** This project uses the package *Microsoft.Data.Sql­­Client*, so you need to install that package. The easiest way to do this is to open **Program.cs**, and hover the mouse cursor over the class name **SQLConnection**. This should prompt Visual Studio to pre­sent you with an option to install the package, as shown below.  Et billede, der indeholder tekst  Automatisk genereret beskrivelse  (exercises continues on next page) |

|  |  |
| --- | --- |
| **Steps** | 1. Create a local database named **DrinkDB** (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 **DrinkDB**, since the project uses that name in the DB connection string. 2. Once the **DrinkDB** 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 should create a table **DrinkFlat** in the database. The table is intentionally quite simple; six columns, and no key definitions. Your should check that the table has in­deed been created and populated before proceeding. 3. Open the **Drink** class. As mentioned above, the class itself is quite simple: six properties of simple types, a constructor and a **ToString**. No changes are needed for this class. 4. Now open **Program.cs**. In the first lines of code (under ***// 1) Setup DB***), the class **SqlConnect­ionStringBuilder** is used for creating the database con­nect­ion string, based on information about the name and location of the database. Once this information has been added, the **Connection­String** property will then contain the full connection string. 5. In step 2 in **Program.cs**, we try to read the data in the **DrinkFlat** table. In step 3, the data is the printed on the screen. 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). 6. Assuming you now get the expected output, take some time to study the code in step 2, in particular how the various **SQL…** classes are used (also note that if you hover the mouse cursor over the **ADONet** classes and method names, some quite detailed explanations will also show up). 7. Once you feel fairly confident about how these **SQL…** classes work, feel free to add a new table (say, a **Customer** class with some simple proper­ties) to **DrinkDB**, put some data into it, and try to write some code to access the new table. Use the given code as inspira­tion, but also feel free to try out different approaches. 8. The next step is to try to delete some data from a table. Try to write code for that purpose. It will look similar – a bit simpler, actually – to the given code, with two significant differences:    1. The query used when creating the **SqlCommand** object will – of course – be different.    2. Instead of calling the method **ExecuteReader** on the **SqlCommand** object, you will need to call the method **ExecuteNonQuery**. 9. [Slightly harder] See if you can figure out how to write new data to the data­base. Hint: Also use **ExecuteNonQuery** here. |

|  |  |
| --- | --- |
| **Exercise** | ADONet.1 |
| **Project** | ADOBarDBv1 |
| **Purpose** | Use **ADO.Net** to access a single table in a local database, using a dedi­cated class for database operations. |
| **Description** | The project contains:   * A class **Drink**, which is the same as in the previous exercise. * A class **DBMethodsForDrink**, which contains a set of methods used for **CRUD**-like operations for **Drink** objects. These methods use the **ADO.Net** class library (from the package *Microsoft.Data.Sql­Client*) to store and retrieve data from a local database. * A class **Helpers**, with a single static method for printing a **List** of **Drink** objects. * 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. |
| **Steps** | 1. Open **Program.cs**. In the first lines of code, we set up the database in the same way as in the previous exercise. In steps 2 to 6 in **Program.cs**, we now use the methods in the **DBMethodsForDrink** class (see below) to work with the data in the *DrinkFlat* table. First, try to run the application, and see if the output on the screen is as ex­pected (see the comments in **Program.cs** for details). 2. Assuming that you actually get the expected output, proceed to open the class **DBMethodsForDrink**. The main feature of the **DBMethodsForDrink** class is the three public methods **ReadAllFromDB**, **WriteToDB** and **Delete­FromDB**. Study all three methods in detail **NB**: This is the main purpose of the exercise, so take some time to fully understand the methods, including the purpose of the private helper methods. Consider the below questions:    1. All methods are very specific to the *DrinkFlat* table. What would it take to make the methods generally applicable to all tables in a database?    2. The **Drink** class has an **Id** property, which seems to be a sort of key. Who decides the **Id** value for a new **Drink** object? 3. Once you feel confident about how **DBMethodsForDrink** work, try to add a new table – or reuse a table you created in the previous exercise – to **DrinkDB**, put some data into it, and try to write some code to access the new table. Use the given code as inspira­tion, but also feel free to try out different approaches. Also consider if a base class for such **DBMethods** can be created in a meaningful way. |

|  |  |
| --- | --- |
| **Exercise** | ADONet.2 |
| **Project** | ADOBarDBv2 |
| **Purpose** | Use **ADO.Net** to access two related tables in a local database |
| **Description** | The project contains:   * The folder **Models**, with the (small) interface **IHasId**, and the two domain classes **Drink** and **Ingredient**. Note that **Drink** con­tains two properties that are references to **Ingredient** objects. * The folder **DBMethods**, which now contains a **DBMethodsBase** class, plus two derived classes specific for **Drink** and **Ingredient**. * A class **Helpers**, with a single, type-parameterized static method for print­ing a **List** of objects. * A file **DBScript.sql**, which contains an SQL script for generating the tables *Drink* and *Ingredient* in the database (see below) and populating them with some data.   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. This implies that we need to be able to map **Drink** objects to data which is “compa­tible” with the **Drink** table when writing new **Drink** objects to the data­base, and vice versa (map table-compatible data to **Drink** objects) when reading from the database. |
| **Steps** | 1. Start out by studying the **DBMethodsBase** class. It is a generalization of the **DBMethods** class from the previous project. Make sure you under­stand the mechanisms used for achieving this (type-parameterization and abstract methods). 2. Proceed to study the class **DBMethodsForIngredient**. Given the structure of the **Ingredient** class and the corresponding table, it should hopefully be fairly straightforward. 3. Next, proceed to the class **DBMethodsForDrink**. This class is more com­plex. Note that the **DBMethodsForDrink** constructor takes a parame­ter of type **DBMethodsForIngredient**. Why is that needed? What is it used for? 4. The **DBMethodsForIngredient** instance in **DBMethodsForDrink** helps to map ingredient identifiers to actual **Ingredient** objects when reading **Drink** data from the database. A “reverse” mapping – from **Ingredient** objects to ingredient identifiers – must then happen when writing a new **Drink** object to the database. Can you find the place in the code where that happens? Why is this a simpler operation? 5. We also need to extend the database. In the **DrinkDB** database – which you created in the previous exercises – now create and populate two new tables **Drink** and **Ingredient**, by running the script **DBScript.sql** on the data­­base. If you haven’t created the database yet, or are in doubt about this step, please refer to the previous exercises. 6. Code for building the database connection string is already present in **Program.cs**. Your job is now to continue the implementation in **Program.cs**, as outlined by the comments. That is, write code – which uses the two derived **DBMethods** classes and the **Helper** class – to:    1. Read all **Ingredient** data from the database, and print it.    2. Read all **Drink** data from the database, and print it.    3. Create a new **Drink** named *Gin and Tonic* (3 cl. Gin, and 15 cl. Tonic), and write it to the database.    4. Create a new **Drink** named *Elefanta* (3 cl. Rum, and 20 cl. Fanta), and write it to the database.    5. Read all **Drink** data (again) from the database and print it (to con­firm that the **Drinks** were added).    6. Delete the two **Drinks** you just created.    7. Read all **Drink** data (again) from the database, and print it (to con­firm that the **Drinks** were deleted). 7. Run the application, and see if the output matches your expectations. 8. The responsibility for mapping back and forth between the two **Drink** representations (object and relational) currently sits in the **DBMethods­ForDrink** class. Does that seem like the right place for this responsibility (this is not a trick question, it might actually be a suitable place 😊)?. If not, where could we place it instead? |

|  |  |
| --- | --- |
| **Exercise** | ADONet.3 |
| **Project** | ADOBarDBv3 |
| **Purpose** | Use **ADO.Net** to access two related tables in a local database. Use a “data service” abstraction layer. |
| **Description** | The project contains the same elements as **ADOBarDBv2**, plus an extra folder named **Services**. This folder contains a technology-neutral interface for a “data service” named **IDataService**. It also contains the class **DataServiceBase**, which pro­vides a generic implementation of **IDataService** by means of a **DBMethods** instan­ce. Finally, it contains two derived classes **DrinkDataService** and **Ingredient­Data­Service**. |
| **Steps** | 1. Start out by studying all the classes/interfaces in the **Services** folder. Make sure you understand the relation between the service classes and the exist­ing **DBMethods** classes. Where are the type-specific **DBMethods** classes used in the service classes? Why have we decided not to allow a caller of **GetAll** to include a *where*-clause as a parameter? Are there other ways in which the service layer “hides” the fact that we are actually using a database as the data source? 2. In **Program.cs**, you will find some code that implements the data access “requirements” from Step 6 in the previous exercise. Your job is to rewrite this code to use the service layer classes instead. The new implementation should – of course – produce the same results as before. 3. Reflect a bit over your revised implementation of the code in **Program.cs**. Do you find it to be signifi­cantly different from the original implementa­tion? What benefits could you gain from this extra layer w.r.t. testing, change of technology, etc..? 4. Feel free to experiment with adding new tables to the database, and using the available base classes to create new classes for handling access to the new tables. If you already created, say, a **Customer** class in the previous exercises, you can reuse it here, and see how much effort it takes to fit it into this framework of classes. |

|  |  |
| --- | --- |
| **Exercise** | ADONet.4 |
| **Project** | ADOBarDBv4 |
| **Purpose** | Use **ADO.Net** to access two related tables in a local database. Use a “repository” abstraction layer between the “data service” layer and the “DBMethods” layer. See the data resolution/desolution concept in practice. |
| **Description** | The project contains (almost) the same elements as **ADOBarDBv3**, plus an extra folder named **Repositories**. This folder contains a technology-neutral interface for a “data storage service” named **IRepository**. It also contains the class **Repository­Base**, which provides a generic implementation of **IRepository** by means of a **DBMethods** instance. Finally, it contains two derived classes **DrinkDataService** and **IngredientDataService**. The **DataService** classes now implement their func­tionality by means of repository instances.  In the **Models** folder, a new class **Drink­Desolved** has been added. The intention of this class is to have a representation of a **Drink** object which is closer to the repre­sentation of a drink in the database. More specifically, this version of a drink has all its object references “desolved” into (nullable) identifiers. It then becomes trivial to store (desolved) drink objects in the database through a generic reposi­tory imple­men­tation (see **RepositoryBase**). The price to pay is additional code to perform this object desolving and resolving on the boundaries between the Data Service layer and the Repository layer. The **Drink** class now implements a method **Desolve** (which in itself is simple), and the **DrinkDataService** correspondingly im­plements a **Resolve** method. The **Resolve** method is, however, somewhat harder to implement, since the **Drink­Data­­Service** needs to access both the repository for desolved drinks and the repo­sitory for ingredients (see **DrinkDataService**). |
| **Steps** | 1. The application has a rather deep structure now (perhaps too deep?). Start out by getting an overview of the layers in the application. Drawing up the layers on paper might be helpful. You can also consider these questions: Which layer is responsible for:    1. Actually reading and writing data to the relational database by use of SQL queries?    2. Choosing to use the **DBMethods** classes for database access?    3. Hiding the actual technology used for data access from the upper layers in the application?    4. Assigning Ids to new objects?    5. Resolving *object-to-object* references by identifiers into actual object references? 2. The **Drink** class implements the **Desolve** method, to convert itself into a **DrinkDesolved** object. Why doesn’t **DrinkDesolved** then implement the “inverse” method **Resolve**? What is it missing in order to be able to do so? 3. The **Setup** class is an attempt to encapsulate the details of setting up the various repositories and services in one place. How well does it also encap­sulate the actual technology used for data access? 4. When you feel you have a reasonable understanding of the application, you can try to add a new class (say, a **Snack** class) to the application. To keep things simple, maybe go for a class with only simple properties (e.g. a **Snack** class with a description and a price). See if you can implement all the elements (the class itself, DB table, DBMethods class, repository and data service) needed for managing data of the new type. The goal should be to add the new data service to the **Setup** class, and finally try it out in **Program.cs**. 5. If you are really ambitious, you could then try to add a class with object references (say, an **Order** class, where an order consists of a snack and a drink), and repeat all the implementation steps. Some of the steps will be a bit harder now. 6. No matter if you did steps 4 and 5, you should close off by reflecting a bit over the structure of the application. Does it make sense to you? Do some elements seem more confusing than helpful? If so, try to imagine taking that element away, and consider how you would replace it. If you feel confident, go ahead and give it a try! It’s really important to gain a sense of what a “properly” structured application looks like to you. |

|  |  |
| --- | --- |
| **Exercise** | ADONet.5 |
| **Project** | ADOBarDBv5 |
| **Purpose** | Use **ADO.Net** to access three related tables in a local database. See how the concepts introduced in the previous exercises play out when the complexity increases. |
| **Description** | The project contains the same elements as **ADOBarDBv4**, 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. Both tables are as such simple, but there is definitely some complexity hiding in the mapping from these two tables to actual **Cocktail** objects. 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 contains a **List** of **Cock­tail­Ingredient** objects.  W.r.t. repositories, these additions imply the need for two new repositories, named **CocktailDesolvedRepository** and **CocktailIngredientDesolvedRepository**. As the names indicate, they both manage data of “desolved” types, specifically **Cocktail­Desolved** and **CocktailIngredientDesolved** (these two new classes have been added to the **Models** folder). These new repository classes are however trivial as such, since they just use the base class implementation. Note that the new reposi­tories make use of to new DBMethods classes **DBMethodsForCocktail­Desolved** and **DBMethodsForCocktailIngredientDesolved**, respectively. Since the tables themselves are rather simple, these new classes are also fairly simple.  The complexity mostly hides in the new **CocktailDataService** class. This service needs access to three repositories (see the class definition), and the logic for the methods **Create**, **Delete** and **Resolve** is not trivial. In a sense, this can be conside­red to be the *object-relation mapping* (**ORM**) layer w.r.t. **Cocktail** objects. |

|  |  |
| --- | --- |
| **Steps** | 1. The total number of classes has increased somewhat from the previous exercise, but the overall structure of the application is the same. In some folders, a number of subfolders have been introduced to improve the file structure of the application. Start out by getting an overview of what has been added to the various folders. 2. Open the **Cocktail** class. Note the property **Ingredients**, which contains a collection of references to **CocktailIngredient** objects. Also take a look at the **CocktailIngredient** class itself. Note that is has two properties of type **Cocktail** and **Ingredient**, respectively. What does this tell us about the rela­tionship between cocktails and ingredients in general? 3. Open the **CocktailDataService** class; this is the most complex class in the Service layer. Note that it refers to three repositories. Why is that needed? Who will “inject” actual repositories into a **CocktailDataService** object? 4. Work your way through the implementations of the **IDataService** interface in **CocktailDataService**. Make sure you understand the use of the **Desolve** and **Resolve** methods. Why don’t we need any desolve/resolve in **Delete**? 5. The **Resolve** method itself is probably the hardest method to understand. Use whatever means you need – perhaps pen and paper – to gain a full under­standing of what it does. You may also need to peek into the **Add­Ingredient** method in the **Cocktail** class. Once you feel you have such an understanding, try to explain the method to another student. Keep dis­cussing and analyzing until you reach an agreement. 6. Proceed to the **Setup** class, and see what has been added as compared to the previous exercise. Why haven’t we implemented a **CocktailIngredient­Service**? Could such a service be useful? 7. In **Program.cs**, use the **Setup** object created at the top to implement code that performs these steps (also see the comments in the code):    1. Read all **Ingredients** from the DB, and print them.    2. Read all **Cocktails** from the DB, and print them.    3. Add two new **Cocktails** to the DB (feel free to compose some ima­gi­native cocktails, but be sure to use some ingredients that actually exist in the database…).    4. Read all **Cocktails** from the DB again, and print them.    5. Delete the two **Cocktails** just created.    6. Read all **Cocktails** from the DB again, and print them. 8. Feel free to experiment further with the application. Add new tables and associated classes. For the really ambitious, you could consider this to be a “back-end” to a Razor Page application, where you can define new ingredi­ents and cocktails, and show e.g. a menu of all cocktails available. Charge ahead and create such a Razor Page application 😊. |