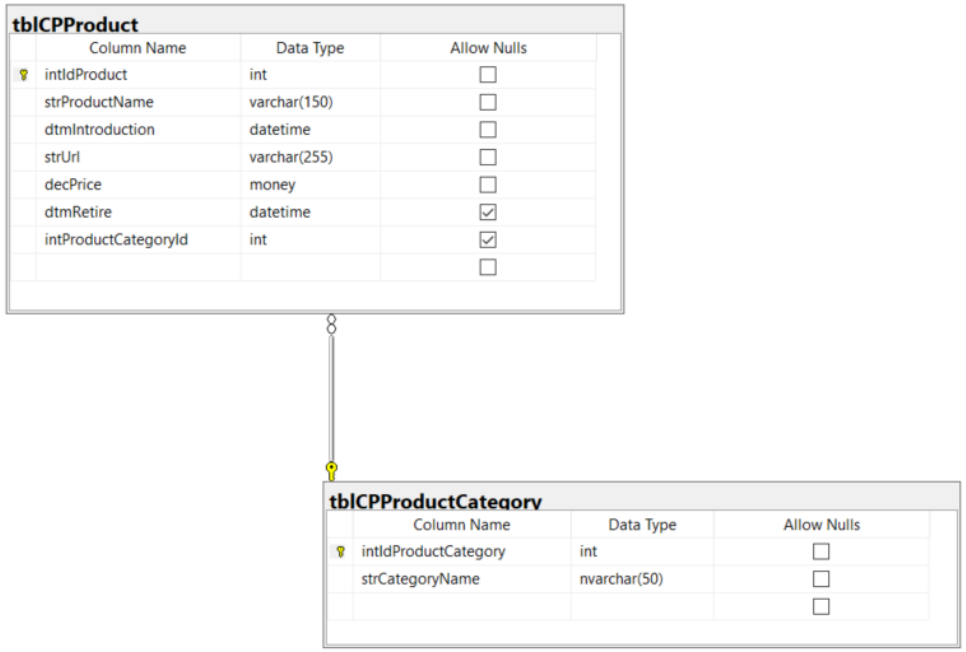

Figure 1: Database Diagram

Notes

Creating Mocked Data Service

Basic info

At the end, we will use the SQL Server data from database cpADONET.

The script to create this database can be found on GitHub and on Moodle.

https://github.com/DontDestroy/C-Sharp.NET-2019

- Look for Database Scripts.
- Look for cpADONET.sql.

This contains:

- 2 tables.
  - tblCPProduct.
  - tblCPProductCategory.
- 1 relation between the 2 tables.
- And a bit of data in both tables.
- The Database diagram can be created, but is also shown in picture beside. (*SEE FIGURE 1*)

Exercise 1: Creating a Mocked Data Service

The end goal is a working application, with a connection to the database.

We build up slowly.

The goal is to explain the process to getting there.

In this step we mock the data. We will use faked data to show some stuff.

Task 1: Make sure you have a backup (starting point of the project).

1. Use GitHub.

Task 2: Add fields and properties for a ProductCategory

In the Model "ProductCategory", we will add all the fields and properties that later will correspond with the fields in the table "tblCPProductCategory".

ProductCategory is based on ObservableObject and we will use the OnPropertyChanged Method when the value of field changes.

We want to create a 2-way binding towards the User Interface (UI), so the view where the ProductCategories are shown.

The product categories will be shown on the screen in the control under the Views Folder.

If the data changes in the database changes, the change is visible in the screen, if the screen information changes, the data is updated in the database.

1. Add a field for the unique identifier of a Product Category.
2. Add a field for the Product Category Name.
3. Add a property (Get and Set) the Product Category Key.
4. Add a property (Get and Set) for the Product Category Name.

Notes

Begin code (Example of a property)

```
public string Name
{

  get
  {
    return _strCategoryName;
  }

  set
  {
    OnPropertyChanged(ref _strCategoryName, value);
  }

}
```

End code

Task 3: Add fields and properties for a Product

In the Model "Product", we will add all the fields and properties that later will correspond with the fields in the table "tblCPProduct".

Product is based on ObservableObject and we will use the OnPropertyChanged Method when the value of field changes.

If the data changes in the database changes, the change is visible in the screen, if the screen information changes, the data is updated in the database.

1. Add a field for the unique identifier of a Product.
2. Add a field for the Product Name.
3. Add a field for the Introduction date.
4. Add a field for the Url.
5. Add a field for the Price.
6. Add a field for the Retire date (This can be null).
7. Add a field for the ProductCategory (This can be null).
   (Watch out: Don't use int, but the class ProductCategory)
8. Add all corresponding properties.

Notes

intIdProduc...	strCategory...
1	Development
2	Soft Skills
4	DevOps

intIdProduct	strProductName	dtmIntroduction	strUrl	decPrice	dtmRetire	intProductCategoryId
1	Extending Bootstrap with CSS, JavaScript and jQuery	2015-06-11 00:00:00.000	http://bit.ly/1SNzc0i	23,0000	NULL	1
2	Build your own Bootstrap Business Application Template in MVC	2015-01-29 00:00:00.000	http://bit.ly/1I8ZqZg	21,0000	NULL	1
3	Building Mobile Web Sites Using Web Forms, Bootstrap, and HTML5	2014-08-28 00:00:00.000	http://bit.ly/1J2dcrj	19,0000	NULL	1
4	How to Start and Run A Consulting Business	2013-09-12 00:00:00.000	http://bit.ly/1L8kOwd	9,9900	NULL	NULL
5	The Many Approaches to XML Processing in .NET Applications	2013-07-22 00:00:00.000	http://bit.ly/1DBfUqd	9,0000	2019-03-20 00:00:00.000	1
6	WPF for the Business Programmer	2009-06-12 00:00:00.000	http://bit.ly/1UF858z	29,0000	NULL	1
7	WPF for the Visual Basic Programmer - Part 1	2013-12-16 00:00:00.000	http://bit.ly/1uFxS7C	29,0000	NULL	1
8	WPF for the Visual Basic Programmer - Part 2	2014-02-18 00:00:00.000	http://bit.ly/1MjQ9NG	29,0000	NULL	1
10	Practical Team Management for Software Engineers	2017-05-19 00:00:00.000	http://bit.ly/2qcWO5m	15,0000	NULL	2
11	Leadership and Communication Skills for Software Engineers	2016-05-13 00:00:00.000	http://bit.ly/2aq2i4F	15,0000	NULL	2
12	Best Practices for Project Estimation	2014-12-08 00:00:00.000	http://bit.ly/1ulLVJK	15,0000	NULL	2
23	Using PowerShell	2019-05-28 12:40:03.200	www.fairwaytech.com	100,0000	2019-08-28 12:40:03.670	4

Figure 2: Data in the tables tblCPProduct and tblCPProductCategory.

Notes

COPY PASTE)

Task 4: Create a Mocked Service

In the project, we will create a folder “Service”. There we create a class “MockDataService”.

This class will be used to fake a database source, and later it will be replaced by a correct data service.

To ease the use, we will first create a class, and then based on that class, we will generate an interface that will be used as the base of that class.

The interface will also become the base of the correct data service.

1. Create a folder “Services” in the project.
2. Create a class “MockDataService”.
3. Create in the class “MockDataService” a list for products, use the data type IList<T>. Replace T with Product.

Why IList<T> and not List<T>.

List<T> is strongly typed, meaning, it must be a list.

IList<T> is an interface, so I can use all data types that inherits from IList<T>. Collection<T>, List<T> are examples.

4. Create in the class “MockDataService” a list for product categories, use the data type IList<T>. Replace T with ProductCategory.
5. Fill in the public constructor those lists with several products and productcategories. You can create constructors where needed.

In the screenshots beside (SEE Figure 2) you can see the data that is actually in the database.

Take a part of the data to create the mocked (faked) data.

In the script you can find the insert statements.

6. Create for every list you have also a method that returns the complete list. Choose your names carefully.

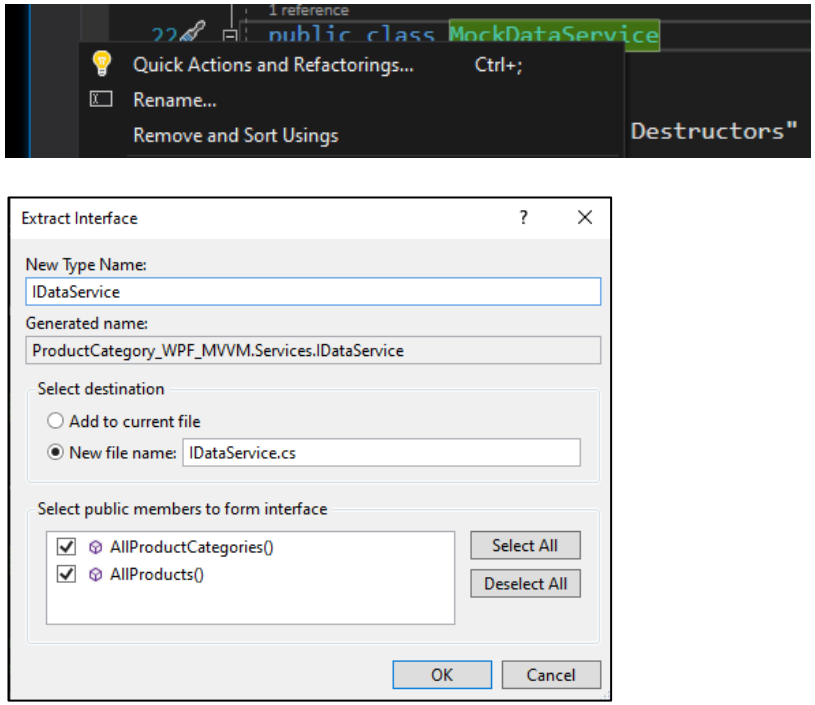

Figure 3: Create interface bases on an already created class.

Notes

.....

.....

.....

.....

.....

.....

.....

COPY PASTE)

7. Create an interface "IDataService".

Right click the classname "MockDataService" in the code (not in the solution explorer (SEE Figure 3).

Choose: "Quick Actions and Refactorings …

Choose: "Extract Interface …"

Change the name into "IDataService" and mark all the methods you want to define in the interface. Normally you have 2 methods. All the products and all the product categories.

It is a good practice to create a new file for newly created classes or interfaces.

8. Check if the class "MockDataService has now the created interface as parent.

Task 5: Start using the DataService (at this moment mocked data)

1. In the folder "ViewModels", you have the view model that you use to start up the application. In my example it is "StartupViewModel", but it is possible you have another name.
2. Open the code and create a field of the type IDataService. You probably must add also a reference to the "YourProject.Services" namespace.

We use the Interface data type, so that the field can contain all type that inherits from the type IdataService.

3. Create a constructor in that class and initialise the created field with a new instance of MockDataService. Later we will change this in the correct service, but now we are mocking the data.

Task 6: Use Data Binding to show the Product Categories

In the view where you show the Product Categories, we will remove the TextBlock and replace this with a ListBox.

The listbox will be binded with the content of the list of Product Categories.

- The source of the list will be the list of Product Categories.
- We show the names of the Product Categories.
- We select one item in the list.

First we create the needed elements, and then we change the xaml of the view.

1. Add "ObservableObject" as the base class for the ViewModel of the Product Categories. You will have to add a reference in this class.
2. Add three fields to the class. One that contains the DataService, another that will contain the list of Product Categories. Use the ObservableCollection<T> as data type. And the last field is a selected Product Category. Add the references when needed.

ObservableCollection<ProductCategory> is a datatype that allows a dynamic set of data.

When things are added, removed or the whole list is refreshed, you can trigger these events so you can refresh your view (screen) when needed.

3. Create a property for your ObservableCollection and make sure that the set use the OnPropertyChanged method. See the example of a property above.
4. Create a property for the Selected Product Category and make sure that the set use the OnPropertyChanged method. See the example of a property above.

Notes

5. Add a constructor that receives a parameter of the type IDataService, and that fills the list of Product Categories using the Method “AllProductCategories”.

You have a syntax like this. You probably have different variable names.

YourCollection = new ObservableCollection<YourDataType> (YourDataService.AllProductCategories())

6. Change the xaml of the product category screen / control by removing the TextBlock and adding a ListBox in the same location. Give the ListBox a good name. We will show Product Categories in it.
7. Add a reference in the xaml towards the ViewModels.
8. Property ItemSource of the ListBox will become the list of Product Categories like defined in ViewModel of Product Categories. Use the syntax {Binding YourPropertyName}.
9. Property SelectedItem of the ListBox will become the selected Product Category like defined in ViewModel of Product Categories. Use the syntax {Binding YourPropertyName}.
10. Property DisplayMemberPath of the ListBox will become the property that you want to show on the screen. In our case it will be the name of the Product Category.

```
<Grid>
  <ListBox ItemsSource="{Binding ProductCategoryCollection}" SelectedItem="{Binding SelectedProductCategory}" DisplayMemberPath="Name"/>
</Grid>
```

Notes

It is possible that you have different property names.

Take the correct ones.

- *List of the Product Categories.*
- *Which one is selected?*
- *What do you want to show?*

```
        Title="Products in their Category" Height="450" Width="800">
    <Window.Resources>
      <viewmodels:StartupViewModel x:Key="StartupViewModel"/>
    </Window.Resources>
    <Grid DataContext="{StaticResource StartupViewModel}">
      <views:ctrlProductCategory DataContext="{Binding ProductCategoryVWM}"/>
    </Grid>
</Window>
```

Figure 4: Example Xaml of the Startup Screen (Your variablenames can be different).

Notes

Task 7: Set DataContext in the two views

In the View Model of the Startup screen, we will adapt the constructor so that it creates a ProductCategoryViewModel.

Use a variable and the corresponding Property for doing this, and set the property in the constructor. We will also trigger the OnPropertyChange in the set, like in the previous tasks.

And then we set the data context property of the Grid an the view ProductCategory Control.

1. Add a field that represents a ProductCategoryViewModel in the startup view model.
2. Add a property that gets and sets the ProductCategoryView Model field. Add OnPropertyChange Method in the set.
3. In the Xaml of the startup screen, add a DataContext Property for the Grid. Use {StaticResource StartupViewModel} (*SEE FIGURE 4*).
4. In the Xaml of the startup screen, add a DataContext Property for the view used in the startup screen for the Product Categories. Use {Binding ProductCategoryVWM} (*SEE FIGURE 4*).
5. Rebuild your application.
6. Check if your application still runs. You should see the mocked information of Product Categories.

End result

The end result can be found in 00061-b ProductCategory WPF MVVM.zip, but not all step are followed to the letter. The example is just for checking, it is not to copy paste the solution.