|  |
| --- |
| **EASJ Notes** |
| Object-Oriented Pro-gramming with C# |
| Application Development, Part III |

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

[Introduction 2](#_Toc527374281)

[Overall application functionality and structure 3](#_Toc527374282)

[Menu-based navigation 3](#_Toc527374283)

[Exercises 8](#_Toc527374284)

# Introduction

The previous chapters have provided most of the tools needed to put together a “real” application, which in this context is defined as an application founded on the MVVM architecture, and using UWP (Universal Windows Platform) as the applica­tion framework, in particular with regards to creating the User Interface (UI). This chapter will provide an example of how to construct such an application.

The application will make use of the **MVVMGo** class libraries[[1]](#footnote-1), which provide various classes supporting an MVVM architecture. We will not go into details about the inner workings of the **MVVMGo** class library; see the class library documentation for fur­ther details on **MVVMGo** itself.

The application used for demonstration is the **CarRetailDemo** application[[2]](#footnote-2). Describing each and every detail in the demo will become too tedious – and many details may also be irrelevant for other applications – so we are only describing the main points of interest with regards to structure and functionality. Many of the shown code snip­pets will be abbreviated for clarity, and you will thus need to study the source code directly, if you wish to dig deeper into the functionality.

# Overall application functionality and structure

It is of course impossible to give general advice which is applicable to all kinds of appli­cations, so we limit the scope to fairly straightforward applications, where we are interested in being able to perform CRUD (Create, Read, Update and Delete) operations for a set of domain classes. The structure should be general enough to allow extensions with more specific functionality. The overall functionality of the application will thus be:

* Ability to perform general, menu-based navigation.
* Ability to perform CRUD operations on a set of classes
* Ability to save data to persistent storage.

## Menu-based navigation

The navigation to the main functionalities can be implemented by using a **Naviga­tion­View** control. This control will thus be the page element on the main page, i.e. defined in **MainPage.xaml**. The (abbreviated) XAML code looks like this:

**<NavigationView Header="Car Retail"**

**SelectedItem ="{Binding SelectedMenuItem, Mode=TwoWay}"**

**… (other options can be defined here)>**

**<NavigationView.MenuItems>**

**<NavigationViewItem Style="{…}" Icon="…" Content="File"**

**Tag="OpenFileView"/>**

**<NavigationViewItemSeparator/>**

**<NavigationViewItem Style="{…}" Icon="…" Content="Cars"**

**Tag="OpenCarsView"/>**

**<NavigationViewItem Style="{…}" Icon="…" Content="Customers"**

**Tag="OpenCustomersView"/>**

**<NavigationViewItem Style="{…}" Icon="…" Content="Employees"**

**Tag="OpenEmployeesView"/>**

**<NavigationViewItem Style="{…}" Icon="…" Content="Sales"**

**Tag="OpenSalesView"/>**

**</NavigationView.MenuItems>**

**<Frame x:Name="AppFrame">**

**<Frame.ContentTransitions>**

**<TransitionCollection>**

**<NavigationThemeTransition/>**

**</TransitionCollection>**

**</Frame.ContentTransitions>**

**</Frame>**

**</NavigationView>**

The most noteworth features of this control are:

* You can define several view options as attributes in the staring tag; we have only shown a few here.
* A set of **menu items** is defined as part of the control, in the form of several **NavigationViewItem** controls. In this example, the first item is intended to open a “File” view, while the rest will open a class-specific view, e.g. a Custom­ers view.
* The **Content** property contains the text shown in the application for a specific menu item. You can also e.g. specify a graphical icon in the **Icon** property.
* The **Tag** property is here used to contain a text describing the intended action. This is not a general prerequisite, but is used in this application for handling menu item selec­tion (see later).
* The navigation view also contains a **Frame** control, explicitly named **AppFrame**. The content of each view will be shown inside this **Frame** control.
* The data context for the page (i.e. the main page) is set to **AppViewModel**, which is a class we describe below.
* The **NavigationView** property **SelectedItem** is bound to the property **Selected­MenuItem**, which is defined in **AppViewModel**.

Running the application will produce this initial view, which is the **File** view.



The menu contains the menu items discussed earlier. In the bottom right corner, a number of view-specific **Button** controls are shown; we will discuss these controls in a moment.

The **NavigationView** control interacts with the rest of the application via data bind­ing, more specifically via the binding of the property **SelectedItem**. The class **App­View­Model** contains the “navigation logic” for the application:

**public class AppViewModel : AppViewModelBase**

**{**

**private NavigationViewItem \_selectedMenuItem;**

**public AppViewModel()**

**{**

**\_selectedMenuItem = null;**

**}**

**public NavigationViewItem SelectedMenuItem**

**{**

**get { return \_selectedMenuItem; }**

**set**

**{**

**\_selectedMenuItem = value;**

**if (\_selectedMenuItem == null) return;**

**string tag = \_selectedMenuItem.Tag.ToString();**

**if (!NavigationCommands.ContainsKey(tag))**

**{**

**throw new ArgumentException($"Menu entry {tag} has**

**no matching navigation command");**

**}**

**NavigationCommands[tag].Execute(null);**

**}**

**}**

**public override void AddCommands()**

**{**

**NavigationCommands.Add("OpenFileView", new RelayCommand(() =>**

**{**

**AppFrame.Navigate(typeof(FileView));**

**}));**

**NavigationCommands.Add("OpenCarView", new RelayCommand(() =>**

**{**

**AppFrame.Navigate(typeof(CarView));**

**}));**

**// Additional commands may follow here**

**}**

**}**

First of all, note the the class inherits from the class **AppViewModelBase** from the MVVMGo library. The main purpose of this class is to tie together a description of an action (e.g. “OpenCarView”), and a command which navigates to a specific view. The specific descriptions and commands must be defined in the derived class, which is done by overriding the **AddCommands** method. In this method, a number of descrip­tion-command pairs are added to the dictionary **NavigationCommands**, which is defi­ned in the base class. Note that the description string is intended to match the string used in the **Tag** property for each **NavigationViewItem** control. In this way, the string forms the link between a menu item and the corresponding view action.

The property **SelectedMenuItem** was the property which was bound to the **Selected­Item** property for the navigation view. So, whenever the user selects a new menu item, the **set**-part of the property will be executed. If the selected item is ***null***, no further action is taken. If a non-null selection has been made, the **Tag** property is extracted. Two situations may now occur:

1. **The extracted value can not be found in the navigation command dictionary**: In this case, an exception is thrown, since some sort of mismatch between menus and commands must have occurred.
2. **The extracted value is found in the navigation command dictionary**: In this case, the corresponding navigation command is executed by invoking **Execute** on the corresponding command object.

The code in the **set**-part of **SelectedMenuItem** is fairly generic, and it could make sense to place it in the base class. This has however not been done, in order to avoid tying the base class implementation to menu-based navigation.

As a final note on application-level navigation, it turns out that a minor trick is need­ed in order to properly establish the reference to the **Frame** control from the **App­View­Model** object. For this end, a small class called **AppConfig** has been created, which essentially contains a single, static method **Setup**:

**public static void Setup(Page mainPage, Frame appFrame)**

**{**

**appFrame.Navigate(typeof(FileView));**

**((AppViewModel)mainPage.DataContext).SetAppFrame(appFrame);**

**}**

In the code-behind for **MainPage.xaml** (found in **MainPage.xaml.cs**), the **MainPage** constructor now needs a small amendment:

**public MainPage()**

**{**

**this.InitializeComponent();**

**AppConfig.Setup(this, AppFrame);**

**}**

## Button-based navigation

Once we have navigated to a specific view, we also need to be able to invoke func­tionality which is specific for the view. This is done through use of the **CommandBar** control. A **CommandBar** control is essentially just a wrapper around a set of **Button** controls, more specifically a set of **AppBarButton** controls. In the **FileView** definition (found in **FileView.xaml**), a **CommandBar** control is defined (see below), and placed in the bottom part of the view:

**<Page.BottomAppBar>**

**<CommandBar Style="{…}">**

**<AppBarButton Icon="Stop" Label="Quit"**

**Command="{Binding NavigationCommands[Quit]}"/>**

**<AppBarButton Icon="Sync" Label="Load"**

**Command="{Binding NavigationCommands[Load]}"/>**

**<AppBarButton Icon="Save" Label="Save"**

**Command="{Binding NavigationCommands[Save]}"/>**

**</CommandBar>**

**</Page.BottomAppBar>**

Each **AppBarButton** control can invoke one specific functionality. The **Icon** and **Label** properties relate to the visual presentation of the button, and have no significance beyond that. As opposed to menu items, you can actually tie a specific command ob­ject to an **AppBarButton** control, by binding such an object (which must imple­ment the **ICommand** interface) to the **Command** property. The data context for the file view is the **FileViewModel** class, which inherits from **AppViewModelBase**, just as **AppViewModel** did. We can therefore bind the **Command** properties directly to the corresponding entries in the **NavigationCommands** dictionary.

The file view has the simplest setup with regards to view-specific functionalities. The views specific for each domain class – like e.g. the car view – all have the same gene­ral CRUD functionality, and the setup for invoking these functionalities is therefore made as general as possible, and it relies quite heavily on some of the classes from the MVVMGo library. In the car view, the **CommandBar** definition looks like this (it looks almost identical in the remaining class-specific views:

<CommandBar Style="{…}">

<CommandBar.Content>

<TextBlock Style="{…}" Text="{Binding ViewStateDescription}" />

</CommandBar.Content>

<AppBarButton Icon="Add" Label="Create"

IsEnabled="{Binding ControlStates[CreateControl].Enabled}"

Command="{Binding DataCommand[CreateCommand]}"/>

<AppBarButton Icon="Edit" Label="Update"

IsEnabled="{Binding ControlStates[UpdateControl].Enabled}"

Command="{Binding DataCommand[UpdateCommand]}"/>

<AppBarButton Icon="Delete" Label="Delete"

IsEnabled="{Binding ControlStates[DeleteControl].Enabled}"

Command="{Binding DataCommand[DeleteCommand]}"/>

<AppBarSeparator/>

<AppBarButton Icon="Setting" Label="View State">

<AppBarButton.Flyout>

<MenuFlyout Placement="Bottom">

<MenuFlyoutItem Text="Create"

Command="{Binding StateCommand[CreateStateCommand]}"/>

<MenuFlyoutItem Text="Read"

Command="{Binding StateCommand[ReadStateCommand]}"/>

<MenuFlyoutItem Text="Update"

Command="{Binding StateCommand[UpdateStateCommand]}"/>

<MenuFlyoutItem Text="Delete"

Command="{Binding StateCommand[DeleteStateCommand]}"/>

</MenuFlyout>

</AppBarButton.Flyout>

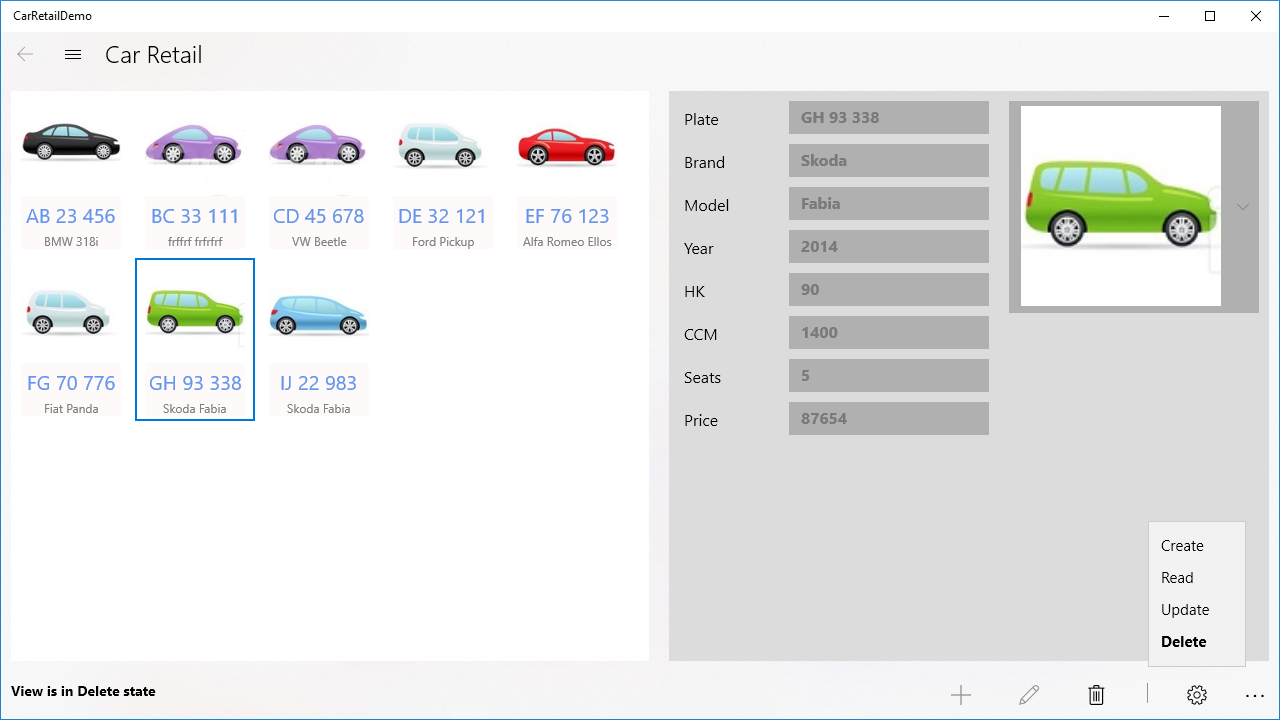
</AppBarButton>

</CommandBar>

The general idea for these views is that all CRUD operations can be performed within the same view. In order to enable this, a view can be set in a “state” corresponding to each of these four operations. This is the purpose of the four **MenuFlyoutItem** con­trols inside the last of the **AppBarButton** controls. When the user click on this control (which has the typical cogwheel icon for a settings-like functionality) a flyout menu appears with an entry for each for the four CRUD view states. Each **MenuFlyoutItem** con­trol is then bound to a command object which switches the state of the view. The current state of the view can also be seen in the view, through the **Content** property defined in the top of the **CommandBar** definition.

Finally, the **CommandBar** definition also contains three **AppBarButton** controls, one for each data-altering operation (i.e. Create, Update and Delete). Since it only makes sense to enable these controls when the view is in the corresponding state (other conditions may also apply), the state of each control with regards to being enabled or not is also managed through data binding. More specifically, the **IsEnabled** property is bound to a dictionary named **ControlStates**, which as mentioned is defined in the base classes in the MVVMGo class library (see later). This also goes for the binding of the **Command** properties.

Below is an example of how these controls manifest themselves in the application. In this case, the car view has been opened, the view has been set in the Delete state, and a specific car has been selected:



We observe that:

* The user can see that the view in the Delete state, from the text in the lower-left corner.
* The Delete button (with the trash can icon) is enabled, while the two other button are disabled.
* All property fields in the right-hand side have been disabled, since it does not make sense to be able to edit these in the Delete state (also see later).

These views do as mentioned rely heavily on base classes from the MVVMGo class library. For the car view, the data context is the **CarPageViewModel** class, which inherits from the base class **PageViewModelCRUD**. We will take a closer look at these view model classes in the next section.

## Application-specific view model classes

In addition to the application-oriented view model base classes (like the **AppView­ModelBase** class), the MVVMGo class library operates with two kinds of class-specific view model classes: Data view model and Page view model classes.

### Data view model classes

The Data view model classes are intended to be used for data binding at object level, meaning that whenever we need to present a single object (e.g. a single **Car** object) in a view, the data binding in the relevant controls is done to a Data view model class. This happens in two places in the class-specific views:

1. When a “compressed” version of the object is shown in a collection-oriented control like a **ListView** or **GridView**.
2. When details of a specific object are shown in the “details” part of the view, where the value of each property is shown in e.g. a **TextBox** control.

In the **CarRetailDemo** application, four Data view model classes are defined, one for each of the four domain classes. Each of these classes inherit from a class defined in the application, named **DataViewModelAppBase**. Note that this is not a class from the MVVMGo class library. The idea is that instead of letting each class-specific Data view model class inherit directly from a class from the class library, we add this extra layer of indirection, such that it is **DataViewModelAppBase** which inherits from a class from the class library. In this way, we only need to change the code in a single place, if we decide that all Data view model classes need to change in a certain way.

The **DataViewModelAppBase** class itself inherits from **DataViewModelWithSelect­ableImage** (from the class library), but this is as such not that important. It is more interesting to see the structure of a class-specific data view model class. Below fol­lows the code for the **CarDataViewModel** class:

**public class CarDataViewModel : DataViewModelAppBase<Car>**

**{**

**public CarDataViewModel(Car obj) : base(obj, "Car")**

**{**

**}**

**public string Plate**

**{**

**get { return DataObject.LicensePlate; }**

**set**

**{**

**DataObject.LicensePlate = value;**

**OnPropertyChanged();**

**}**

**}**

**// More properties with the same structure follow**

**public override string HeaderText**

**{**

**get { return Plate; }**

**}**

**public override string ContentText**

**{**

**get { return Brand + " " + Model; }**

**}**

**}**

Properties like the **Plate** property will be used for data binding for an individual control in the car view, like this:

**<StackPanel Orientation="Horizontal">**

**<TextBlock Style="{…}"**

**Text="{Binding ControlStates[Plate].Description}" />**

**<TextBox Style="{…}"**

**IsEnabled="{Binding ControlStates[Plate].Enabled}"**

**Visibility="{Binding ControlStates[Plate].VisibilityState}"**

**Text="{Binding ItemDetails.Plate, Mode=TwoWay}" />**

**</StackPanel>**

The type of the property **ItemDetails** will in this case be **CarDataViewModel**, so the **Plate** property will now be bound to the **Text** property of this specific **TextBox** con­trol. The remaining bindings are related to the visual presentation of the control, and are again a result of using functionality from the class library.

The properties **HeaderText** and **ContentText** are used in the item template definition for the collection control presenting all of the objects, in this case a **GridView** control (the property **ImageSource** has been defined in the **DataViewModelAppBase** class) :

**<GridView.ItemTemplate>**

**<DataTemplate>**

**<StackPanel Style="{…}">**

**<Image Style="{…}" Source="{Binding ImageSource}" />**

**<TextBlock Style="{…}" Text="{Binding HeaderText}" />**

**<TextBlock Style="{…}" Text="{Binding ContentText}" />**

**</StackPanel>**

**</DataTemplate>**

**</GridView.ItemTemplate>**

If you look back at the application screenshot a couple of pages ago, you can hope­fully see how these definitions result in what you actually see in the application. In the grid view, each car is presented with an image, the license plate in a large font (i.e. the “header text“) and the brand and model information in a smaller font (i.e. the “content text”.

### Page view model classes

The page view model classes are intended to act as data contexts for the class-speci­fic views. In the **CarRetailDemo** application, four Page view model classes are thus defi­ned, one for each of the four domain classes. Each of these classes inherit from a class defined in the application, named **PageViewModelAppBase**, with the same rea­son­ing as for the Data view model base class.

Taking a look inside e.g. the **CarPageViewModel** class reveals a quite small class defi­ni­tion:

**public class CarPageViewModel : PageViewModelAppBase<Car>**

**{**

**public CarPageViewModel()**

**: base(DomainModel.Catalogs.Cars,**

**new List<string> { "Plate", "Model", "Brand", "Year" },**

**new List<string> { "CCM", "HK", "Seats", "Price", "Image" })**

**{**

**}**

**public override IDataWrapper<Car> CreateDataViewModel(Car obj)**

**{**

**return new CarDataViewModel(obj);**

**}**

**}**

Comparing this class definition to other Page view model classes reveals two points in which the definitions differ:

1. The use of the name of the domain class in question (e.g. **Car**) is (obviously) different.
2. The two lists of strings in the contructor argument list.

The two lists of strings contain identifiers for controls used in the view for which this class becomes the data context. Recall this example of data binding from above:

**<TextBox Style="{…}"**

**IsEnabled="{Binding ControlStates[Plate].Enabled}"**

**Visibility="{Binding ControlStates[Plate].VisibilityState}"**

**Text="{Binding ItemDetails.Plate, Mode=TwoWay}" />**

The use of the dictionary key **Plate** here relates to these two lists. The two lists define which of the controls that are considered “immutable” ad “mutable”, respectively. By “immutable” is meant that this control contains a value which may not be changed after an object has been created. A “mutable” control on the other contains a value which may be changed after creation. When does this distinction come into play? When the view is set in the Update state. If you wish to update an object after it has been created, it will only be the fields in the latter of the two lists which will be open for editing. In this specific example, it hopefully makes good sense that you cannot change the plate, brand and model of a car, while you can indeed change e.g. the price and the image (it can be debated if it makes sense to be able to change e.g. the number of seats… ☺).

The ability to distinguish between immutable and mutable controls is not a manda­tory requirement for using these base classes. If you don’t wish to make this distinc­tion, you can simply leave out the data bindings to the **IsEnabled** and **Visibility** pro­perties, and enter two empty lists into the constructor argument list.

A final point of interest in the Page view model class definitions is the first argument to the constructor, like **DomainModel.Catalogs.Cars** in the example. This brings us to how to work with domain data in the application.

# Exercises

This document is not ready yet.

1. https://github.com/perl-easj/OOProg/tree/master/General/Libs/MVVMGo [↑](#footnote-ref-1)
2. https://github.com/perl-easj/OOProg/tree/master/General/Demos/CarRetailDemo [↑](#footnote-ref-2)