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

|  |
| --- |
| By Per Laursen  11-11-2018 |

[Introduction 2](#_Toc529439164)

[Overall application functionality and structure 3](#_Toc529439165)

[Menu-based navigation 3](#_Toc529439166)

[Button-based navigation 6](#_Toc529439167)

[Application-specific view model classes 9](#_Toc529439168)

[Data view model classes 9](#_Toc529439169)

[Page view model classes 11](#_Toc529439170)

[Working with Domain data 13](#_Toc529439171)

[Domain object collections 15](#_Toc529439172)

[Catalog base class 15](#_Toc529439173)

[The DomainModel class 17](#_Toc529439174)

[Exercises 19](#_Toc529439175)

# Introduction

The previous chapters have provided us with 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 for 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 starting tag; we have only shown a few options 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 in this application used 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 later in this chapter.
* 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**. This property is defined in the class which acts as data context for the page, i.e. the class **App­View­Model**. This is the (abbreviated) content of the **AppViewModel** class:

**public class AppViewModel : AppViewModelMenu**

**{**

**public override void AddCommands()**

**{**

**NavigationCommands.Add("OpenFileView",**

**new NavigationCommand(AppFrame, typeof(FileView)));**

**NavigationCommands.Add("OpenCarView",**

**new NavigationCommand(AppFrame, typeof(CarView)));**

**// Additional commands are added for Customer, Employee and Sale view**

**}**

**}**

Note that the class inherits from the class **AppViewModelMenu** 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. The command object is created by using the **NavigationCom­mand** class from the **MVVMGo** library. The property **SelectedMenuItem** - which was the property to which the **Selected­Item** property for the navigation view was bound – is also defined in the base class.

Note that the description strings are intended to match the strings used in the **Tag** properties for the **NavigationViewItem** controls. In this way, the strings form the link between menu items and corresponding view actions.

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 rest of the 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>**

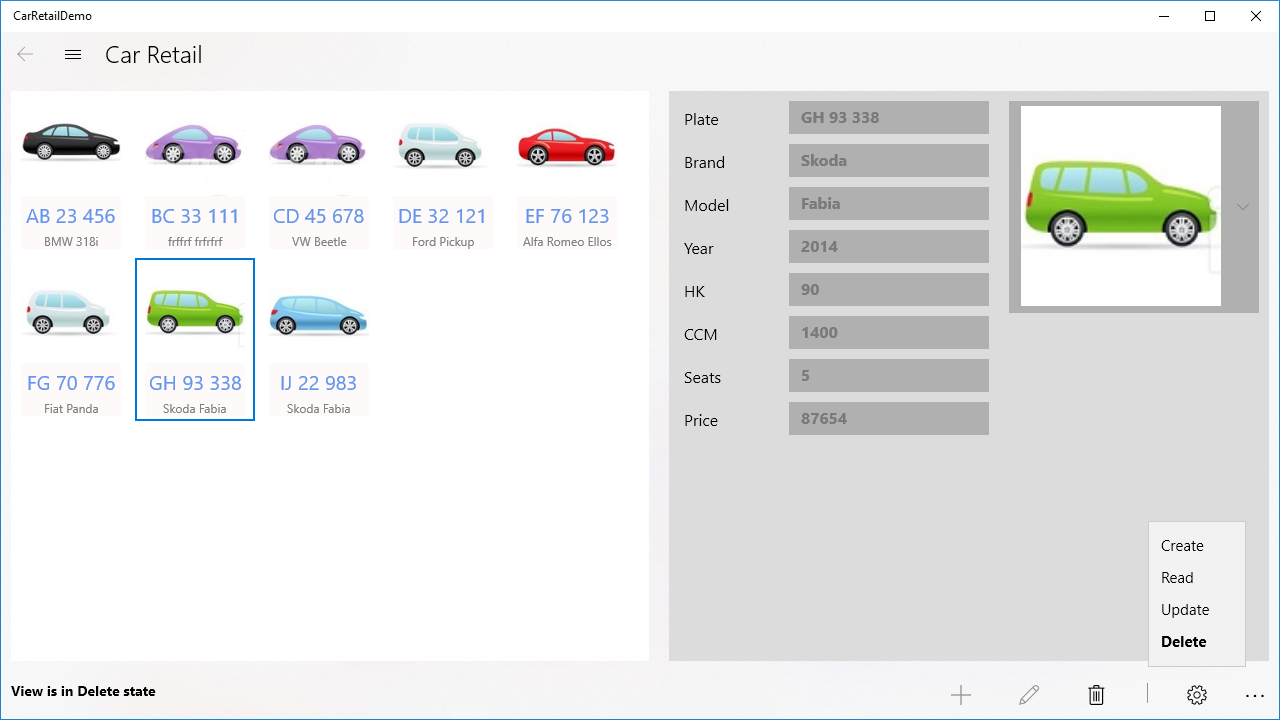
**</AppBarButton>**

**</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 clicks 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 buttons 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 the **DataViewModelAppBase** class which inherits from a class in the **MVVMGo** 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 as Plate follow**

**public override string HeaderText { get { return Plate; } }**

**public override string ContentText { get { return + $"{Brand} {Model}"; } }**

**}**

The first of these properties (i.e. the **Plate** property) has an typical structure for a property in a data view model class:

* It refers to another property **DataObject**, which is the domain data object (in this case of type **Car**) which was provided as an argument to the constructor. **DataObject** is defined in the **DataViewModelAppBase** class.
* The **get**-part simply returns the corresponding value in the domain data object.
* The **set**-part sets the corresponding value in the domain data object, but also calls **OnPropertyChanged**, to inform any other properties bound to this pro­perty that its value has changed.

The majority of properties in a data view model class will follow this pattern. How­ever, it is also perfectly valid to define “aggregated” properties, like e.g. the **Content­Text** property shown above.

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 **MVVMGo** 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.

## Working with Domain data

The bottom layer in the MVVM architecture is the Model layer, which will contain collections of domain model objects, and any business logic pertaining to the domain objects themselves or the domain model as a whole. The starting point will be the definition of a set of **domain classes**.

Domain classes are usually the result of a design activity preceeding the actual imple­mentation, and the implementation may in itself therefore by trivial. However, if the domain data is to be stored in a relational database, there may be some additional steps to perform. We have previously seen that if domain data is stored in a database – in which tables have been defined to store the data – we can use the Entity Frame­work to access the data. Also, we can use tools to assist us in generating the domain classes themselves, an example being the *EF Core Power Tools*. We assume in the following that this is indeed the scenario: a database has been created with table definitions corresponding to the domain model classes, but the classes have not been implemented in C# yet.

The first step is then to execute e.g. the EF Core Power Tools wizard, and generate the domain classes. In the **CarRetailDemo** application, four domain classes are gene­rated (**Car**, **Customer**, **Employee** and **Sale**). The generated classes are found in the folder **Data/Domain**. A **DBContext** class named **CarRetailDBAzureContext** is also generated, and found in the folder **Data/Persistent**.

The generated classes have a couple of noteworthy features:

* They are partial classes
* They only contain properties, corresponding to the columns in the correspon­ding table in the database.

This means that we can “supplement” the class definition with e.g. constructors, methods, base classes, etc.. In the demo, it has been chosen to create a file named e.g. **CarLogic.cs**, which contains the “supplement” to the **Car** class.

In order to understand this “supplement” to the domain classes, note that all domain classes inherit from the base class **DomainClassAppBase**, which is actually empty, but itself inherits from the **DomainClassBase** class from the **MVVMGo** class library. This is again to enforce a single-point-of-contact policy between the application classes and the library classes. This inheritance has two implications for the domain classes:

* They must override the method **SetDefaultValues**. This method is used to specify the “placeholder” values which are initially seen in a view, when it is requested to create e.g. a new **Car** instance.
* They must override the property **Key**.

The latter point deserves some explanation. The model classes in the **MVVMGo** class library require that model objects implement the property **Key**, which is used as a uni­que identifier for an object. However, specific domain classes might not imple­ment this property directly. In the demo, all four domain classes contain the property **Id**, since it has been decided to use **Id** as the column name for the primary key in all four tables in the database. Each class must therefore define how to “map” between the **Key** property and the specific unique identifier. For this reason, all four domain classes contain this code in the **…Logic.cs** file:

**public override int Key**

**{**

**get { return Id;}**

**set { Id = value; }**

**}**

This is a pretty simple mapping, and it seems like an obvious candidate for implemen­tation in the base class. However, this is hindered by the fact that the **Id** property is defined in the generated part of the class definitions. We could define an abstract property **Id** in the base class – and thereby also implement **Key** in the base class – but that would require adding the keyword **override** to **Id** in the generated code. There is no formal restriction on changing the generated code, but we consider it a somewhat fragile approach, and have therefore decided to implement **Key** in the domain classes themselves.

Introducing an additional property in the domain classes has a subtle consequence: When the Entity Framework tries to map an object to a row in a corresponding table, it will per default try to map all properties – be they simple or calculated (like **Key**) – to columns in the table. Since there is no **Key** column in the tables, an error will be reported. It is therefore necessary to add a bit of code to the model building code in the **ContextDB** class, which instructs the model builder to ignore this property. In the demo, the code looks like this:

**// Code from the OnModelCreating method in CarRetailDBAzureContext.**

**modelBuilder.Entity<Car>().Ignore(item => item.Key);**

**modelBuilder.Entity<Customer>().Ignore(item => item.Key);**

**modelBuilder.Entity<Employee>().Ignore(item => item.Key);**

**modelBuilder.Entity<Sale>().Ignore(item => item.Key);**

With this addition in place, we can now read data from the database tables and create corre­sponding domain objects.

### Domain object collections

The “unit” in which we store a set of domain objects of the same type has previously been named a **Catalog** class, and we follow that convention here. As before, we have created an application-specific base class for catalog classes named **CatalogAppBase** (see later). The type-specific catalog classes now become extremely simple, like this example shows:

**public class CarCatalog : CatalogAppBase<Car>**

**{**

**}**

As long as a catalog object simply acts as a container for a set of domain objects, it should indeed be possible to implement the entire “container logic” (like e.g. CRUD methods) in the base classes. Will all type-specific catalog classes then be empty? Not necessarily; if there is some sort of business logic which applies at this level in the do­main model, it would be suitable to implement it in the type-specific catalog classes. Since there is no such business logic in the **CarRetailDemo** project, all type-specific catalog classes are indeed empty.

### Catalog base class

We stated earlier that all type-specific catalog classes inherit from the **CatalogApp­Base** class. What does this base class look like? This will obviously depend on whet­her you want to implement a catalog base class yourself, or want to inherit from a base class from the **MVVMGo** class library. If you wish to implement your own base class, a good way forward is first to agree on a suitable interface for a catalog base class. Such an interface should probably as a minimum contain CRUD-like methods, and should probably also be type-parameterised. In the demo project, it has been cho­sen to inherit from an **MVVMGo** catalog base class.

The **MVVMGo** class library contains a number of catalog base classes, the main diffe­rence between them being the data source the cata­log uses for persistent data sto­rage. One of these catalog base classes is the **EFCoreCatalog** class. The **EFCoreCatalog** class implements the typical CRUD methods, a method for loading all data from the corresponding table, and a couple of additional properties. The class takes two type parameters:

* **T**: The type of domain objects to store (e.g. **Car**).
* **TDbContext**: The type of the **DBContext** class generated when running the EF Core Power Tools (in the demo, this is the **CarRetailDBAzureContext** class).

The **CatalogAppBase** class therefore looks like this:

**public class CatalogAppBase<T> : EFCoreCatalog<CarRetailDBAzureContext, T>**

**where T : class, IStorable, ICopyable, new()**

**{**

**public CatalogAppBase() : base(KeyManagementStrategyType.CollectionDecides)**

**{**

**}**

**}**

Again, a class which seems empty, but the type and constructor parameters is where the “configuration” of the catalog happens. Three features are noteworthy:

* **CatalogAppBase** itself has a single type parameter **T** (the type of the domain objects), while the second type parameter to **EFCoreCatalog** is set to **CarRetail­DBAzureContext**. This ensures that all type-specific catalogs will use the same database context class.
* The type parameter **T** is constrained. However, all domain classes which inherit from **DomainClassBase** will meet these constraint requirements.
* The base class constructor is called with the parameter **CollectionDecides**. This parameter informs the catalog about the chosen strategy for key management. In most modern databases, it is possible to define that the primary key should be auto-generated, which is often a perfectly fine choice. However, the catalog classes do not assume that somebody else handles assignment of unique keys, and therefore have their own key management system. Some­one (i.e. the cre­a­tor of a catalog object) must therefore decide who manages the key genera­tion process; the catalog or the database. In this case, we have decided that the catalogs handle key generation. It is also possible to choose that the data­base handles key generation.

With these choices in place, we ensure that all catalogs function in the same manner across the entire application. The only thing which varies from catalog to calalog is the type of objects stored in the catalogs. This is also the only choice left in the defi­nition of the type-specific catalogs, where the type parameter to **CatalogAppBase** is set.

### The DomainModel class

The entire domain data model will thus consist of a set of catalog objects, each con­tain­ing a set of domain objects of a specific type. The **DomainModel** class represents the entire domain data model, and therefore contains a property for each catalog. The catalog objects are created in the **DomainModel** constructor. The **DomainModel** class itself is a **Singleton**, which ensures that there will never be created more than a single **DomainModel** object.

The **DomainModel** class also contains two methods **LoadAsync** and **SaveAsync**. The intention is to make it possible to load/save the entire data model in response to a single user action, e.g. a click on a button control. The two methods will therefore simply call **LoadAsync** or **SaveAsync** on each catalog instance. Note that the calls are preceeded with the keyword **await**, meaning that it is possible to await the calls of **LoadAsync** and **SaveAsync**, if it is desired to keep the application responsive during load and save operations. In the demo application, a Windows 10-style waiting wheel is shown while load and save progresses.

Since the **DomainModel** class does not know the specific type of the catalogs, some care should be taken w.r.t. whether or not the **Save** operation makes sense or not to execute. For e.g. file-based persistency, a **Save** operation – which will save the entire data model to e.g. a text file – makes perfect sense, while it does not always make sense in a database scenario. Note that the **DomainModel** class does not inherit from any base classes, so you should see this implementation as an example of how to implement a class which represents the entire domain data model. The **Domain­Model** class also contains some elements used for being able to subscribe to events related to the **Load** and **Save** operations, which are e.g. used to manage the waiting wheel described above. These elements are by no means mandatory, so omit them if you do not see any need for them.

A final feature of the **DomainModel** class is a single, small piece of business logic, in the form of the method **CarsSoldByEmployee**. The logic is rather simple: given the key (i.e. a unique identifier) for an employee, find the number of cars sold by that employee:

**public int CarsSoldByEmployee(int employeeKey)**

**{**

**return Sales.All.FindAll(obj => obj.EmployeeKey == employeeKey).Count;**

**}**

The logic itself is here written as a LINQ query, but that is not in itself particularly important. A more interesting issue is if this is the correct place to implement such a piece of logic. You could argue that since we are performing a search through the **Sales** catalog, the logic should be implemented in that class. The correct choice will probably depends somewhat on other architectural choices, so we will not discuss the issue further. The main point is just to show that this could be a valid place to imple­ment business logic, since it is easy to use this logic elsewhere. Invoking the logic could look like this:

**DomainModel.Instance.CarsSoldByEmployee(anEmployee.Key);**

In the demo, this is utilized to implement a method **CarsSold** in the **Employee** class (in the **EmployeeLogic.cs** file):

**public int CarsSold()**

**{**

**return DomainModel.Instance.CarsSoldByEmployee(Key);**

**}**

As a final point, note that this logic has here been implemented as a method, even though it seems more appropriate to implement it as an aggregated property. Doing that would however also require that the property is added to the model building “ignore list” discussed earlier, since we should (obviously) not include a **CarsSold** column in the **Employee** table.

# Exercises

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)