ANKARA UNIVERSITY

ENGINEERING FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

INTERNSHIP REPORT

Login Register System with MVVM Pattern

Faik Görkem Atabay

16290076

17.08.2020-11.09.2020

ABSTRACT

The aim of this summer internship report is to provide an overview of MVVM pattern. We begin by defining key concepts, including Visual Studio, Windows Presentation Foundation, and MVVM pattern. We examine several differences between MVVM and MVC in terms of testability. Then, we cover project development according to the MVVM structure while explaining the fundamentals of MVVM pattern.

INSTITUTION INFORMATION

Name	: ASELSAN A.Ş.
Deparment (if it can be stated)	: HBT
Address	: METU Teknokent
Telephone	: +90 (312) 592 10 00
E-mail	: aselsan@hs02.kep.tr
Web Page (if exists)	: https://www.aselsan.com.tr

ASELSAN is a company of Turkish Armed Forces Foundation, established in 1975 in order to meet the communication needs of the Turkish Armed Forces by national means. ASELSAN is the largest defense electronics company of Turkey whose capability/product portfolio comprises communication and information technologies, radar and electronic warfare, electro-optics, avionics, unmanned systems, land, naval and weapon systems, air defence and missile systems, command and control systems, transportation, security, traffic, automation and medical systems.

TABLE OF CONTENTS

1. INTRODUCTION

During my summer internship, I got a chance to work in the department (Test ve Doğrulama Müdürlüğü) of ASELSAN. My department's main purpose is test automation that is used to automate repetitive tasks which are difficult to perform manually. Sometimes it can be difficult to test source code, depending on the structure of the code. So, to see if the MVVM pattern is better than the MVC pattern in terms of testability, I was told to develop a software using the MVVM pattern instead of the MVC pattern and also to learn the MVVM pattern.

The aim of this report is to introduce the MVVM pattern and explain the fundamentals of development with this design pattern over the MvvmLoginRegister project, as well as to present the softwares used during development, and show how these softwares support the MVVM pattern. Furthermore, this report covers the basics of the C# programming language used during development and demonstrate how the C# programming language supports this design pattern thanks to the powerful features of C#.

2. MVVM PATTERN AND SOFTWARES

2.1. Visual Studio

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft to develop computer programs. Visual Studio uses Microsoft software development platforms such as Windows API, Windows Forms, Windows Presentation Foundation (WPF), Windows Store, and Microsoft Silverlight. We will use Windows Presentation Foundation in the next sections.

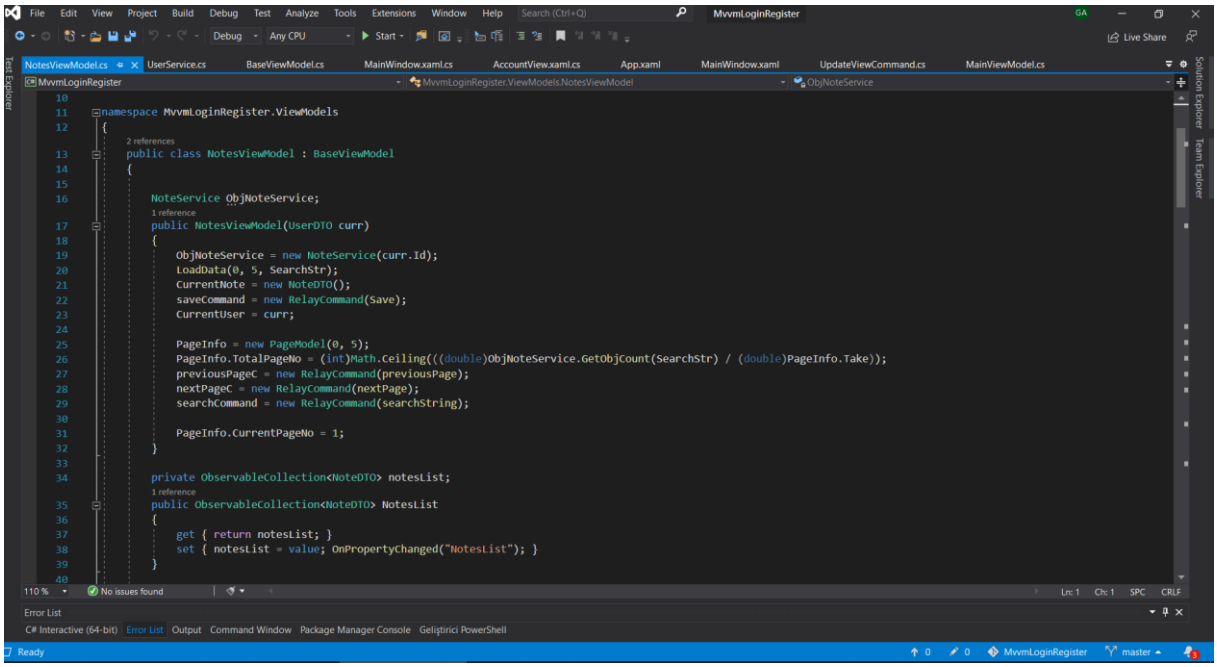

Figure 2.1. Example Visual Studio from the MvvmLoginRegister project

2.2. What Is Windows Presentation Foundation?

Windows Presentation Foundation (WPF) is a graphical framework, like WinForms, developed by Microsoft that renders user interfaces in Windows-based applications. WPF development platform supports a wide variety of application development features such as application model, controls, resources, layout, security, graphics, and data binding. The framework is part of the .NET framework. WPF uses the Extensible Application Markup Language (XAML) to provide a declarative model for application programming, and also WPF separates the user interface from business logic.

2.2.1. Data Binding

Data binding in Windows Presentation Foundation (WPF) provides a simple and consistent way for apps to present and interact with data. Elements can bound to data from a variety of data sources in the form of .NET objects and XAML.

Data binding is the process that establishes a connection between the app UI and the data it displays. When the data changes its value, the UI elements that are bound to the data reflect changes automatically if the binding has the correct settings and the data provides the proper notifications. For example, if the user edits the value in a TextBox element, the underlying data value is automatically updated to reflect that change.

WPF has a built-in set of data services to enable application developers to bind and manipulate data within applications. WPF supports four types of data binding:

- OneWay binding causes changes to the source property to automatically update the target property, but changes to the target property are not propagated back to the source property.

- TwoWay binding causes changes to either the source property or the target property to automatically update the other.

- OneWayToSource is the reverse of OneWay binding; it updates the source property when the target property changes.

- OneTime binding, which causes the source property to initialize the target property but does not propagate subsequent changes. If the data context changes or the object in the data context changes, the change is *not* reflected in the target property.

2.3. MVVM Pattern

Model, view, view-model (MVVM) is a software architectural pattern that facilitates the separation of the development of the graphical user interface (the view) from the development of the back-end logic or business logic (the model) so that the view is not dependent on any particular model platform. MVVM is a pattern that is used while dealing with views created primarily using WPF technology.

The well-ordered and perhaps the most reusable way to organize source code is to use the MVVM pattern. The MVVM pattern guides us on how to organize and structure our code to write maintainable, testable, and extensible applications.

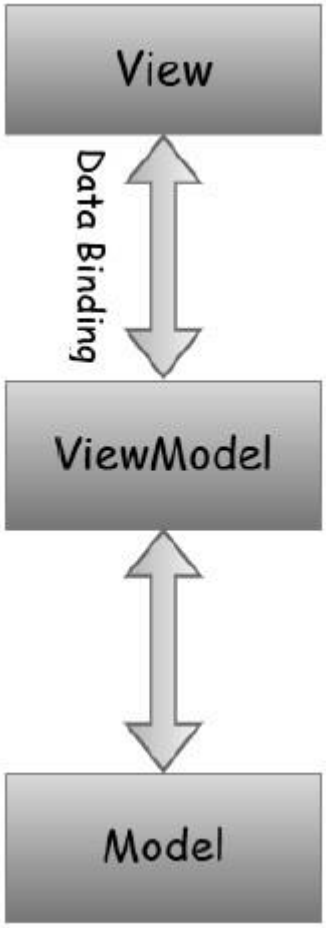

Figure 2.2. MVVM structure

The model simply holds the data and is not responsible for any of the business logic.

As in the model-view-controller (MVC) and model-view-presenter (MVP) patterns, the view is the structure, layout, and appearance of what a user sees on the screen. It displays a representation of the model and receives the user's interaction with the view and it forwards the handling of these to the view model via the data binding that is defined to link the view and the view model. It simply holds the formatted data and essentially delegates everything to the Model.

The view model is an abstraction of the view exposing public properties and commands. Instead of the controller of the MVC pattern or the presenter of the MVP pattern, MVVM has a binder, which automates communication between the view and its bound properties in the view model. It acts as the connection between the model and the view.

2.3.1. Why MVVM Is Better Than MVC in Terms of Testability?

MVC framework is an architectural pattern that separates an application into three main logical components model, view, and controller. On the other hand, MVVM makes separation of development of the graphical user interface easier with the help of mark-up language XAML. Because of that, the MVC Model component can be tested separately from the user while MVVM is easy for separate unit testing.

Table 2.1. Differences between MVVM and MVC

MVVM	MVC
The view is the entry point to the application.	Controller is the entry point to the application.
One to many relationships between View & View Model.	One to many relationships between Controller & View.
View have references to the view model.	View does not have reference to the controller.
MVVM is a relatively new model.	MVC is old model
The debugging process will be complicated when we have complex data bindings.	Difficult to read, change, to unit test, and reuse this pattern.
Easy for separate unit testing and code is event-driven.	MVC model component can be tested separately from the user.

3. LOGIN REGISTER SYSTEM

In this section we learn fundamentals of development with MVVM pattern and WPF technology through MvvmLoginRegister project. Moreover, we learn how to apply the MVVM pattern to our project and how to organize files.

3.1. Project Structure

First, we need to organize files as shown in figure 3.1. below to achieve MVVM structure.

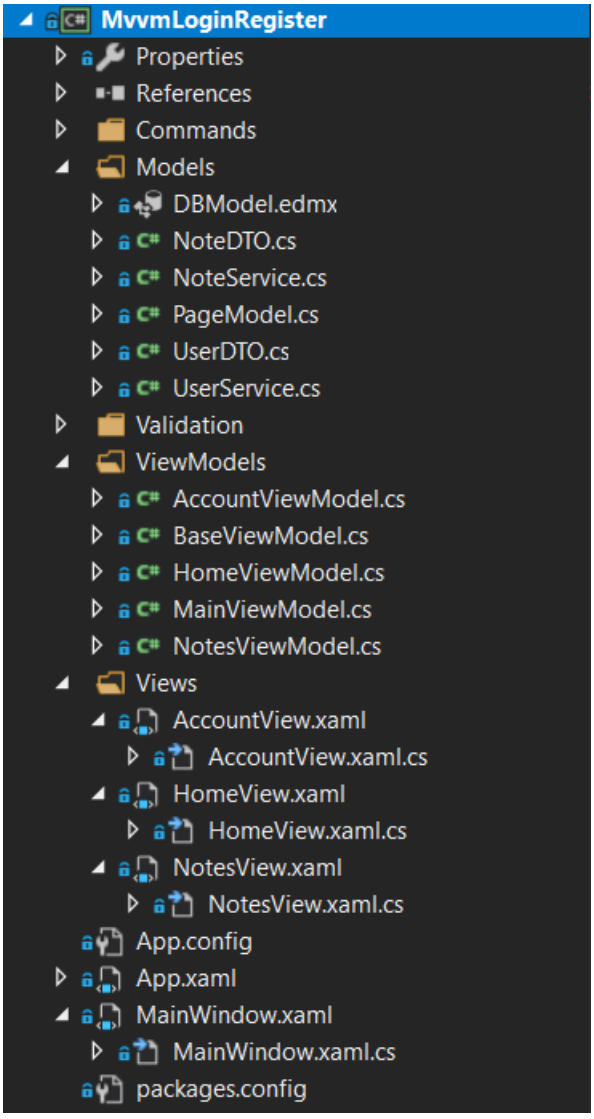

Figure 3.1. MvvmLoginRegister project folders and files

We have three essential folders as Views, ViewModels, Models according to the MVVM pattern. Views are simply GUIs. Inside the Views folder, we have our GUI that is constructed using XAML. We have one main window as MainWindow and three other GUI. AccountView and HomeView are user control type GUIs, NotesView is a window type GUI. User control is the concept of grouping markup and code into a reusable container, so that the same interface, with the same functionality, can be used in several different places.

In the Models folder, we have our business object, all the services that are required in order to perform operations.

In order to connect the models with the views and execute certain commands on clicking buttons, we have our view models.

We also have a folder as Commands that contain all the commands that must be executed when the user interacts with the menus and buttons. We have our commands code inside this folder. When the user click a button what needs to be done is inside this folder.

3.2. Hooking Up Views and ViewModels

For a connection between views and view models, we need to bind those together. There are several ways to achieve this. We will cover different ways in which we can get our views hooked up to view model.

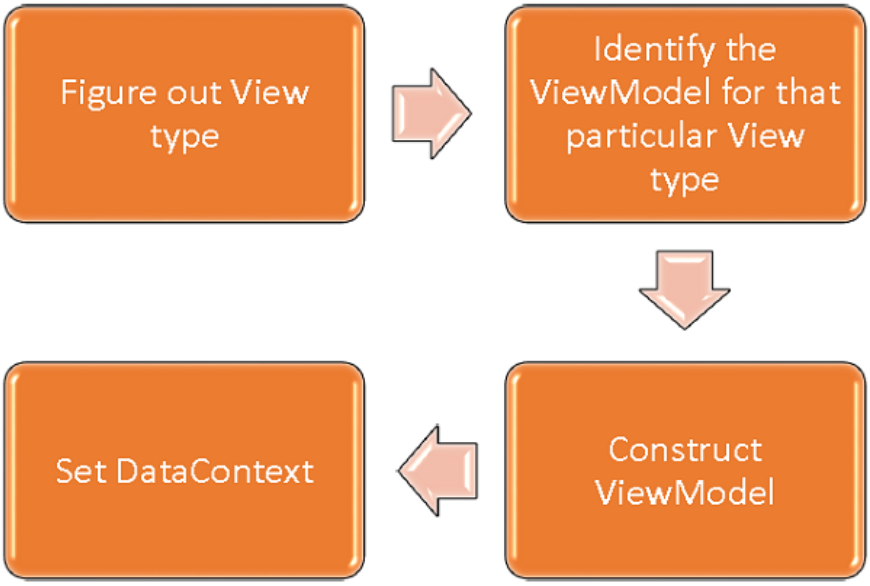

Figure 3.2. MVVM algorithm

According to the figure 3.2., we must first create a view. After creating the view, we must construct a view model for logical operations. Subsequently, we must set view's DataContext for communication between view and view model.

There are two ways to construct a view and bind view and view model. We can use any of them according to the our needs.

- View First Construction in XAML
- View First Construction in Code-behind

3.2.1. View First Construction in XAML

One way is to simply add our ViewModel as a nested element in the setter for the DataContext property as shown in figure 3.3. and figure 3.4.

```
<UserControl.DataContext>
    <viewmodels:HomeViewModel/>
</UserControl.DataContext>
```

Figure 3.3. HomeView.xaml

```
<UserControl.DataContext>
    <viewmodels:AccountViewModel/>
</UserControl.DataContext>
```

Figure 3.4. AccountView.xaml

However, for ease of use, we can use WPF's powerful feature data template. By using data template, we can define code in figure 3.3. and 3.4. in a code segment like in figure 3.5.

```
<Window.Resources>

    <DataTemplate DataType="{x:Type viewmodels:HomeViewModel}">
        <views:HomeView />
    </DataTemplate>

    <DataTemplate DataType="{x:Type viewmodels:AccountViewModel}">
        <views:AccountView />
    </DataTemplate>

</Window.Resources>
```

Figure 3.5. MainWindow.xaml

3.2.2. View First Construction in Code-Behind

Another way is that we can get view first construction by constructing the view model in the code behind of our view or a separate class by setting the DataContext property there with the instance.

Typically, the DataContext property is set in the constructor method of view. In our project, we use this approach to bind MainWindow and MainViewModel as shown in figure 3.6.

```
public partial class MainWindow : Window
{
    1 reference
    public MainWindow()
    {
        InitializeComponent();
        DataContext = new MainViewModel();
    }
}
```

Figure 3.6. MainWindow.xaml.cs

One reason for constructing the view model in code-behind or a separate class instead of XAML is that the view model constructor takes parameters, but XAML parsing can only construct elements if defined in default constructor.

```
NotesView win1 = new NotesView();
win1.DataContext = new NotesViewModel(viewModel2.TempUser);
win1.Show();
```

Figure 3.7. UpdateViewCommand.cs

For example, in figure 3.7. from the MvvmLoginRegister project, we create an instance NotesView which is a window. Then, we bind NotesView and NotesViewModel by defining DataContext. When creating instance NotesViewModel, we send a parameter that contains the current user's informations. We send the user's information to the new window that will open because if we do not do this, the user's information will be lost and will not be accesible. Another way to send the user's information to the new window is by creating a global variable, however, instead of defining global variables and making code tedious, we send the user's informations to the view model constructor as a parameter. In this way, we get a smoother and more effective code.

3.3. Data Binding

We will learn how data binding supports the MVVM pattern. Data binding is the key feature that differentiates MVVM from other UI separation patterns like MVC and MVP.

We need to have a view or set of UI elements constructed, and then we need some other object that the bindings are going to point to for data binding. The UI elements in a view are bound to the properties which are exposed by the view model. Since we have connected views with view models in the previous section, elements in the view like text boxes, buttons can access the properties in the linked view model.

```
<TextBox Text="{Binding CurrentNote.UserNotes, Mode=TwoWay}" HorizontalAlignment="Left"/>
<Button Content="Save" Command="{Binding SaveCommand}" HorizontalAlignment="Left" />
```

Figure 3.8. NotesView.xaml

If we look at the XAML code as shown in figure 3.8. above, we see that TextBox is bound to the CurrentNote property exposed by NotesViewModel. The TextBox is able to bind to the CurrentNote property because the overall DataContext for the NotesView is set to the NotesViewModel. We have covered this concept in section 3.2. You can also see that the property of SaveCommand has its own individual bindings as well, and this is bound to the button. But, we will cover this concept in the next section.

Bindings can either be OneWay or TwoWay data bindings to flow data back and forth between the view and view model. For example, in the MvvmLoginRegister project, user can save a note. When the user enters some characters in the text box, the text box updates the property which is bound to.

How do view models know when the textbox content has changed? At this point, the INotifyPropertyChanged interface is here to help us. INotifyPropertyChanged interface is used to notify the view or view model that which property is binding. In our project, we have BaseViewModel implementing the INotifyPropertyChanged interface, and all other view models extend BaseViewModel class, therefore, we can notify our view models when the value of any UI element changes.

3.4. Commands

In this section, we will learn how to add interactivity to our MVVM application. We will also see that all of this is done by good structuring which is the heart of the MVVM pattern.

The command pattern has been well documented and frequently uses design patterns for a long time. In this pattern, there are two main actors, the invoker and the receiver.

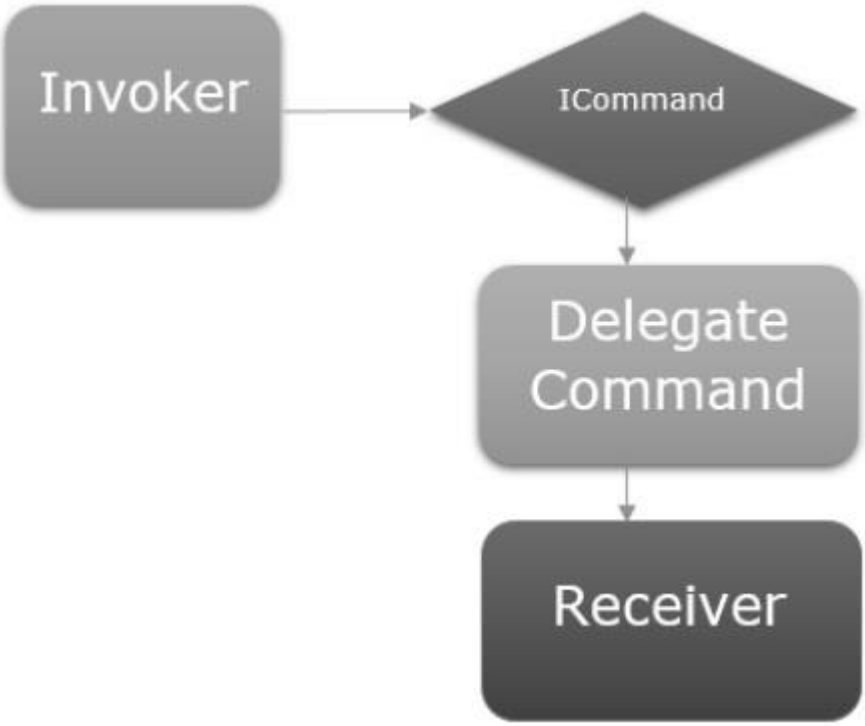

Figure 3.9. Command pattern

The invoker is a piece of code that can execute some imperative logic. Typically, it is a UI element that the user interacts with, in the context of a UI framework. In our project, buttons are invokers.

The receiver is the logic that is intended for execution when the invoker fires. In the context of MVVM, the receiver is typically a method in our view model that needs to be called. In our project UpdateViewCommand property in view models is our receiver.

Let’s have a look into our project in which we will learn the commands and how to use them to communicate between views and view models.

In MainWindow.xaml file, we have Login and Register buttons that allow the users to switch between views. The important thing is that working with commands on button is very easy because they have a command property to hook up to an ICommand. So, we can expose a property in our MainViewModel that has an ICommand and binds to it from the button’s command property as shown in figure 3.10.

```
<Button Margin="10"
        Width="200"
        Content="Log In"
        Command="{Binding UpdateViewCommand}"
        CommandParameter="Home"
        AutomationProperties.AutomationId="MainLog"/>
<Button Margin="10"
        Width="200"
        Content="Register"
        Command="{Binding UpdateViewCommand}"
        CommandParameter="Account"
        AutomationProperties.AutomationId="MainRegister"/>
```

Figure 3.10. MainWindow.xaml

We have classes that implement the ICommand interface in the Commands folder. Following is the default ICommand interface.

```
public class UpdateViewCommand : ICommand
{
    public event EventHandler CanExecuteChanged;

    1 reference
    public bool CanExecute(object parameter)
    {
        throw new NotImplementedException();
    }

    1 reference
    public void Execute(object parameter)
    {
        throw new NotImplementedException();
    }
}
```

Figure 3.11. ICommand interface

As you can see, this is a simple delegating implementation of ICommand where we have two delegates one for the Execute method and one for the CanExecute method which can be passed in on construction. We define what needs to be done when the user press any button in the Execute method.

```
public void Execute(object parameter)
{
    if (parameter.ToString() == "Home")
    {
        viewModel.SelectedViewModel = new HomeViewModel();
    }
    else if (parameter.ToString() == "Account")
    {
        viewModel.SelectedViewModel = new AccountViewModel();
    }
}
```

Figure 3.12. UpdateViewCommand.cs

We see the Execute method in figure 3.12. that is in the UpdateViewCommand class. Now we need to construct an instance of the UpdateViewCommand in the MainViewModel. After that, we can bind our buttons to this property.

Eventually, when a user clicks any of these buttons, the UI element fires an event and passed the UI element's CommandParameter to the UpdateViewCommand property in the MainViewModel. In the MainViewModel, this property takes the parameter then runs the Execute method and sets the SelectedViewModel to the appropriate view model according to the command parameter. Thus, the user can switch between views. In the next section, we will learn more about changing views.

3.5. Navigation Between Views

When building MVVM applications, we typically decompose complex screens of information into a set of parent and child views, where the child views are contained within the parent views in panels or container controls, and forms a hierarchy of use themselves. In our project, we can think HomeView and AccountView are child views and MainWindow is the parent view because HomeView and AccountView which are user controls are rendered in the main window frame, not in a new window or frame.

Our MainWindow hosts these views. We need a container control that we can place our views and switch them in a navigation fashion. For this purpose, we have ContentControl in our MainWindow.xaml file and we will be using its content property and bind that to a ViewModel reference.

We have defined data templates for each view as you remember. Figure 3.8. is the piece of code from MainWindow.xaml file. Note how each data template maps a data type (the ViewModel type) to a corresponding view.

```
<Window.Resources>

    <DataTemplate DataType="{x:Type viewmodels:HomeViewModel}">
        <views:HomeView />
    </DataTemplate>

    <DataTemplate DataType="{x:Type viewmodels:AccountViewModel}">
        <views:AccountView />
    </DataTemplate>

</Window.Resources>
```

Figure 3.13. MainWindow.xaml

```
<ContentControl  Content="{Binding SelectedViewModel}" Grid.ColumnSpan="2"/>
```

Figure 3.14. MainWindow.xaml

ContentControl holds a System.Object instance, therefore we can assign any value to this property. In this project, we want to assign ViewModel that is hooked up to the corresponding View so that view can be filled into the MainWindow frame. We have SelectedViewModel property in MainViewModel and some logic and commanding to be able to switch the current reference of ViewModel inside the property. So that, we can hold views in it. As the covered previous section, we have two buttons in the MainWindow. When the user clicks any of these buttons, SelectedViewModel is set to appropriate ViewModel. Anytime the SelectedViewModel is set to an instance of AccountViewModel or HomeViewModel, it will render out an AccountView or HomeView with the corresponding ViewModel is hooked up in the MainWindow's frame.

CONCLUSION

In this internship experience, I have studied MVVM design pattern and developed a project using this design pattern. When I encountered problems during the project work, I received support from engineers and they guided me during this internship period.

In this report we have started to learn softwares used in the project and how we can use these softwares according to our needs. We also looked at the benefits of the MVVM model over the MVC model.

In the second part of this report, we examined several topics central to the project development using MVVM pattern. We first examined the project structure. Then we examined the MVVM pattern features and how this pattern is used in the MvvmLoginRegister project. Finally, we combined the WPF and MVVM pattern features to see how easy, smooth and efficient to implement.

BIBLIOGRAPHY

İrgin, D. 2012. MVC, MVP ve MVVM Patternleri, https://denizirgin.com/mvc-mvp-ve-mvvm-patternleri-aa7d1011daff. Access Date: 19.10.2020.

Silay, R. 2020. WPF MVVM (Model-View-ViewModel) Mimarisine giriş, https://medium.com/@resulsilay/wpf-mvvm-model-view-viewmodel-mimarisine-giri%C5%9F-afe2c3879203. Access Date: 19.10.2020.

Wikipedia, 2020. Web site. https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel. Acess Date: 20.10.2020.

Koirola, S. 2014. WPF MVVM step by step (Basics to Advance Level), https://www.codeproject.com/Articles/819294/WPF-MVVM-step-by-step-Basics-to-Advance-Level. Acess Date: 21.10.2020.

Oneill, A. 2015. WPF: MVVM Step by Step 1, https://social.technet.microsoft.com/wiki/contents/articles/31915.wpf-mvvm-step-by-step-1.aspx. Acess Date: 21.10.2020

Tutorialspoint, 2020. Web site. https://www.tutorialspoint.com/mvvm/index.htm. Acess Date: 24.10.2020

Smith, J. 2016. Patterns - WPF Apps with The Model-View-ViewModel Design Pattern, https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern. Access Date: 24.10.2020