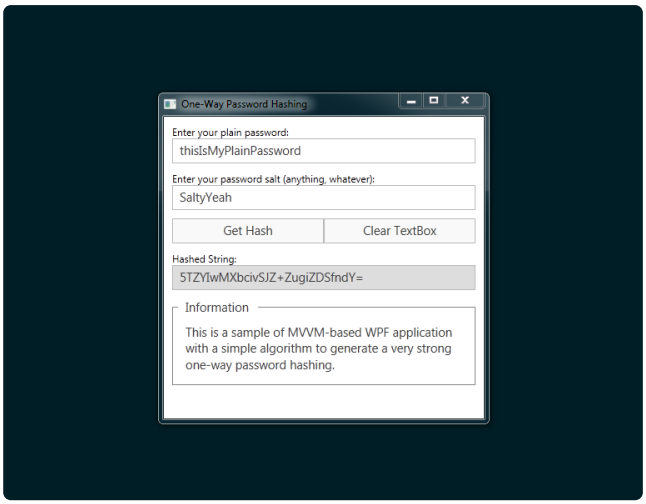
Heiswayi Nrird on Tech
Another Software Guy's Blog

SEPTEMBER 18, 2021 · 15 MINUTES · READ

Developing WPF Application Using MVVM Design Pattern



In this post, I'll try my best to explain about an implementation of MVVM design pattern in a WPF application development. This tutorial is based on my sample WPF MVVM application project known as **Password Hashing Tool**. The full source code for the project can be downloaded here. Let's begin!

tl;dr

Let's start with an introduction to WPF

Talking about software and programming, in the year 2006, I started to learn and developed my first Windows-based application. At that time, I used **VB6** - the most used programming language on the planet and then Microsoft killed it. My interest to programming continued as I learned a new high-level programming language in the year 2009 for developing Windows-based applications which is called **C#**. During that time, most of my applications had been built with C# using **WinForms** - a graphical (GUI) class library to write rich client applications.

However, as the technology started aging faster, the software requirements started to require more special features. At this time, WinForms wasn't able to catch up with the requirements due to its limitations in term of

features and performance, unless it was just enough for simple, small and basic applications. So, in the last two years, I started using **Windows Presentation Foundation (WPF)** - a graphical subsystem for rendering UI in Windows-based applications by Microsoft for software development.

Then I started to love WPF because it is now the current platform for developing Windows desktop applications. It's modern, advanced, hardware-accelerated framework for developing applications that maintain separation of concerns. It supports advanced rendering of 2D vector and 3D graphics, providing an immense range of capabilities for building rich, interactive and quality user interfaces.

Introduction to MVVM design pattern

The **Model View ViewModel (MVVM)** is an architectural pattern used in software development that originated from Microsoft, which is from the WPF creator itself. It's a variation of Martin Fowler's Presentation Model design pattern and it's based on a derivation of the Model-View-Controller (MVC) pattern.

WPF + XAML + MVVM

MVVM facilitates a separation of the development of the GUI from the development of the business logic or back-end logic (the data model). It's targeted at modern UI development platforms (WPF and Silverlight) in which there is a UX developer who has different requirements than a more "traditional" developer. With MVVM, it allows you to keep your code clean and easy to maintain. MVVM is a way of creating client applications that leverages core features of the WPF platform while allows for simple unit testing of application functionality and helps developers and designers work together with less technical difficulties.

- **Model** - A Model is a simple class object that hold data. It should only contain properties and property validation. It's **not responsible** for getting data, saving data, click events, complex calculations, business rules, or any of that stuff.

- **View** - A View is the UI used to display data defined in XAML and should not have any logic in the code-behind. In most cases, it can be DataTemplates which is simply a template that tells the application how to display a class. It binds to the ViewModel by only using data-binding. It's OK to put code behind your view **if** that code is related to the View only, such as setting focus or running animations.

- **ViewModel** - A ViewModel is where the magic happens. This is where the majority of your code-

behind goes such as data access, click events, complex calculations, business rules validation, etc. It's typically built to reflect the View. It's an abstraction of the View that exposes public properties and commands.

Why MVVM?

- **Collaboration** - Programmers and non-programmers (designers) can work together easily.
- **Reliability** - The code is testable (unit testing) to maintain the consistency of code quality.
- **Flexibility** - It's much easier to change view without messing with the rest of the code.
- **Code Friendly** - With a good separation between Model, UI and logic (code-behind), it makes easier for other people to understand overall process in order to take over the project, make improvements or debug it.

Let's have a look on "Password Hashing Tool" project

OK, here's how the Tree View for directories and files looked like inside the project.

```
Solution 'SimpleOneWayHashing' (1 project)
└── SimpleOneWayHashing
    ├── Properties
    ├── References
    ├── Commands
    |   ├── CommandReference.cs
    |   └── DelegateCommand.cs
    ├── Models
    |   └── HashingModel.cs
    ├── Themes
    |   └── Metro
    |       ├── Metro.MSControls.Core.Implicit.xaml
    |       ├── Metro.MSControls.Toolkit.Implicit.xaml
    |       ├── Styles.Shared.xaml
    |       ├── Styles.WPF.xaml
    |       └── Theme.Colors.xaml
    ├── ViewModels
    |   ├── HashingViewModel.cs
    |   └── ViewModelBase.cs
    ├── Views
    |   └── HasherView.xaml
    |       └── HasherView.xaml.cs
    ├── App.config
    ├── App.xaml
    |   └── App.xaml.cs
    └── ClassDiagram1.cd
```

> **NOTE:** I used Visual Studio 2013 to develop the "Password Hashing Tool" project but the solution and project files are named as `SimpleOneWayHashing`.

As can be seen in the Tree View above, there are primary folders called **Models**, **ViewModels**, and **Views**. This is how we basically separate between the GUI and the logics. There are a lot of online resources for MVVM, each with their own way of implementing the design pattern. As the development grows, we will have more folders. This is to ensure our project looks organized and clean.

I created another folder called **Commands**. Inside this

folder, there are some helper classes related to the use of `ICommand`. Let's skip this one first.

Let's start with the Model first

```
├── Models
│   └── HashingModel.cs
```

This is how the code looked like inside `HashingModel.cs` file.

```
using System;
using System.Collections.Generic;
using System.Security.Cryptography;
using System.Text;
using System.Threading.Tasks;

namespace SimpleOneWayHashing.Models
{
    public class HashingModel
    {
        #region Private Members
        private string appTitle = "One-Way Password Hashir
        private string result;
        #endregion

        #region Constructor
        public HashingModel(string _plainText, string _sa
        {
            PlainText = _plainText;
            Salt = _salt;
            result = string.Empty;
        }
        public HashingModel()
        {
            PlainText = string.Empty;
            Salt = string.Empty;
            result = string.Empty;
        }
        #endregion

        #region Public Properties and Methods
```

The class contains private members, public properties and methods. As you can see, `ComputingResult()` is a method I used to compute a hash from `PlainText` and `Salt` properties, then return the result to `result` property. This class uses its own **namespace** called `SimpleOneWayHashing.Models`.

The ViewModel

```
├── ViewModels
│   ├── HashingViewModel.cs
|   └── ViewModelBase.cs
```

Inside **ViewModels** folder, there are two files; `HashingViewMode.cs` and `ViewModelBase.cs`.

Inside `ViewModelBase.cs`, there is a helper class which inherits `INotifyPropertyChanged` interface.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Text;
using System.Threading.Tasks;

namespace SimpleOneWayHashing.ViewModels
{
    /// <summary>
    /// Provides common functionality for ViewModel class
    /// </summary>
    public abstract class ViewModelBase : INotifyPropertyCha
    {
        public event PropertyChangedEventHandler PropertyCha

        protected void OnPropertyChanged(string propertyName
        {
            PropertyChangedEventHandler handler = PropertyCh

            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(p
            }
        }

    }
}
```

`HashingViewMode.cs` is where it contains all the logics for this “Password Hashing Tool” program to work with the View.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Input;

using SimpleOneWayHashing.Commands;
using SimpleOneWayHashing.Models;

namespace SimpleOneWayHashing.ViewModels
{
    public class HashingViewModel : ViewModelBase
    {
        #region Private Members
        private Models.HashingModel hashing;

        private DelegateCommand computeHashCommand;
        private DelegateCommand clearTextBoxCommand;

        private string hashingTitle;
        private string computedHash;
        #endregion

        #region Constructor
        public HashingViewModel()
        {
            this.hashing = new HashingModel();
            hashingTitle = hashing.AppTitle;
            this.PlainText = string.Empty;
            this.Salt = string.Empty;
            this.computedHash = string.Empty;
```

`ViewModelBase.cs` and `HashingViewModel.cs` are using the same namespace `SimpleOneWayHashing.ViewModels`. As you can see from the source code above, there is a class called `HashingViewModel` which inherits from `ViewModelBase` class. Since they both are using the same namespace, so I don’t need to include it via `using`, but not for the Model and `ICommand` interface helper class.

The Model and `ICommand` interface helper class are using different classes and namespaces. So, I need to include them like this in `HashingViewModel.cs`:

```
using SimpleOneWayHashing.Commands;
using SimpleOneWayHashing.Models;
```

Overall, this is the code structure inside `HashingViewModel.cs`:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Input;

using SimpleOneWayHashing.Commands;
using SimpleOneWayHashing.Models;

namespace SimpleOneWayHashing.ViewModels
{
    public class HashingViewModel : ViewModelBase
    {
        // Private Members

        // Constructor

        // Public Properties

        // Commands
    }
}
```

Next is the View

```
    ├── Views
    │   └── HasherView.xaml
    |       └── HasherView.xaml.cs
```

For this project, I just have one View file only `HasherView.xaml`. This is usually our **MainWindow.xaml**. OK, I don't use `DataTemplate` here. Here's how the XAML code looked like inside `HasherView.xaml`:

```
<Window x:Class="SimpleOneWayHashing.Views.HasherView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xam
        xmlns:local="clr-namespace:SimpleOneWayHashing.ViewM
        WindowStartupLocation="CenterScreen"
        Title="{Binding HashingTitle}" Height="400" Width="4
    <Grid>
        <StackPanel Orientation="Vertical" Margin="10">
            <TextBlock Text="Enter your plain password:"/>
            <TextBox x:Name="PlainText" Height="Auto" Horizo
            <TextBlock Text="Enter your password salt (anyth
            <TextBox x:Name="Salt" Height="Auto" HorizontalA
            <Grid Margin="0,10,0,0">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*"/>
                    <ColumnDefinition Width="*"/>
                </Grid.ColumnDefinitions>
                <Button Content="Get Hash" Grid.Column="0" C
                <Button Content="Clear TextBox" Grid.Column=
            </Grid>
            <TextBlock Text="Hashed String:" Margin="0,10,0,
            <TextBox x:Name="HashedString" Height="Auto" Hor
            <GroupBox Header="Information" Margin="0,10,0,0"
                <TextBlock TextWrapping="Wrap" Margin="5">
                    This is a sample of MVVM-based WPF applic
                </TextBlock>
            </GroupBox>
        </StackPanel>
    </Grid>
</Window>
```

As can be seen there I used "Binding" to bind with the Model and ViewModel. Below is how the **untouched** code-behind of `HasherView.xaml.cs` looked like:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace SimpleOneWayHashing.Views
{
    /// <summary>
    /// Interaction logic for HasherView.xaml
    /// </summary>
    public partial class HasherView : Window
    {
        public HasherView()
        {
            InitializeComponent();
        }
    }
}
```

ICommand interface helper class

There is another new class here called `DelegateCommand`, basically inherits from `ICommand` interface. This helper class is essential for MVVM to work. It's a command that is meant to be executed by other classes to run code in this class by invoking delegates. Some people may called it the **RelayCommand**.

```
    ├── Commands
    │   ├── CommandReference.cs
    |   └── DelegateCommand.cs
```

Inside this "Password Hashing Tool" project, there are two files in the **Commands** folder; CommandReference.cs and DeletegateCommand.cs. The source code of both helper classes are well-documented, self-explainable and easily reusable for your MVVM project as well.

Finally, starting the application

This is usually known as `App.xaml` file. Since my project has **Themes** folder where it contains the XAML theme files, so this is how I created the resources inside `App.xaml`:

```
<Application x:Class="SimpleOneWayHashing.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/
             xmlns:x="http://schemas.microsoft.com/winfx/200
             Startup="Application_Startup">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Themes/Metro/Met
                <ResourceDictionary Source="Themes/Metro/Met
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

But, to start the application, this is the code snippet I used which can be found in the code-behind of the

startup file: `App.xaml.cs`.

```
Views.HasherView view = new Views.HasherView();
view.DataContext = new ViewModels.HashingViewModel();
view.Show();
```

This is how overall code looked like in `App.xaml.cs` file:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;

namespace SimpleOneWayHashing
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
        private void Application_Startup(object sender, Star
        {
            // Create the ViewModel and expose it using the \
            Views.HasherView view = new Views.HasherView();
            view.DataContext = new ViewModels.HashingViewMod
            view.Show();
        }
    }
}
```

Conclusion

Well, this is how “Password Hashing Tool” application has been developed with MVVM design pattern. This is one of the ways implementing MVVM design pattern in a project. Different people use different way implementing the MVVM design pattern. If you *google* around, there are a lot of resources about the implementation of MVVM design pattern available online, from the basic or simpler version to the advanced, larga-scale development.

Don’t forget to download the source code for this “Password Hashing Tool” project in my GitHub repo first and try to go through the code. It’s much easier for you to understand how it works. I think the source code are explainable enough. If you want to implement MVVM in your project, you may use this project as a starter or you may reuse some of the code inside as the code snippets.

You also can start developing your own MVVM application with MVVM frameworks that available online and free.

References

- Simple MVVM Pattern in WPF
- MVVM in WPF
- A Simple MVVM Example
- Wikipedia: Model View ViewModel
- What is the difference between WPF and

WinForms?

- Why use MVVM?
- Basic MVVM and ICommand Usage Example
- Most Developers Use MVVM Incorrectly. Are You?

‹ B A C K