MVVM Design Pattern in universal windows apps (UWP):

Case study: Microsoft applications built with C# using the UWP framework

Author: Eze-Odikwa Tochukwu jed

Introduction:

A design pattern solves many problems in software development by providing a framework for building an application because Existing design patterns make good templates for your objects, allowing you to build software faster. This article describes how to implement **MVVM** design in your universal windows platform **UWP** application.

So what are design patterns?

In software engineering, a **design pattern** is simply a repeatable solution to commonly occurring problems in the design of software which is accepted generally.

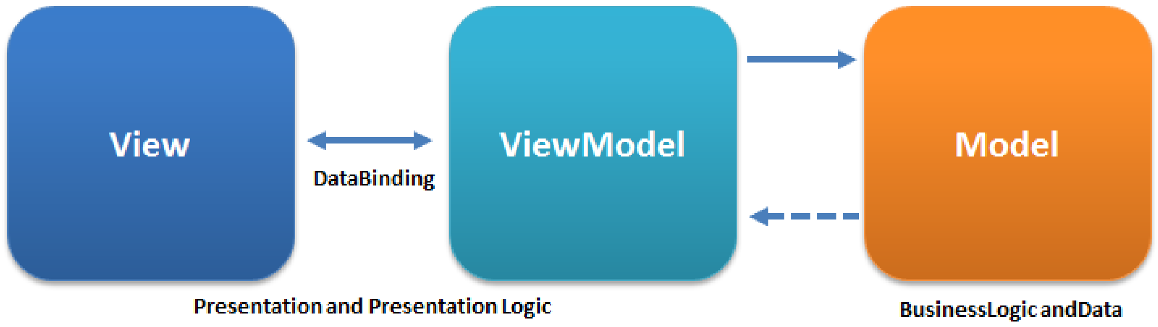
There are 3 main design pattern used in windows apps

(1) MVVM pattern
(2) MVC pattern
(3) MVP pattern

We will be discussing the latter two in separate topics of their own.

Model View View-model (MVVM): The Model View View-Model (MVVM) pattern helps developers separate an application's business and presentation logic from its user interface. Maintaining a clear separation between application logic and user interface helps address development and design issues, making an application easier to unit test, maintain, and develop. It can also improve the reusability of code and it allows multiple developers to collaborate more easily when working on the same project.

On implementation of the MVVM pattern, the user interface of your application, the underlying presentation and business logic are separated into three components as illustrated in the next page.



Three components of the MVVN pattern:

(1) The view component encapsulates the user interface and user interface logic.
(2) The view model component encapsulates presentation logic and state.
(3) The model layer encapsulates the application's business logic and data.

MVVM-light: MVVM-light is a toolkit written in C# which helps to accelerate the creation and development of MVVM applications in WPF, Silverlight, Windows Phone, Xamarin and Universal windows platform (UWP).

http://www.mvvmlight.net/doc

What you will learn:

This tutorial shows you how to create a Universal Windows app with MVVM Light support. You will learn how to:

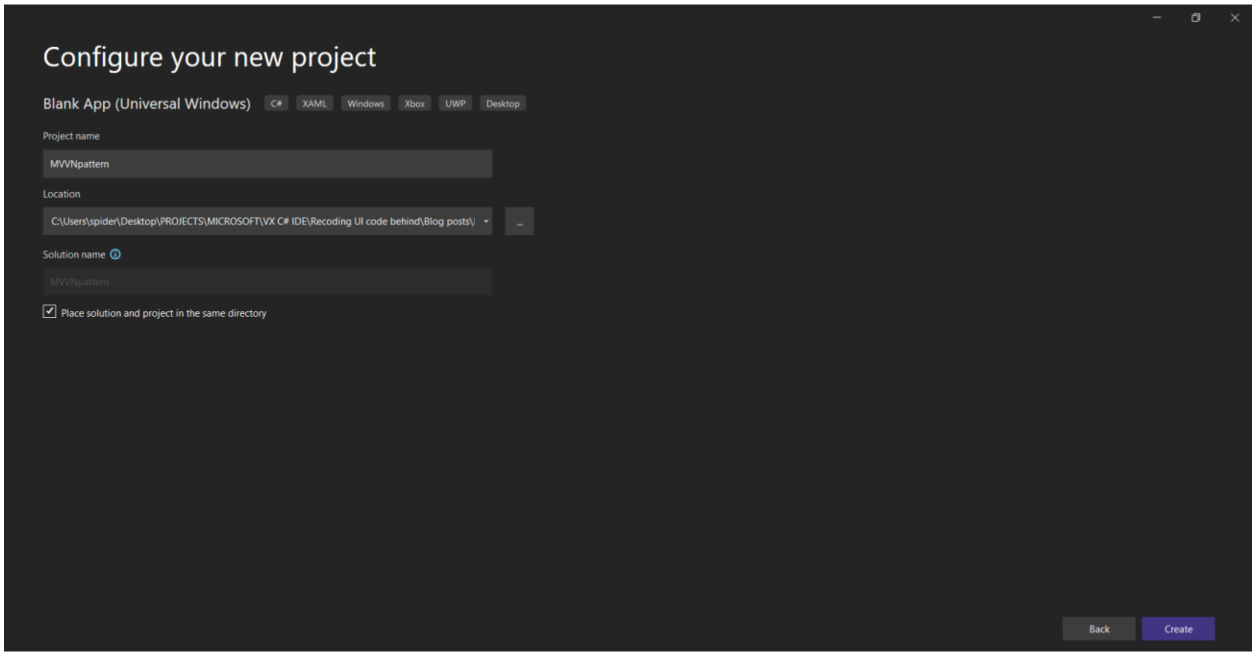
(1) create a Universal Windows platform app and add support for MVVM Light
(2) implement the directory structure
(3) add the view model layer
(4) wire the data context
(5) Run the sample UWP application.

How to Implement MVVN pattern in UWP:

1. Create a Universal Windows Platforms App:

Let's start by creating a Universal Windows app in this case I am using visual studio 2022. Select **create a new Project** from the menu in Visual Studio. In **Templates > C# > Windows > UWP** select Blank App (Universal Windows) from the list of project templates. Name your project in my case I will call it "MVVNpattern" and click create to create the project.

This creates a project. The Windows 10 project is platform specific projects and is responsible for creating the application packages (.appx) targeting the respective platforms. The project is a container for code that runs on its respective platforms.


Configure your new project

Blank App (Universal Windows) C# XAML Windows Xbox UWP Desktop

Project name: MVVNpattern

Location: C:\Users\spider\Desktop\PROJECTS\MICROSOFT\VX C# IDE\Recoding UI code behind\Blog posts\

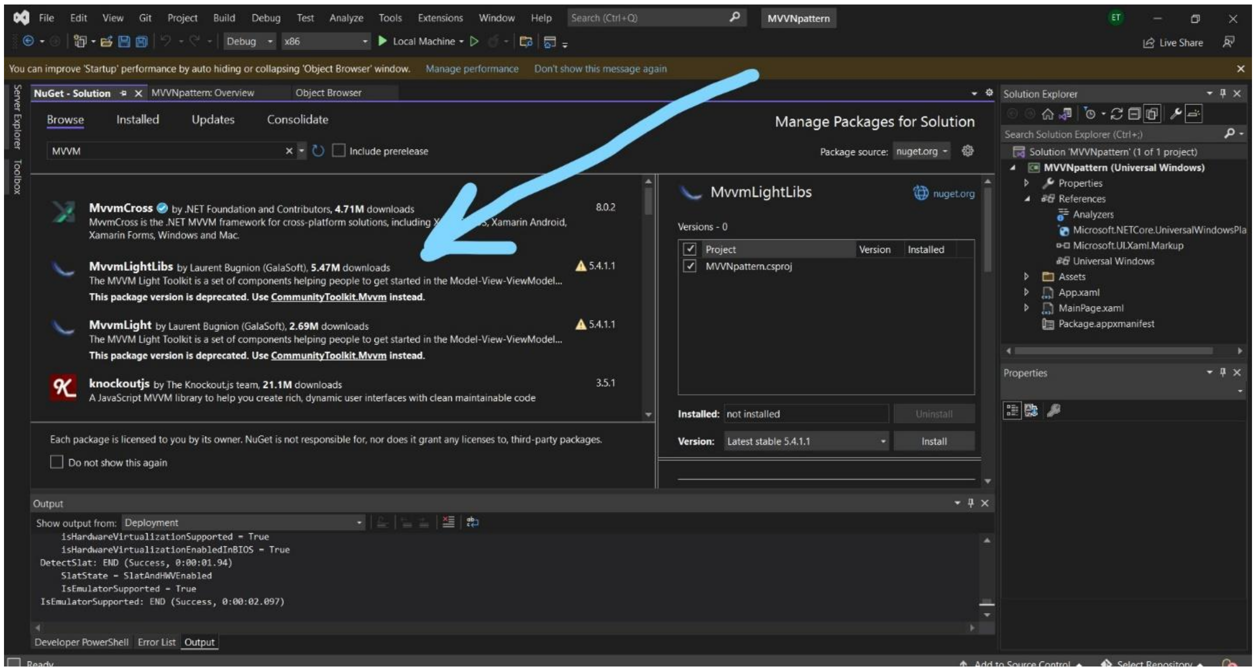
Solution name: MVVNpattern

[✓] Place solution and project in the same directory

<u>2. Add MVVM Light Support:</u>

Right-click the solution name in the Solution Explorer and select **Manage Nuget Packages for Solution**.

Select the Browse tab and search for MVVM Light. Select the package MvvmLightLibs from the search results. Click Install to add the MVVM Light libraries to the project.


You can improve 'Startup' performance by auto hiding or collapsing 'Object Browser' window. Manage performance Don't show this message again

MvvmCross by .NET Foundation and Contributors, **4.71M** downloads 8.0.2
MvvmCross is the .NET MVVM framework for cross-platform solutions, including Xamarin.iOS, Xamarin Android, Xamarin Forms, Windows and Mac.

MvvmLightLibs by Laurent Bugnion (GalaSoft), **5.47M** downloads 5.4.1.1
The MVVM Light Toolkit is a set of components helping people to get started in the Model-View-ViewModel...
This package version is deprecated. Use CommunityToolkit.Mvvm instead.

MvvmLight by Laurent Bugnion (GalaSoft), **2.69M** downloads 5.4.1.1
The MVVM Light Toolkit is a set of components helping people to get started in the Model-View-ViewModel...
This package version is deprecated. Use CommunityToolkit.Mvvm instead.

knockoutjs by The Knockout.js team, **21.1M** downloads 3.5.1
A JavaScript MVVM library to help you create rich, dynamic user interfaces with clean maintainable code

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

Project	Version	Installed
✓ MVVNpattern.csproj		

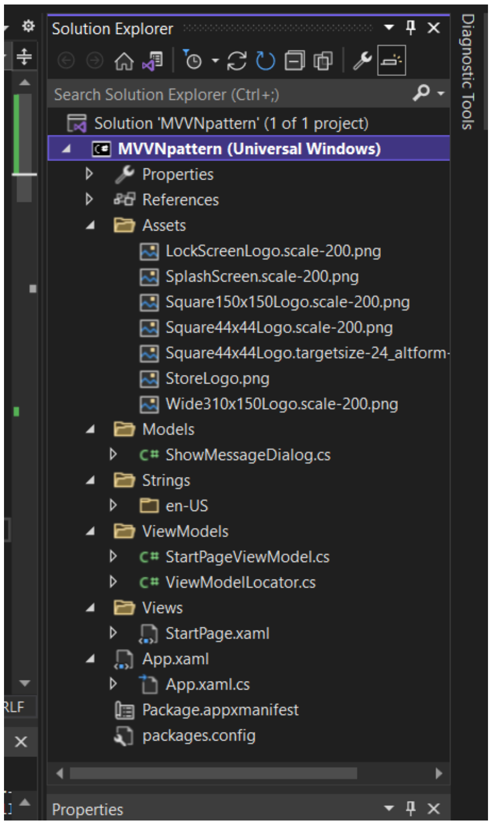
Installed: not installed
Version: Latest stable 5.4.1.1

```
    isHardwareVirtualizationSupported = True
    isHardwareVirtualizationEnabledInBIOS = True
DetectSlat: END (Success, 0:00:01.94)
    SlatState = SlatAndHWVEnabled
    IsEmulatorSupported = True
IsEmulatorSupported: END (Success, 0:00:02.097)
```

At this point, you have added MVVM Light support to your project.

3. Project File Structure:

A Universal Windows app that adopts the MVVM pattern requires a particular directory structure. The following snapshot shows a possible project file structure for a Universal Windows app.



```
Solution 'MVVNpattern' (1 of 1 project)
  MVVNpattern (Universal Windows)
    Properties
    References
    Assets
      LockScreenLogo.scale-200.png
      SplashScreen.scale-200.png
      Square150x150Logo.scale-200.png
      Square44x44Logo.scale-200.png
      Square44x44Logo.targetsize-24_altform-
      StoreLogo.png
      Wide310x150Logo.scale-200.png
    Models
      ShowMessageDialog.cs
    Strings
      en-US
    ViewModels
      StartPageViewModel.cs
      ViewModelLocator.cs
    Views
      StartPage.xaml
    App.xaml
      App.xaml.cs
    Package.appxmanifest
    packages.config
```

Let me walk you through the project structure of a typical Universal Windows app that adopts the MVVM pattern:

Strings: This directory contains strings and resources for application localization. The Strings directory contains separate directories for every supported language. The en-US directory, for example, contains resources for the English (US) language.

Models: In the MVVM pattern, the model encapsulates the business logic and data. Generally, the model implements the facilities that make it easy to bind properties to the view layer. This means that it supports "property changed" and "collection changed" notifications through the **INotifyPropertyChanged** and **INotifyCollectionChanged** interfaces.

ViewModels: The view model in the MVVM pattern encapsulates the presentation logic and data for the view. It has no direct reference to the view or any knowledge about the view's implementation or type.

Services: This section can include classes for web service calls, navigation service, etc.

Utils: includes utility functions that can be used across the app. Examples include AppCache, FileUtils, Constants, NetworkAvailability, GeoLocation, etc.

Views: This directory contains the user interface layouts. Platform specific views are added directly to the platform specific project and common views are added to the project.

Depending on the type of view, the name should end with:

- Window, a non-modal window
- Dialog, a (modal) dialog window
- Page, a page view (mostly used in Windows Phone and Windows Store apps)
- View, a view that is used as subview in another view, page, window, or dialog

The name of a view model is composed of the corresponding view's name and the word "Model". The view models are stored in the same location in the ViewModels directory as their corresponding views in the Views directory.

4. Adding the View Model Layer:

The view model layer implements properties and commands to which the view can bind data and notify the view of any state changes through change notification events. The properties and commands the view model provides, define the functionality offered by the user interface.

The following list summarizes the characteristics, tasks, and responsibilities of the view model layer:

- It coordinates the view's interaction with any model class.
- The view model and the model classes generally have a one-to-many relationship.
- It can convert or manipulate model data so that it can be easily consumed by the view.
- It can define additional properties to specifically support the view.
- It defines the logical states the view can use to provide visual changes to the user interface.
- It defines the commands and actions the user can trigger.

Now Let's Set up the Project Structure since Folders and classes are not added to project resources automatically as used to early in windows 8.1

Step 1: Right Click on the project in Solution Explorer and add three folders **Models**, **Views**, **ViewModels**.

Step 2: Delete the existing page MainPage.xaml and add a new item StartPage.xaml in Views folder. Simply right Click on Views folder and add New Item and select Blank Page name it StartPage.xaml.

Since we deleted the default Startup page MainPage.xaml we will now have to redo the App's Startup Page to StartPage.xaml in Views folder. To do this look for App.xaml.cs and search for the word MainPage. You will find following code under OnLaunched method:

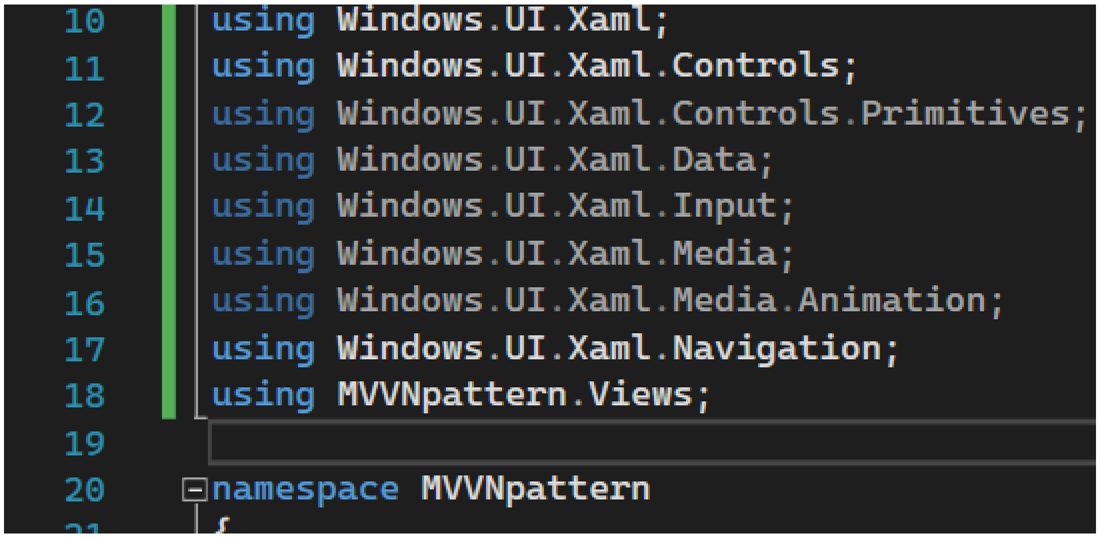
```
rootFrame.Navigate(typeof(MainPage), e.Arguments);
```

Go ahead and change it to StartPage.xaml

```
rootFrame.Navigate(typeof(StartPage), e.Arguments);
```

You will also need to add the dependency views; in app.xaml.cs and in my case

```
using MVVNpattern.Views;
```



```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Media.Animation;
using Windows.UI.Xaml.Navigation;
using MVVNpattern.Views;

namespace MVVNpattern
{
```

Step 3: Add a new Class ViewModelLocator.cs to the ViewModels folder. Right click on the models folder select Add > New class.cs.

ViewModelLocator is a class that centralizes the definitions of all the ViewModels in an app so that they can be cached and retrieved on demand, usually via Dependency Injection. ViewModelLocator is an idiom that lets you keep the benefits of DI(Dependency Injection) in your MVVM application while also allowing your code to play well with visual designers. ViewModelLocator needs to setup the IOC provider, register each individual ViewModel, and finally expose those registered ViewModels for use by the rest of the application.

Copy and paste the following code to your class:

```
using GalaSoft.MvvmLight;

using GalaSoft.MvvmLight.Ioc;

using GalaSoft.MvvmLight.Views;

using Microsoft.Practices.ServiceLocation;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;



namespace MVVMPattern.ViewModels

{

    /// <summary>

    /// This class contains static references to all the view models in the

    /// application and provides an entry point for the bindings.

    /// </summary>
    public class ViewModelLocator

    {

        /// <summary>

        /// Initializes a new instance of the ViewModelLocator class.

        /// </summary>

        public ViewModelLocator()

        {

       //Incase you experience error on this line downgrade nuget to 5.20

            ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);

            if (ViewModelBase.IsInDesignModeStatic)

            {

                // Create design time view services and models

            }

            else

            {

                // Create run time view services and models

            }



            //Register your services used here

            SimpleIoc.Default.Register<INavigationService,
NavigationService>();
```

```
            SimpleIoc.Default.Register<StartPageViewModel>();



        }





        // <summary>

        // Gets the StartPage view model.

        // </summary>

        // <value>

        // The StartPage view model.

        // </value>

        public StartPageViewModel StartPageInstance

        {

            get

            {

                return
ServiceLocator.Current.GetInstance<StartPageViewModel>();

            }

        }



        // <summary>
        // The cleanup.

        // </summary>

        public static void Cleanup()

        {

            // TODO Clear the ViewModels

        }

    }



}
```

Add another Class in ViewModles folder StartPageViewModel.cs and replace the code inside with the following:

```
using GalaSoft.MvvmLight;



namespace Win10MVVMLightDemo.ViewModels

{

    public class StartPageViewModel : ViewModelBase

    {

        private bool _isLoading = false;

        public bool IsLoading

        {
            get

            {

                return _isLoading;

            }

            set

            {

                _isLoading = value;

                RaisePropertyChanged("IsLoading");



            }

        }

        private string _title;

        public string Title

        {



            get

            {

                return _title;

            }

            set
            {

                if (value != _title)

                {

                    _title = value;

                    RaisePropertyChanged("Title");

                }

            }

        }



        public StartPageViewModel()

        {

            Title = "Hello Tochy";

        }

    }

}
```

The ServiceLocator is responsible for retrieving the ViewModel instances, using the SimpleIoc.Default implementation provided by MVVM Light. By registering them via the SimpleIoc.Default instance in the constructor, we can retrieve those instances from the Views via the public properties defined in the locator class.

Step 4: Head over to app.xaml and change the code to the following below

```
<Application


    x:Class="MVVNpattern.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"


    xmlns:local="using:MVVNpattern"


    xmlns:vm="using:MVVNpattern.ViewModels"


    RequestedTheme="Light">


    <Application.Resources>


        <vm:ViewModelLocator xmlns:vm="using:MVVNpattern.ViewModels"
x:Key="Locator" />


    </Application.Resources>


</Application>
```

5. Wiring up the Data Context:

The view and the view model can be constructed and associated at runtime in multiple ways. The simplest approach is for the view to instantiate its corresponding view model in XAML. You can also specify in XAML that the view model is set as the view's data context.

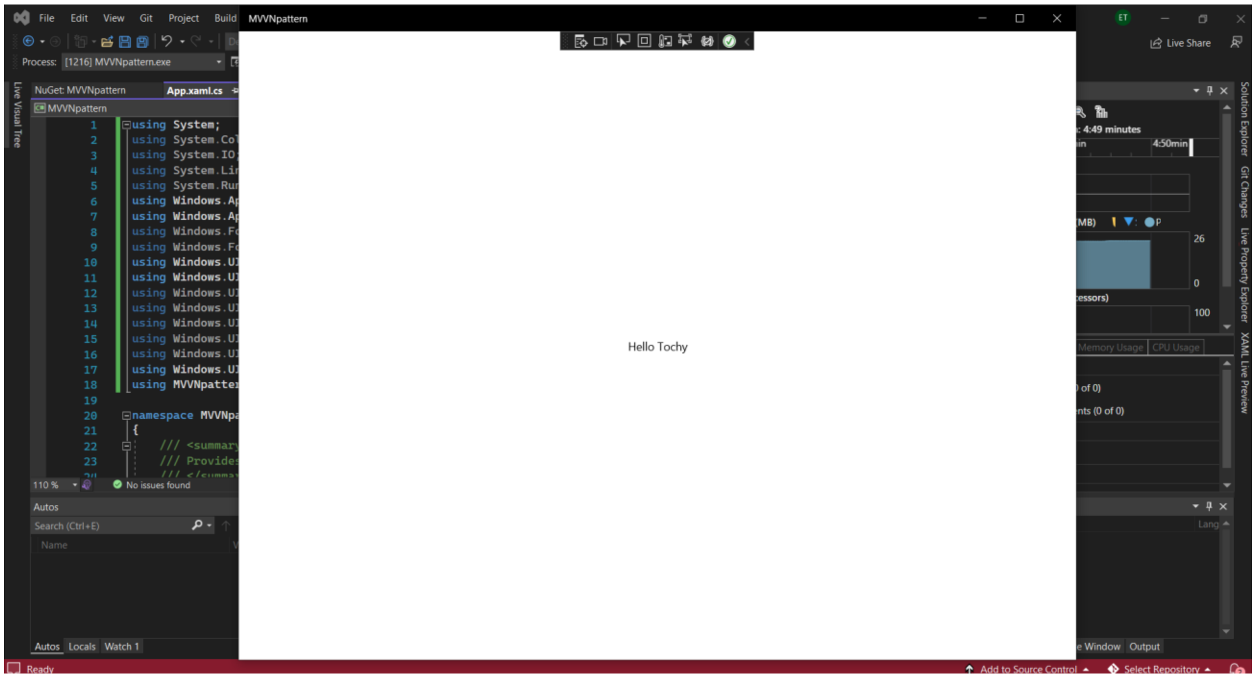
Now we will edit the StartPage.xaml to add the ViewModel to its appropriate page via the DataContext for this add following code to StartPage.xaml and set the DataContext to SartPageInstance that we defined in the ViewModelLocator.cs.

Copy and paste the code below in your StartPage.xaml:

```
<Page

    x:Class="MVVNpattern.Views.StartPage"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

    xmlns:local="using:MVVNpattern.Views"

    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

    DataContext="{Binding StartPageInstance, Source={StaticResource
Locator}}"

    mc:Ignorable="d">



    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

        <TextBlock Name="Title"  HorizontalAlignment="Center"
VerticalAlignment="Center" Text="{Binding Title}" FontFamily="Segoe UI
Historic" />

    </Grid>

</Page>
```

When the StartPage.xaml page is initialized, an instance of the StartPageViewModel is automatically constructed and set as the view's data context. Note that the view model must have a default parameter-less constructor for this approach to work.

6. Running the project: Now our project is ready to run click on green play icon with and for the smoothest results make sure it’s set to local machine. If all is okay the app in debug Mode will look something like this:



```
using System;
using System.Co
using System.IO
using System.Li
using System.Ru
using Windows.Ap
using Windows.Ap
using Windows.Fo
using Windows.Fo
using Windows.UI
using Windows.UI
using Windows.UI
using Windows.UI
using Windows.UI
using Windows.UI
using Windows.UI
using Windows.UI
using MVVNpatte

namespace MVVNpa
{
    /// <summary
    /// Provides
```

Conclusion

By implementing the MVVM pattern, we have a clear separation between the view, view model, and model layers. Typically, we try to develop the view model so that it doesn’t know anything about the view that it drives. This has multiple advantages:

The developer team can work independently from the user interface team.

The view model can be tested easily, simply by calling some commands and methods, and asserting the value of properties.

Changes can be made to the view without having to worry about the effect it will have on the view model and the model.

https://github.com/tochyodikwa/MVVMPatternUWP_Windows10

Summary:

A design pattern can solve many problems by providing a framework for building an application.

Writing computer code used to be extremely difficult. Back in the old days the only way to make a program run at more than a desultory crawl was to write it using machine code. Even a simple word processing utility could take several weeks to create as you meticulously planned the memory allocations for variables, wrote routines to perform simple tasks such as drawing characters on the screen and decoding keyboard input, and then typed every individual processor instruction into an assembler.

Existing design patterns make good templates for your objects, allowing you to build software faster.