13
Feb 2021

# IMPLEMENTING THE MVVM PATTERN IN A WPF APP WITH THE MVVM COMMUNITY TOOLKIT

by bsonnino · 18 Comments

## Introduction

You have an old, legacy app (with no tests), and its age is starting to show – it's using an old version of the .NET Framework, it's difficult to maintain and every new feature introduced brings a lot of bugs. Developers are afraid to change it, but the users ask for new features. You are at a crossroad: throw the code and rewrite everything or refactor the code. Does this sound familiar to you ?

I'm almost sure that you're leaning to throw the code and rewrite everything. Start fresh, use new technologies and create the wonderful app you've always dreamed of. But this comes with a cost: the old app is still there, functional, and must be maintained while you are developing the new one. There are no resources to develop both apps in parallel and the new app will take a long time before its finished.

So, the only way to go is to refactor the old app. It's not what you wanted, but it can still be fun – you will be able to use the new technologies, introduce good programming practices, and at the end, have the app you have dreamed. No, I'm not saying it will be an easy way, but it will be the most viable one.

This article will show how to port a .NET 4 WPF app and port it to .NET 5, introduce the MVVM pattern and add tests to it. After that, you will be able to change its UI, using WinUI3, like we did in this article.

## The original app

The original app is a Customer CRUD, developed in .NET 4, with two projects – the UI project, CustomerApp, and a library CustomerLib, that access client's data in an XML file (I did that just for the sake of simplicity, but this could be changed easily for another data source, like a database). You can get the app from here, and when you run it, you get something like this:

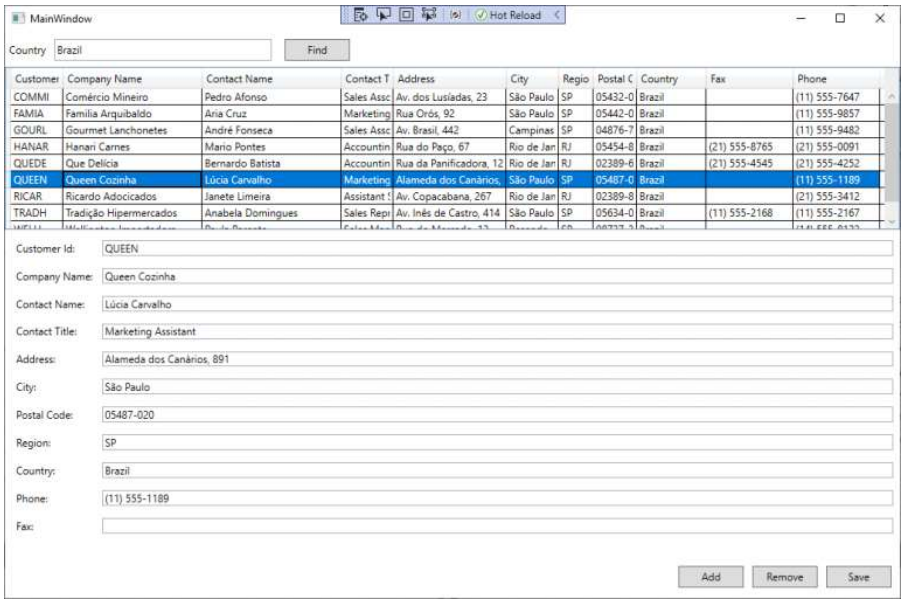
Search

Categories

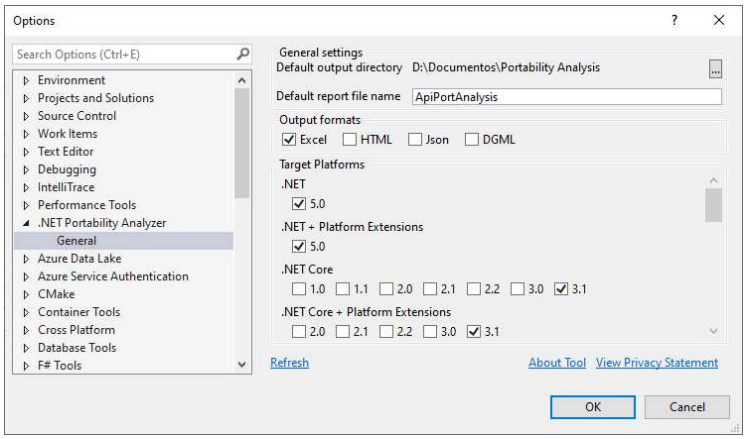
Development
English
Português
Uncategorized
Windows

Archives

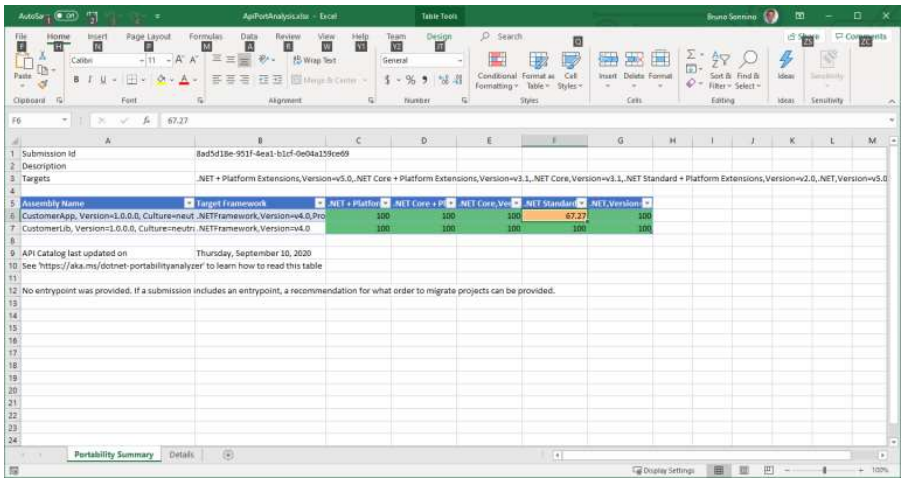
March 2022
February 2022
January 2022
July 2021
June 2021
May 2021
April 2021
March 2021
February 2021
January 2021
December 2020
October 2020
September 2020
April 2020
March 2020
January 2020
November 2019
September 2019
August 2019
July 2019
June 2019
April 2019




The first step will be converting it to .NET 5. Before that, we will see how portable is our app, using the .NET Portability analyzer. It's a Visual studio extension that you can download from here. Once you download and install it, ou can run it in Visual Studio with **Analyze/Portability Analyzer Settings**:




You must select the platforms you want and click OK. Then, you must select **Analyze/Analyze Assembly Portability**, select the executables for the app and click OK. That will generate an Excel file with the report:



As you can see, our app can be ported safely to .NET 5. If there are any problems, you can check them in the **Details** tab. There, you will have a list of all APIs that you won't be able to port, where you should find a workaround. Now, we'll start converting the app.

## Converting the app to .NET 5

To convert the app, we'll start converting the lib to .NET Standard 2.0. To do that, right-click in the Lib project in the Solution Explorer and select **Unload Project**, then edit the project file and change it to:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFrameworks>netstandard2.0</TargetFrameworks>
  </PropertyGroup>
  <ItemGroup>
    <Content Include="Customers.xml">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirecto
    </Content>
  </ItemGroup>
</Project>
```

The project file is very simple, just set the taget framework to **netstandard2.0** and copy the item group relative to the xml file, so it's included in the final project. Then, you must reload the project and remove the **AssemblyInfo** file from the **Properties** folder, as it isn't needed anymore (if you leave it, it will generate an error, as it's included automatically by the new project file).

Then, right click the app project in the Solution Explorer and select **Unload Project**, to edit the project file:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net5.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include="..\CustomerLib\CustomerLib.csproj
      <Name>CustomerLib</Name>
    </ProjectReference>
```

March 2011
December 2010
November 2009
June 2009
April 2009
March 2009
February 2009
January 2009
December 2008
November 2008
October 2008
July 2008
March 2008
February 2008
January 2008
December 2007
November 2007
October 2007
September 2007
August 2007
July 2007

Recent Posts

No Code font tester in WPF
Developing for Linux in Windows with C#
Creating a VS Code Extension
Interactive Notebooks with C#
Linq improvements in .NET 6

Tag Cloud

.NET Algorithms asp.NET C# Debugging Delphi Desktop Bridge Desktop icons Linq mvvm NTFS OpenXML PowerShell Recovery disk Safe mode Sensors sql server Surface Dial Testing Tools TypeScript Unit Testing UWP Visual Studio VS Code

```
    </ItemGroup>
  </Project>
```

We are setting the output type to **WinExe**, setting the target framework to **net5.0-windows**, telling that we will use the .net 5 features, plus the ones specific to Windows. If we don't do that, we wouldn't be able to use the WPF features, that are specific to Windows and set **UseWPF** to **true**. Then, we copy the lib's ItemGroup to the project. When we reload the project, we must delete the **Properties** folder and we can build our app, converted to .NET 5. It should work exactly the way it did before. We are in the good path, porting the app to the newest .NET version, but we still have a long way to go. Now it's time to add good practices to our app, using the MVVM Pattern.

## The MVVM Pattern

The MVVM (Model-View-ViewModel) pattern was created on 2005 by John Gossman, a Microsoft Architect on Blend team, and it makes extensive use of the DataBinding feature existent in WPF and other XAML platforms (like UWP or Xamarin). It provides separation between data (Model) and its visualization (View), using a binding layer, the ViewModel.

The ViewModel is a class that implements the **INotifyPropertyChanged** interface:

```
public interface INotifyPropertyChanged
{
  event PropertyChangedEventHandler PropertyChanged;
}
```

It has just one event, **PropertyChanged** that is activated when there is a change in a property. The Data binding mechanism present in WPF (and in other XAML platforms) subscribes this event and updates the view with no program intervention. So, all we need to do is to create a class that implements **INotifyPropertyChanged** and call this event when there is a change in a property to WPF update the view.

The greatest advantage is that the ViewModel is a normal class and doesn't have any dependency on the view layer. That way, we don't need to initialize a window when we test the ViewModel. This image shows the basic structure of this pattern:

WebView2 Windows Windows 10 WPF XAML

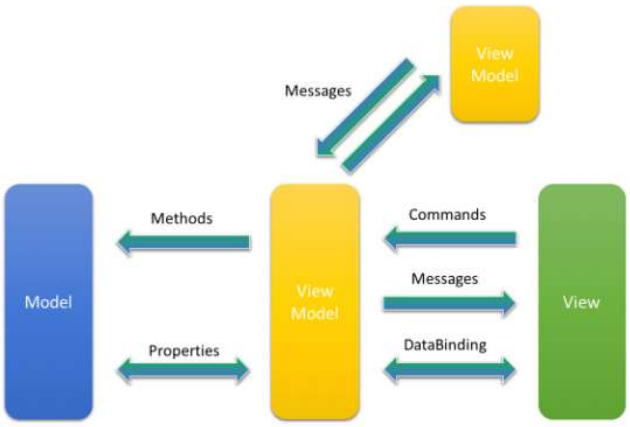
Meta

Log in

Entries RSS

Comments RSS

WordPress.org




The model communicates with the ViewModel by its properties and methods. The ViewModel communicates with the View mainly using Data Binding, it receives Commands from the View and can send messages to it. When there are many ViewModels that must communicate , they usually send messages, to maintain a decoupled architecture. That way, the Model (usually a POCO class – Plain Old CSharp Object) doesn't know about the ViewModel, the ViewModel isn't coupled with the View or other ViewModels, and the View isn't tied to a ViewModel directly (the only tie is the View's DataContext property, that will bind the View and the ViewModel. The rest will be done by Data Binding).

We could implement all this infrastructure by ourselves, it's not a difficult task, but it's better to use a Framework for that. There are many MVVM frameworks out there, each one chooses a different approach to implement the infrastructure and select one of them is just a matter of preference. I've used MVVM Light toolkit for years, it's very lightweight and easy to use, but it isn't maintained anymore, so I decided to search another framework. Fortunately, the Windows Community Toolkit has provided a new framework, inspired on MVVM Light, the MVVM Community Toolkit.

## Implementing the MVVM Pattern

In our project, the Model is already separated from the rest: it's in the Lib project and we'll leave that untouched, as we don't have to change anything in the model. In order to use the MVVM Toolkit, we must add the **Microsoft.Toolkit.Mvvm** NuGet package.

Then, we must create the ViewModels to interact between the View and the Model. Create a new folder and name it **ViewModel**. In it, add a new class and name it **MainViewModel.cs.** This class will inherit from **ObservableObject**, from the toolkit, as it already implements the interface. Then we will copy and adapt the code that is found in the code behind of MainWindow.xaml.cs:

```
1.  public class MainViewModel : ObservableObject
2.  {
3.      private readonly ICustomerRepository _customerRepository;
4.      private Customer _selectedCustomer;
5.
6.      public MainViewModel()
7.      {
8.          _customerRepository =  new CustomerRepository();
9.          AddCommand = new RelayCommand(DoAdd);
```

```
10.           RemoveCommand = new RelayCommand(DoRemove, () => Selecte
11.           SaveCommand = new RelayCommand(DoSave);
12.           SearchCommand = new RelayCommand<string>(DoSearch);
13.       }
14.
15.       public IEnumerable<Customer> Customers => _customerRepositor
16.
17.       public Customer SelectedCustomer
18.       {
19.           get => _selectedCustomer;
20.           set
21.           {
22.               SetProperty(ref _selectedCustomer, value);
23.               RemoveCommand.NotifyCanExecuteChanged();
24.           }
25.       }
26.
27.       public IRelayCommand AddCommand { get; }
28.       public IRelayCommand RemoveCommand { get; }
29.       public IRelayCommand SaveCommand { get; }
30.       public IRelayCommand<string> SearchCommand { get; }
31.       private void DoAdd()
32.       {
33.           var customer = new Customer();
34.           _customerRepository.Add(customer);
35.           SelectedCustomer = customer;
36.           OnPropertyChanged("Customers");
37.       }
38.
39.       private void DoRemove()
40.       {
41.           if (SelectedCustomer != null)
42.           {
43.               _customerRepository.Remove(SelectedCustomer);
44.               SelectedCustomer = null;
45.               OnPropertyChanged("Customers");
46.           }
47.       }
48.
49.       private void DoSave()
50.       {
51.           _customerRepository.Commit();
52.       }
53.
54.       private void DoSearch(string textToSearch)
55.       {
56.           var coll = CollectionViewSource.GetDefaultView(Customers
57.           if (!string.IsNullOrWhiteSpace(textToSearch))
58.               coll.Filter = c => ((Customer)c).Country.ToLower().C
59.           else
60.               coll.Filter = null;
61.       }
62.   }
```

We have two properties, **Customers** and **SelectedCustomer**. **Customers** will contain the list of customers shown in the DataGrid. **SelectedCustomer** will be the selected customer in the DataGrid, that will be shown in the detail pane. There are four commands, and we will use the **IRelayCommand** interface, declared in the toolkit. Each command will be initialized with the method that will be executed when the command is invoked. The **RemoveCommand** uses an overload for the constructor, that uses a predicate as the second parameter. This predicate will only enable the button when there is a customer

selected in the DataGrid. As this command is dependent on the selected customer, when we change this property, we call the **NotifyCanExecuteChanged** method to notify all the elements that are bound to this command.

Now we can remove all the code from **MainWindow.xaml.cs** and leave only this:

```
public MainWindow()
{
    InitializeComponent();
    DataContext = new MainViewModel();
}
```

We can run the program and see that it runs the same way it did before, but we made a large refactoring to the code and now we can start implementing unit tests in the code.

## Implementing tests

Now that we've separated the code from the view, we can test the ViewModel without the need to initialize a Window. That is really great, because we can have testable code and be assured that we are not breaking anything when we are implementing new features. For that, add a new test project and name it **CustomerApp.Tests**. In the Visual Studio version I'm using, there is no template for the .net 5.0 test project available, so I added a .Net Core test project, then I edited the project file and changed the TargetFramework to **net5.0-windows.** Then, you can add a reference to the CustomerApp project and rename **UnitTest1** to **MainViewModelTests**.

Taking a look at the Main ViewModel, we see that there is a coupling between it and the Customer Repository. In this case, there is no much trouble, because we are reading the customers from a XML file located in the output directory, but if we decide to replace it with some kind of database, it can be tricky to test the ViewModel, because we would have to do a lot of setup to test it.

We'll remove the dependency using Dependency Injection. Instead of using another framework for the the dependency injection, we'll use the integrated one, based on **Microsoft.Extensions.DependencyInjection**. You should add this NuGet package in the App project to use the dependency injection. Then, in **App.xaml.cs**, we'll add code to initialize the location of the services:

```
public partial class App
{
    public App()
    {
        Services = ConfigureServices();
    }

    public new static App Current => (App) Application.Current;

    public IServiceProvider Services { get; }

    private static IServiceProvider ConfigureServices()
    {
        var services = new ServiceCollection();

        services.AddSingleton<ICustomerRepository, CustomerRepos
        services.AddSingleton<MainViewModel>();

        return services.BuildServiceProvider();
    }

```

```
22.       public MainViewModel MainVM => Services.GetService<MainViewM
23.   }
```

We declare a static property **Current** to ease using the **App** object and declare a **IServiceProvider**, to provide our services. They are configured in the **ConfigureServices** method, that creates a ServiceCollection and add the CustomerRepository and the main ViewModel to the collection. **ConfigureServices** is called in the constructor of the application. Finally we declare the property **MainVM**, which will get the ViewModel from the Service Collection.

Now, we can change **MainWindow.xaml.cs** to use the property instead of instantiate directly the ViewModel:

```
1.  public MainWindow()
2.  {
3.      InitializeComponent();
4.      DataContext = App.Current.MainVM;
5.  }
```

The last change is to remove the coupling between the ViewModel and the repository using Dependency Injection, in **MainViewModel.cs**:

```
1.  public MainViewModel(ICustomerRepository customerRepository)
2.  {
3.      _customerRepository = customerRepository ??
4.                            throw new ArgumentNullException("custo
5.      _customerRepository = customerRepository;
6.      AddCommand = new RelayCommand(DoAdd);
7.      RemoveCommand = new RelayCommand(DoRemove, () => SelectedCus
8.      SaveCommand = new RelayCommand(DoSave);
9.      SearchCommand = new RelayCommand<string>(DoSearch);
10. }
```

With that, we've gone one step further and removed the coupling between the ViewModel and the repository, so we can start our tests.

For the tests, we will use two libraries, FluentAssertions, for better assertions and FakeItEasy, to generate fakes. You should install both NuGet packages to your test project. Now, we can start creating our tests:

```
1.  public class MainViewModelTests
2.  {
3.      [TestMethod]
4.      public void Constructor_NullRepository_ShouldThrow()
5.      {
6.          Action act = () => new MainViewModel(null);
7.
8.          act.Should().Throw<ArgumentNullException>()
9.              .Where(e => e.Message.Contains("customerRepository")
10.     }
11.
12.     [TestMethod]
13.     public void Constructor_Customers_ShouldHaveValue()
14.     {
15.         var repository = A.Fake<ICustomerRepository>();
16.         var customers = new List<Customer>();
17.         A.CallTo(() => repository.Customers).Returns(customers);
```

```
            var vm =  new MainViewModel(repository);

            vm.Customers.Should().BeEquivalentTo(customers);
        }

        [TestMethod]
        public void Constructor_SelectedCustomer_ShouldBeNull()
        {
            var repository = A.Fake<ICustomerRepository>();
            var vm = new MainViewModel(repository);

            vm.SelectedCustomer.Should().BeNull();
        }
    }
```

Here we created three tests for the constructor, testing the values of the properties after the constructor. We can continue, testing the commands in the ViewModel:

```
    [TestMethod]
    public void AddCommand_ShouldAddInRepository()
    {
        var repository = A.Fake<ICustomerRepository>();
        var vm = new MainViewModel(repository);

        vm.AddCommand.Execute(null);
        A.CallTo(() => repository.Add(A<Customer>._)).MustHaveHappen
    }

    [TestMethod]
    public void AddCommand_SelectedCustomer_ShouldNotBeNull()
    {
        var repository = A.Fake<ICustomerRepository>();
        var vm = new MainViewModel(repository);
        vm.AddCommand.Execute(null);
        vm.SelectedCustomer.Should().NotBeNull();
    }

    [TestMethod]
    public void AddCommand_ShouldNotifyCustomers()
    {
        var repository = A.Fake<ICustomerRepository>();
        var vm = new MainViewModel(repository);
        var wasNotified = false;
        vm.PropertyChanged += (s, e) =>
        {
            if (e.PropertyName == "Customers")
                wasNotified = true;
        };
        vm.AddCommand.Execute(null);
        wasNotified.Should().BeTrue();
    }

    [TestMethod]
    public void RemoveCommand_SelectedCustomerNull_ShouldNotRemoveIn
    {
        var repository = A.Fake<ICustomerRepository>();
        var vm = new MainViewModel(repository);
        vm.RemoveCommand.Execute(null);
        A.CallTo(() => repository.Remove(A<Customer>._)).MustNotHavel
    }

```

```
44.  [TestMethod]
45.  public void RemoveCommand_SelectedCustomerNotNull_ShouldRemoveIn
46.  {
47.      var repository = A.Fake<ICustomerRepository>();
48.      var vm = new MainViewModel(repository);
49.      vm.SelectedCustomer = new Customer();
50.      vm.RemoveCommand.Execute(null);
51.      A.CallTo(() => repository.Remove(A<Customer>._)).MustHaveHap
52.  }
53.
54.  [TestMethod]
55.  public void RemoveCommand_SelectedCustomer_ShouldBeNull()
56.  {
57.      var repository = A.Fake<ICustomerRepository>();
58.      var vm = new MainViewModel(repository);
59.      vm.SelectedCustomer = new Customer();
60.      vm.RemoveCommand.Execute(null);
61.      vm.SelectedCustomer.Should().BeNull();
62.  }
63.
64.  [TestMethod]
65.  public void RemoveCommand_ShouldNotifyCustomers()
66.  {
67.      var repository = A.Fake<ICustomerRepository>();
68.      var vm = new MainViewModel(repository);
69.      vm.SelectedCustomer = new Customer();
70.      var wasNotified = false;
71.      vm.PropertyChanged += (s, e) =>
72.      {
73.          if (e.PropertyName == "Customers")
74.              wasNotified = true;
75.      };
76.      vm.RemoveCommand.Execute(null);
77.      wasNotified.Should().BeTrue();
78.  }
79.
80.  [TestMethod]
81.  public void SaveCommand_ShouldCommitInRepository()
82.  {
83.      var repository = A.Fake<ICustomerRepository>();
84.      var vm = new MainViewModel(repository);
85.      vm.SaveCommand.Execute(null);
86.      A.CallTo(() => repository.Commit()).MustHaveHappened();
87.  }
88.
89.  [TestMethod]
90.  public void SearchCommand_WithText_ShouldSetFilter()
91.  {
92.      var repository = A.Fake<ICustomerRepository>();
93.      var vm = new MainViewModel(repository);
94.      vm.SearchCommand.Execute("text");
95.      var coll = CollectionViewSource.GetDefaultView(vm.Customers)
96.      coll.Filter.Should().NotBeNull();
97.  }
98.
99.  [TestMethod]
100. public void SearchCommand_WithoutText_ShouldSetFilter()
101. {
102.     var repository = A.Fake<ICustomerRepository>();
103.     var vm = new MainViewModel(repository);
104.     vm.SearchCommand.Execute("");
105.     var coll = CollectionViewSource.GetDefaultView(vm.Customers)
106.     coll.Filter.Should().BeNull();
```

```
107.    }
```

Now we have all the tests for the ViewModel and have our project ready for the future. We went step by step and finished with a .NET 5.0 project that uses the MVVM pattern and have unit tests. This project is ready to be updated to WinUI3, or even to be ported to UWP or Xamarin. The separation between the code and the UI makes it easy to port it to other platforms, the ViewModel became testable and you can test all logic in it, without bothering with the UI. Nice, no ?

The full source code for the project is at https://github.com/bsonnino/MvvmApp

Posted in: Development, English, Windows · Tagged: .NET, C#, mvvm, WPF

## 18 Thoughts on "Implementing the MVVM pattern in a WPF app with the MVVM Community toolkit"

Charles

on February 28, 2021 at 5:23 pm said:

After removing all of the code from the "MainWindow.Xaml.cs" file, it no longer compiles because of the missing methods needed for the 'Click' property on the button xaml. Do you have a sample of how to convert the xaml so that it works with the 'Command' property and binding? Specifically, how to reference the view model in the binding. Thanx

Reply↓

bsonnino

on March 1, 2021 at 12:24 pm said:

You can go to my github sample https://github.com/bsonnino/MvvmApp. There are two projects there, the original and the modified, and there you can see how to work with the Command property

Reply↓

Charles

on March 2, 2021 at 12:51 am said:

I have downloaded both folders (net4 and net5). The net5 folder only includes the changes to get the net 4 to compile using .net 5 and still uses the click command. Also, the net5 folder does not include any of the test or DI changes.

Reply↓

Marcelo

on October 8, 2021 at 12:31 am said:

Very good article but the content of your repository does not have the correct code
Reply↓

bsonnino
on December 30, 2021 at 1:44 pm said:

I've checked the repository and both projects work fine (I've tested with VS 2022).

I've just cloned the repository and opened both projects in VS 2022, both ran fine.

Can you give me more details on what happened there ?
Reply↓

Kevin you Lockerby
on October 19, 2021 at 9:34 pm said:

What do you mean the github sample has the original and the modified. No such thing. Did you LOOK AT IT?

You didn't mention what using statements you had to use, references, etc. etc

I spent over an hour and Still couldn't get it to run.

VERY POORLY WRITTEN !
Reply↓

bsonnino
on December 30, 2021 at 1:44 pm said:

I'm sorry you couldn't get it to run. I'll double check the repository
Reply↓

bsonnino
on January 14, 2022 at 11:24 am said:

The code should be there, in the MVVM Project. Sorry for the delay.
Reply↓

Kevin you Lockerby
on October 19, 2021 at 9:36 pm said:

I finally got it to run, the WPF Grid is not bound to the repository.customers. I had to go in and add the bindings.

Awful.
Reply↓

bsonnino

on December 30, 2021 at 1:46 pm said:

I'm glad you could run it. Can you give more details on what you did?

I didn't have any errors, I just cloned the repository and opened both projects in VS 2022, both ran fine

Reply↓

Chris

on November 26, 2021 at 8:10 am said:

Greate article!
Thank you so much!!!

Reply↓

Tide

on January 10, 2022 at 8:39 pm said:

The github version is not like what you have posted here. What Kevin you Lockerby is saying is that the .Net 5 version works but does not have the changes you show in this article. For example if you look at https://github.com/bsonnino/MvvmApp/blob/main/CustomerApp%20-%20Net5/CustomerApp/App.xaml.cs you will see it does not contain any of the code mentioned above in the new App.xaml.cs like Services = ConfigureServices();. Great article and thank you for writing it.

Reply↓

Tide

on January 11, 2022 at 5:36 pm said:

I am trying to post about how the github repo does not have the code shown in this article. Many people have pointed this out but bsonnino is missing what people are saying. The .Net5 version will run bit it does not contain the changes mentioned in this article. The project will run but that is not the point. We were just wanting to see a working app with these changes. If you look at https://github.com/bsonnino/MvvmApp/blob/main/CustomerApp%20-%20Net5/CustomerApp/App.xaml.cs you will see it has none of the code mentioned above. This is true for most everything you talked about above. Hopefully, you can upload the updated solution. Thanks again for writing this article.

Reply↓

bsonnino

on January 14, 2022 at 11:25 am said:

The code should be there, in the MVVM Project

Reply↓

Tide

on January 14, 2022 at 6:53 pm said:

That is great thank you!
Reply↓

Pierre Bernard
on March 5, 2022 at 1:57 pm said:

I stumbled upon this page by chance. I have learned so much more than expected! Of course, no pain no gain, so I started with the .NET 4 project adding whatever frameworks it required to make it work. Then followed your suggestions to move it to a net5.0-windows project including DI and unit testing. Next step for me: move it to .NET 6 and make changes so it becomes a sort of template for my future WPF projects.
Thank you so much.
Reply↓

bsonnino
on March 10, 2022 at 7:44 pm said:

I'm glad you liked it. You will see that the move to .net 6 will be much easier
Reply↓

Jace Malloy
on March 23, 2022 at 1:44 pm said:

I've used MVVM Light, Catel, MvvmCross, Prism and the packaged piece in the DevExpress commercial library, so I thought I'd look into Microsoft's take.

I ran across this example, and picked it up quickly. A lot of the same, since MVVM is just a pattern.

All in all, this is a nice example.

Good job, boss!

—
Jace

p.s. One good thing that came out of Covid, is that I'll be developing remotely in my underwear indefinitely. Starbucks doesn't like it, so at home on the couch seems to work the best. 🙂
Reply↓

## Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

I'm not a robot

reCAPTCHA
Privacy - Terms

Post Comment

← Previous Post

Next Post →