MVVM Cross with Xamarin.Platform – jump start

This document and sample was created to help developers start using MVVM Cross framework with Xamarin.Platform applications. I tried to include many important details to make it easier to understand how MVVM Cross works with Xamarin applications.

I encourage to visit official MVVM Cross website to read more.

This sample has been created with support from In'saneLab team. Thank you!

Prepared by: Daniel Krzyczkowski

Version: 1.0

TABLE OF CONTENTS

1. MVVM Cross framework fundamentals

1.1. Application architecture

MVVM Cross framework was created to make it easier to developers create cross-platform application architecture with Model-View-ViewModel (MVVM) pattern.

MVVM architecture pattern enables developers to separate application business logic and application user interface. Below I included diagram with short description of each part.

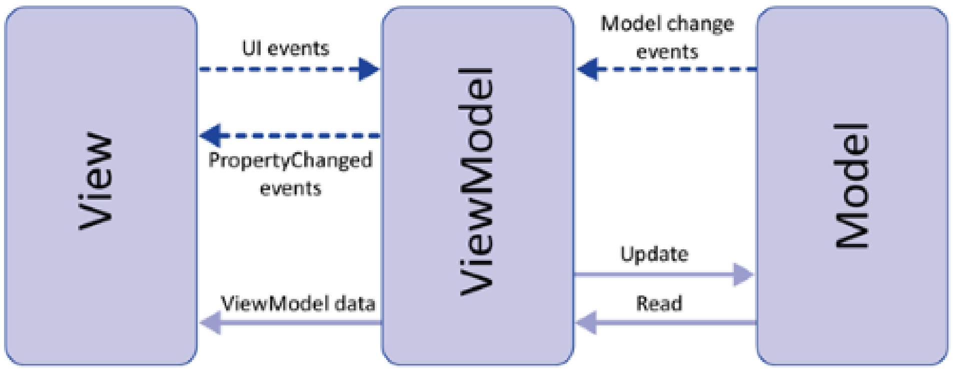


Figure 1 - MVVM architecture pattern

- Model - represents the actual data and/or information we are dealing with (for instance car with brand and color properties)
- View – application user interface
- ViewModel – middle component responsible for data and events exchange between View and Model

When talking about application architecture and MVVM pattern MVVM Cross framework helps a lot. It is much easier for developer to separate different components in the code.

1.2. MVVM Cross fundamentals

MVVM Cross makes it much easier to implement cross-platform code for mobile applications create with Xamarin. Below you can find some crucial components which enhances development.

1.2.1. MvxViewModel

This class can be base for each ViewModel you implement in the code. It provides mechanisms like:

- Constructor base dependency injection:

```
public BaseApplicationMvxViewModel(IMvxNavigationService navigationService)
 {
     _navigationService = navigationService;
  }
```

- INotifyPropertyChanged interface implementation:

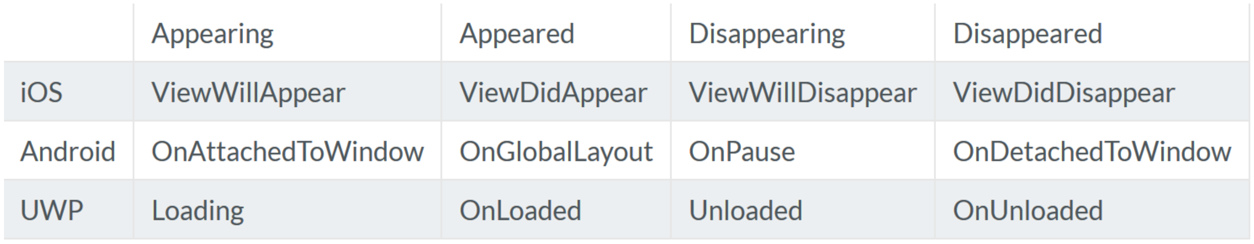
```
private string _userName;
public string Username
{
    get { return _userName; }
    set
    {
        SetProperty(ref _userName, value)
    }
 }
```

- Handling passing parameters:

```
public virtual Task Initialize(TInitParams parameter)
{
    return Task.FromResult(true);
}
```

- Application lifecycle events support – four states are supported:
    - Appearing
    - Appeared
    - Disappearing
    - Disappeared

These states are mapped for each state on the different platform:



	Appearing	Appeared	Disappearing	Disappeared
iOS	ViewWillAppear	ViewDidAppear	ViewWillDisappear	ViewDidDisappear
Android	OnAttachedToWindow	OnGlobalLayout	OnPause	OnDetachedToWindow
UWP	Loading	OnLoaded	Unloaded	OnUnloaded

Figure 2 - MvxViewModel states mapping

- Navigation between ViewModels (using methods from MvxNavigationObject class)

MvxViewModel inherits from MvxNavigationObject described in the next section.

You can see how exactly MvxViewModel class is implemented on GitHub: [MvxViewModel.cs]

1.2.2. MvxNavigationObject

MvxNavigationObject class is responsible for providing functionality to handle navigation changes between ViewModels. Yes – it is worth to mention that with MVVM Cross there is navigation between ViewModels not Pages (or anything else) in the application.

It provides mechanisms like:

- MvxNotifyPropertyChanged class responsible for property changes
- Closing ViewModel:

```
protected bool Close(IMvxViewModel viewModel)
{
   return ChangePresentation(new MvxClosePresentationHint(viewModel));
}
```

- Navigation to new ViewModel:

```
protected bool ShowViewModel<TViewModel>(object parameterValuesObject,
                                        IMvxBundle presentationBundle = null)
   where TViewModel : IMvxViewModel
 {
     return ShowViewModel(
         typeof(TViewModel),
         parameterValuesObject.ToSimplePropertyDictionary(),
         presentationBundle);
 }
```

- Changing presentation (what exactly should be displayed on the screen, we will discuss Presenters later in the document):

```
protected bool ChangePresentation(MvxPresentationHint hint)
{
       MvxTrace.Trace("Requesting presentation change");
       var viewDispatcher = ViewDispatcher;
       if (viewDispatcher != null)
              return viewDispatcher.ChangePresentation(hint);

       return false;
 }
```

You can see how exactly MvxNavigationObject class is implemented on GitHub: [MvxNavigationObject.cs]()

1.2.3. MvxViewPresenter

View Presenters are a key object in the MVVM Cross architecture. You can imagine that View Presenter is like glue between ViewModel and View. As I mentioned before MVVM Cross is based on navigation between ViewModels so View Presenter is responsible for displaying selected ViewModel with correct View in proper for each platform.

Example:

- On iOS to show ViewController modally
- On Android to show selected Fragment
- On UWP display correct page

Each platform has its own implementation of View Presenter:

- Android – MvxAndroidViewPresenter
- iOS – MvxIosViewPresenter
- UWP – MvxWindowsViewPresenter

When you navigate to selected ViewModel, platform specific View Presenter handles displaying View properly.

There is also great feature called MvxPresentationHint. Imagine that when you navigating to selected ViewModel you can add instructions how exactly View connected with this ViewModel should be displayed.

```
public class CustomHint : MvxPresentationHint
{
      public bool ShouldHideAppBar { get; set; }

      public CustomHint(bool shouldHideAppBar)
       {
            ShouldHideAppBar = shouldHideAppBar;
       }
 }
```

Above example shows CustomHint with information to hide application top bar and View Presenter will handle it differently on each platform.

Below code should be invoked in ViewModel to pass CustomHint to View Presenter:

```
ChangePresentation(new CustomHint(true));
```

I created below visualization to make it easier to understand View Presenter concept:

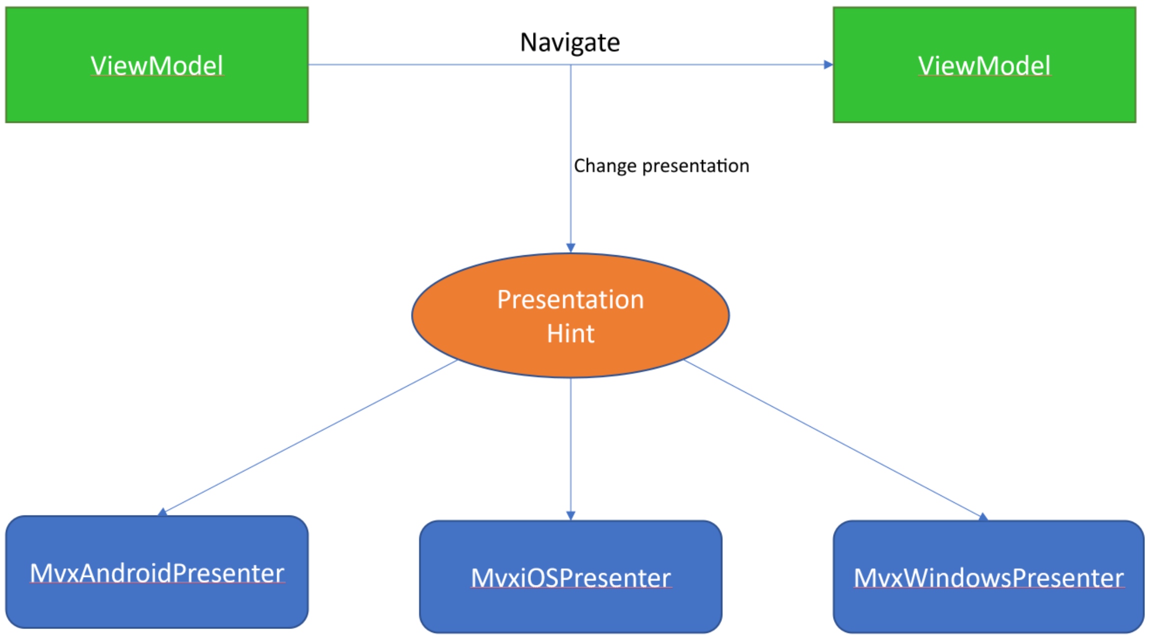


Figure 3 - View Presenter concept diagram

You can see how exactly McxViewPresenter class is implemented on GitHub: [MvxViewPresenter.cs]

1.2.4. MvxNavigationService

MvxNavigationService is new functionality introduced in MVVM Cross 5.0 version. It enables easy navigation between ViewModels and is more testable. Main advantage of using MvxNavigation service is full async support.

MvxNavigationService supports not only methods to show ViewModels but also even handlers connected with navigation:

- BeforeNavigateHandler
- AfterNavigateEventHandler
- BeforeCloseEventHandler
- AfterCloseEventHandler

There are also methods for navigation like:

- Close
- Navigate (including parameter to pass)
- CanNavigate

Below you can find example how easily MvxNavigationService can be added to your ViewModel class:

```
public BaseApplicationMvxViewModel(IMvxNavigationService navigationService)
{
            _navigationService = navigationService;
}
```

Now you can navigate to other ViewModel and pass some parameter:

```
await _navigationService.Navigate<HomeViewModel, UserData>(userData);
```

You can see how exactly MvxNavigationService class is implemented on GitHub: [MvxNavigationService.cs]

1.2.5. Mvx – Dependency injection

MVVM Cross has its own dependency container represented by static class called Mvx. It provides mechanism based on reflection which automatically resolves parameters when new object is being created.

Below I pasted simple example how does it work:

```
public MainMvxViewModel(IMvxNavigationService navigationService)
        {
            _navigationService = navigationService;
        }
```

We can create BaseApplicationMvxViewModel object using below construction:

```
Mvx.IocConstruct<BaseApplicationMvxViewModel>();
```

- You can register specific types in Mcx container like below:

  ```
  Mvx.RegisterType<IAuthenticationService, AuthenticationService>();
  ```

- Then you can get specific type instance:

  ```
  Mvx.Resolve<IAuthenticationService>()
  ```

- You can also register type as singleton:

  ```
  Mvx.ConstructAndRegisterSingleton<IAuthenticationService, AuthenticationService>();
  ```

You can read more about Dependency Injection with in MVVM Cross
in the official documentation website.

2. APPLICATION SOLUTION SETUP

Important:

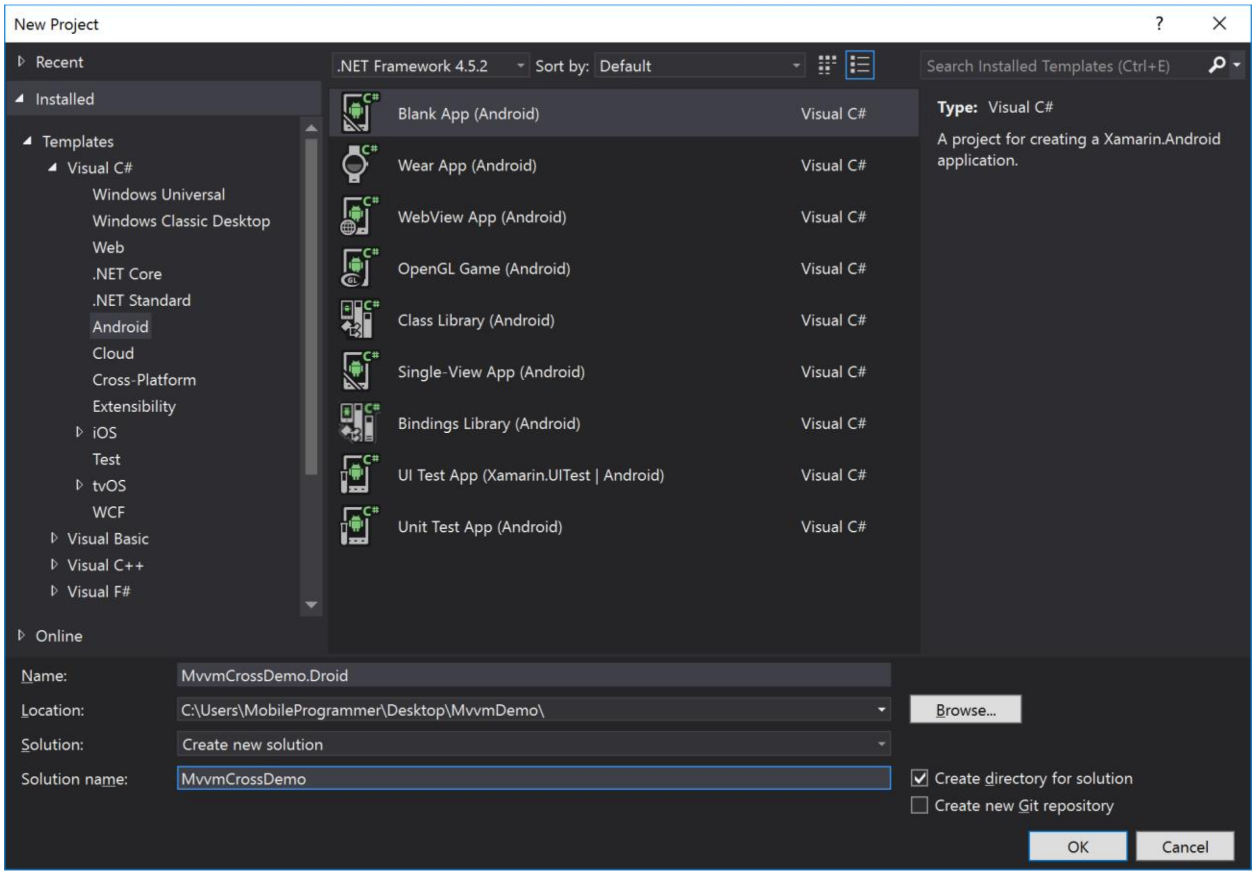
This sample was created with Visual Studio 2017 on Windows 10 (to provide support for UWP). You can use it on MAC but then UWP project will be unloaded.

- MvvmCross version: 5.0 and above

Source code of this application is available on my GitHub.

2.1. Add Xamarin.Android project

- Project name: MvvmCrossDemo.Droid
- Project template: Blank App (Android)
- Solution name: MvvmCrossDemo


New Project
Recent
Installed
Templates
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Android
Cloud
Cross-Platform
Extensibility
iOS
Test
tvOS
WCF
Visual Basic
Visual C++
Visual F#
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Templates (Ctrl+E)
Blank App (Android) Visual C#
Wear App (Android) Visual C#
WebView App (Android) Visual C#
OpenGL Game (Android) Visual C#
Class Library (Android) Visual C#
Single-View App (Android) Visual C#
Bindings Library (Android) Visual C#
UI Test App (Xamarin.UITest | Android) Visual C#
Unit Test App (Android) Visual C#
Browse...
OK
Cancel

Type: Visual C#

A project for creating a Xamarin.Android application.

Name: MvvmCrossDemo.Droid

Location: C:\Users\MobileProgrammer\Desktop\MvvmDemo\

Solution: Create new solution

Solution name: MvvmCrossDemo

☑ Create directory for solution

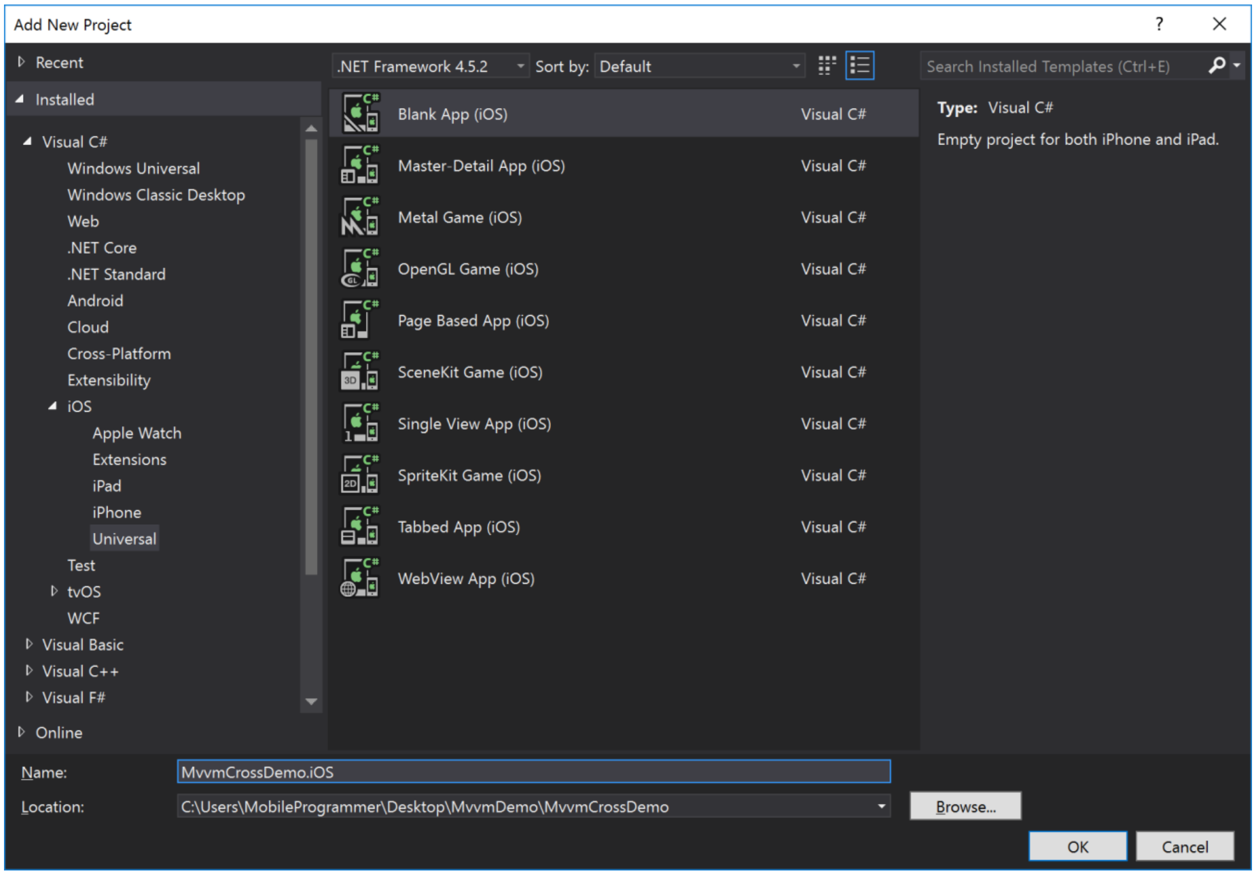
☐ Create new Git repository

Figure 4 - Solution setup

- Compile using Android version: 7.1 (Nougat)
- Minimum Android version: 4.4 (API level 19 - KitKat)
- Target Android version: Use Compile using SDK version

2.2. Add Xamarin.iOS project

- Project name: MvvmCrossDemo.Ios
- Project template: Blank App (iOS)



Add New Project
Recent
Installed
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Android
Cloud
Cross-Platform
Extensibility
iOS
Apple Watch
Extensions
iPad
iPhone
Universal
Test
tvOS
WCF
Visual Basic
Visual C++
Visual F#
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Templates (Ctrl+E)
Blank App (iOS) Visual C#
Master-Detail App (iOS) Visual C#
Metal Game (iOS) Visual C#
OpenGL Game (iOS) Visual C#
Page Based App (iOS) Visual C#
SceneKit Game (iOS) Visual C#
Single View App (iOS) Visual C#
SpriteKit Game (iOS) Visual C#
Tabbed App (iOS) Visual C#
WebView App (iOS) Visual C#
Browse...
OK
Cancel

Type: Visual C#

Empty project for both iPhone and iPad.

Name: MvvmCrossDemo.iOS

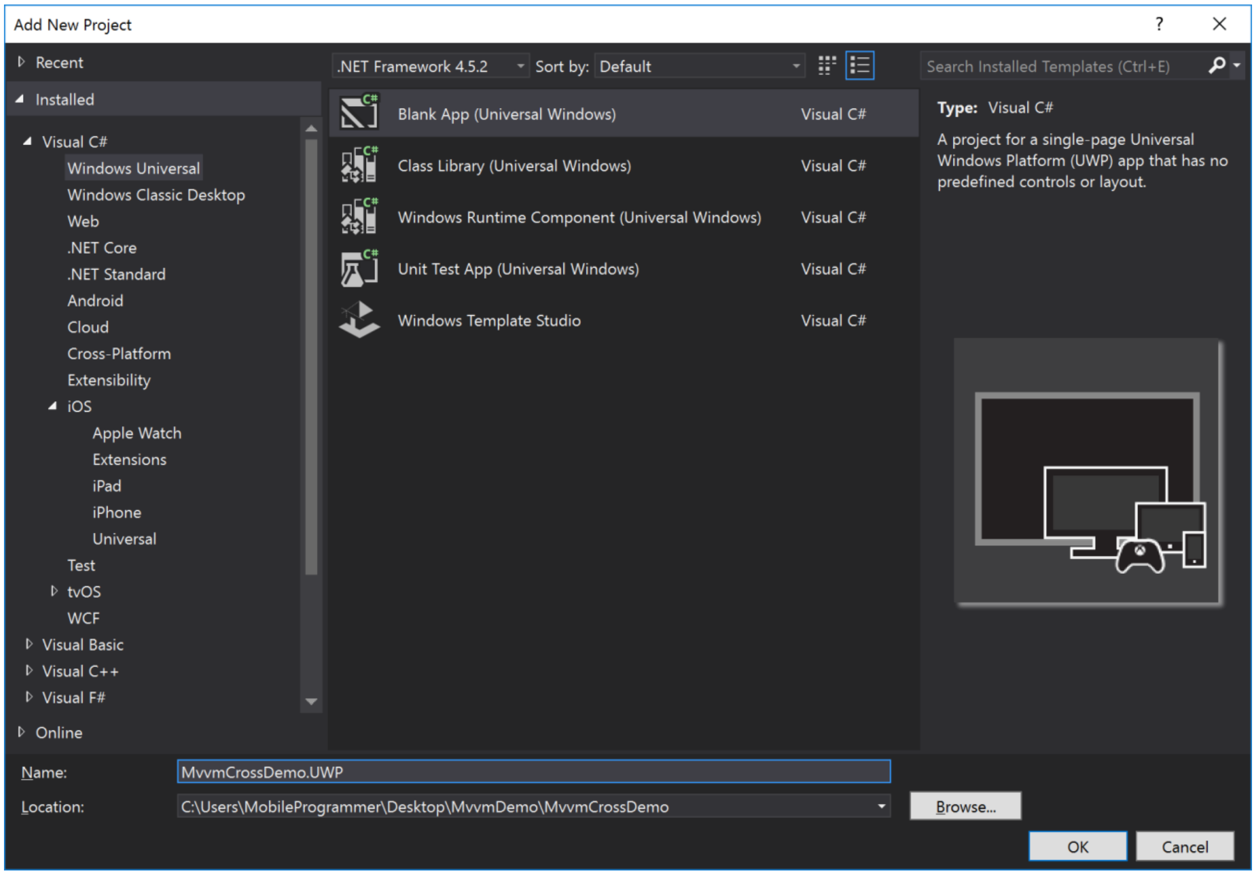
Location: C:\Users\MobileProgrammer\Desktop\MvvmDemo\MvvmCrossDemo

Figure 5 - Including iOS project

- Deployment iOS target: 9.0

2.3. Add Universal Windows Platform project

- Project name: MvvmCrossDemo.UWP
- Project template: Blank App (Universal Windows)



Type: Visual C#

A project for a single-page Universal Windows Platform (UWP) app that has no predefined controls or layout.

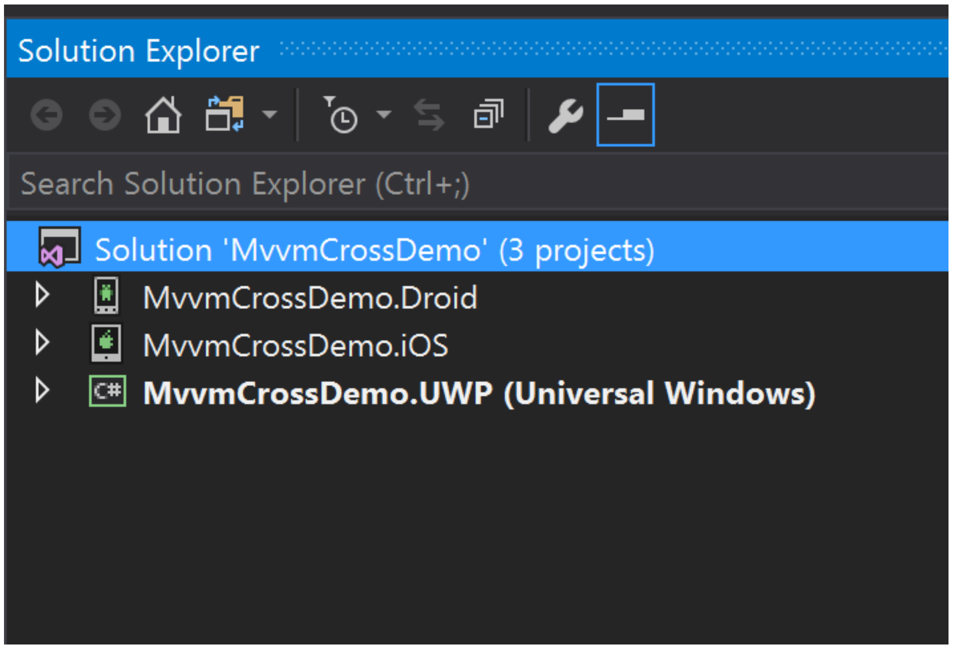
Name: MvvmCrossDemo.UWP

Location: C:\Users\MobileProgrammer\Desktop\MvvmDemo\MvvmCrossDemo

Figure 6 - Including UWP project

- Target version: Windows 10 Creators Update (10.0; Build 15053)
- Min version: Windows 10 Anniversary Update (10.0; Build 14393)

Current solution structure should look like on the below figure:

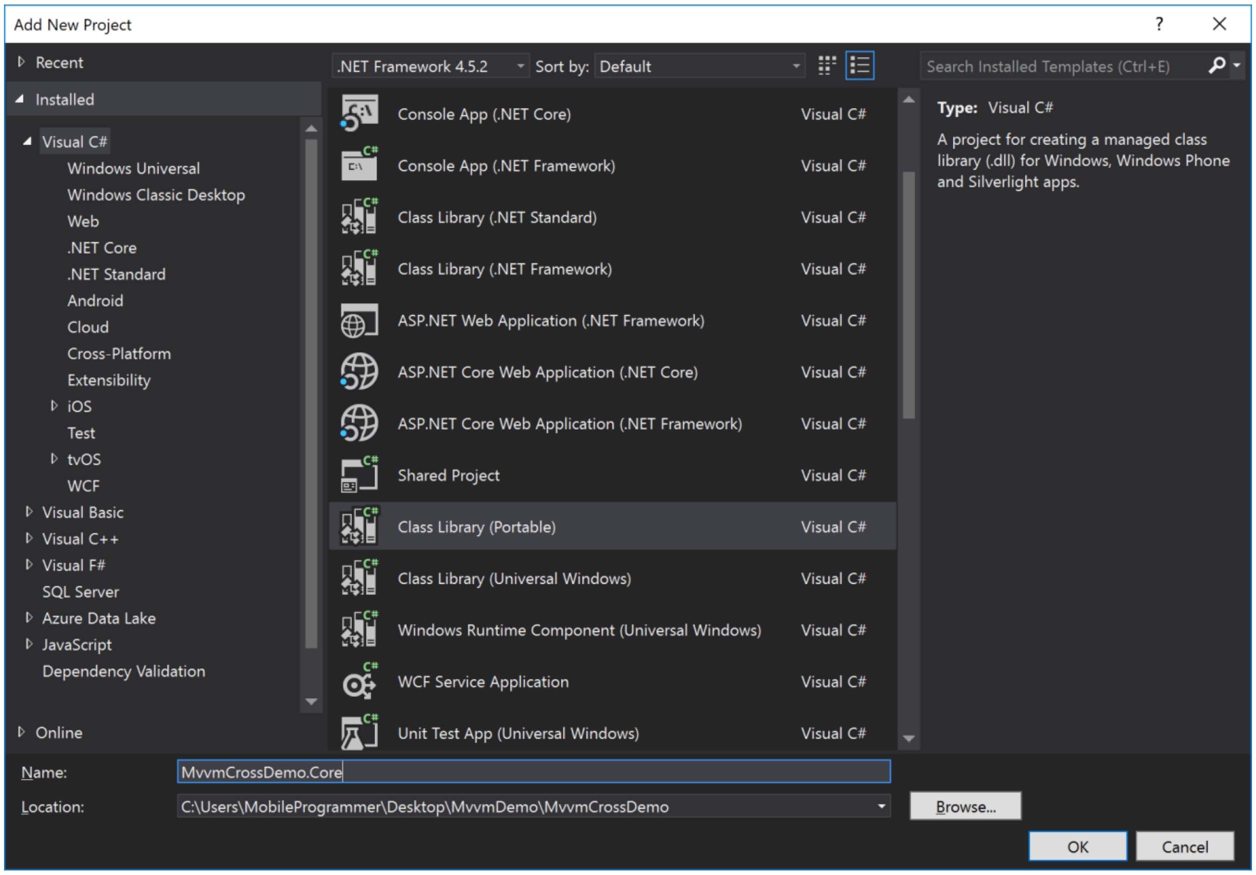


Solution 'MvvmCrossDemo' (3 projects)
MvvmCrossDemo.Droid
MvvmCrossDemo.iOS
MvvmCrossDemo.UWP (Universal Windows)

Figure 7 - Solution structure

2.4. Add Core project

- Project name: MvvmCrossDemo.Core
- Project template: Class Library (Portable)


Add New Project
?
Recent
Installed
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Android
Cloud
Cross-Platform
Extensibility
iOS
Test
tvOS
WCF
Visual Basic
Visual C++
Visual F#
SQL Server
Azure Data Lake
JavaScript
Dependency Validation
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Templates (Ctrl+E)
Console App (.NET Core) Visual C#
Console App (.NET Framework) Visual C#
Class Library (.NET Standard) Visual C#
Class Library (.NET Framework) Visual C#
ASP.NET Web Application (.NET Framework) Visual C#
ASP.NET Core Web Application (.NET Core) Visual C#
ASP.NET Core Web Application (.NET Framework) Visual C#
Shared Project Visual C#
Class Library (Portable) Visual C#
Class Library (Universal Windows) Visual C#
Windows Runtime Component (Universal Windows) Visual C#
WCF Service Application Visual C#
Unit Test App (Universal Windows) Visual C#
Browse...
OK
Cancel

Type: Visual C#

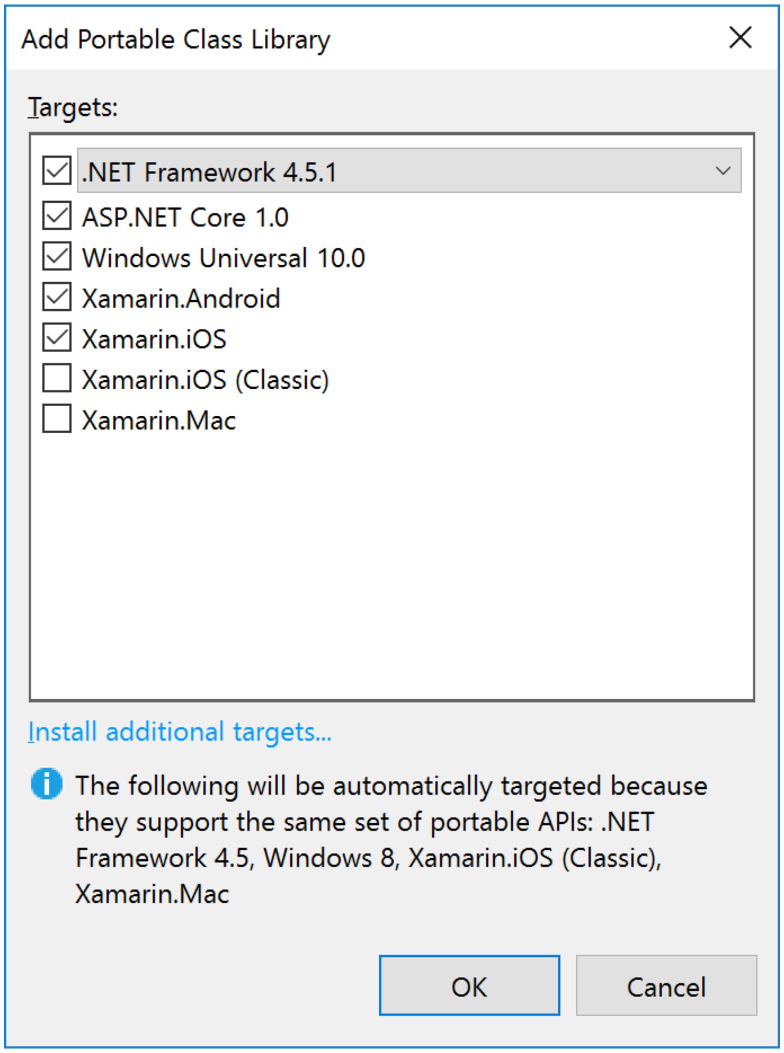
A project for creating a managed class library (.dll) for Windows, Windows Phone and Silverlight apps.

Name: MvvmCrossDemo.Core

Location: C:\Users\MobileProgrammer\Desktop\MvvmDemo\MvvmCrossDemo

Figure 8 - Including Core project

Portable Class Library profile should be selected:



Add Portable Class Library

Targets:

- [✓] .NET Framework 4.5.1
- [✓] ASP.NET Core 1.0
- [✓] Windows Universal 10.0
- [✓] Xamarin.Android
- [✓] Xamarin.iOS
- [ ] Xamarin.iOS (Classic)
- [ ] Xamarin.Mac

Install additional targets...

The following will be automatically targeted because they support the same set of portable APIs: .NET Framework 4.5, Windows 8, Xamarin.iOS (Classic), Xamarin.Mac

Figure 9 - PCL profile configuration

Final solution structure should look like on the below figure:

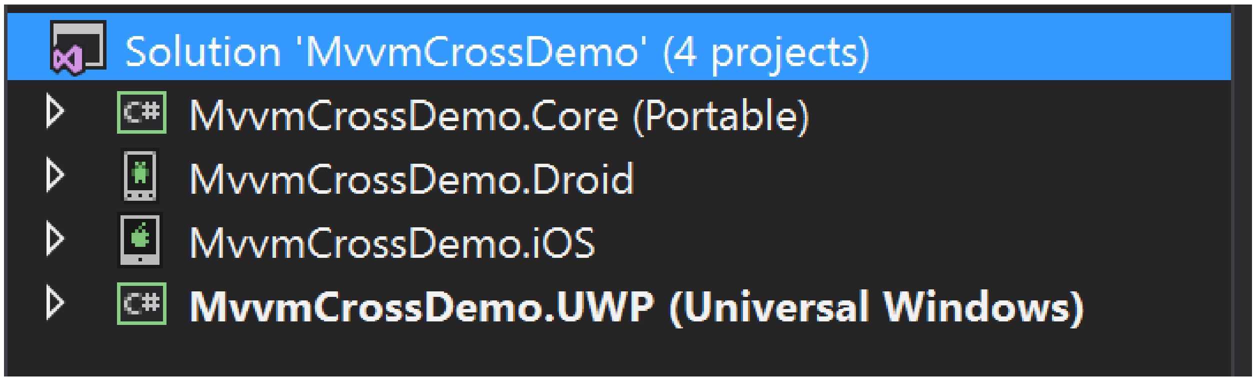

Solution 'MvvmCrossDemo' (4 projects)
MvvmCrossDemo.Core (Portable)
MvvmCrossDemo.Droid
MvvmCrossDemo.iOS
MvvmCrossDemo.UWP (Universal Windows)

Figure 10 - Final solution structure

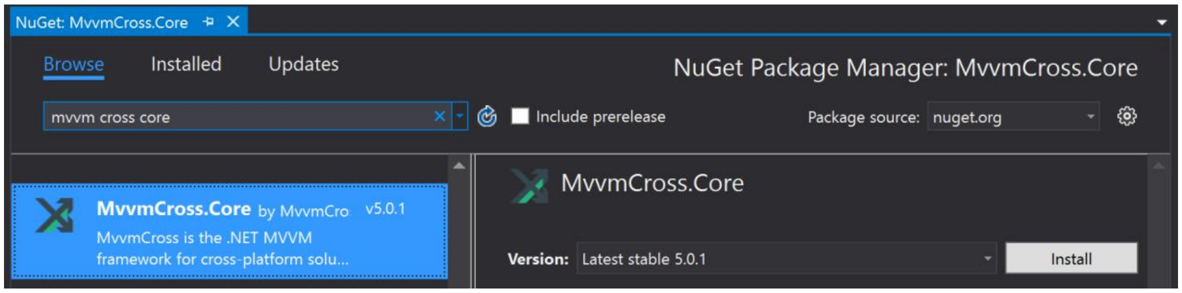
3. MVVM CROSS FRAMEWORK SETUP

3.1. NuGet packages configuration

3.1.1. Core project setup

To configure MVVM Cross for Core project below NuGet packages are required:

- NuGet package to add: MvvmCross.Core

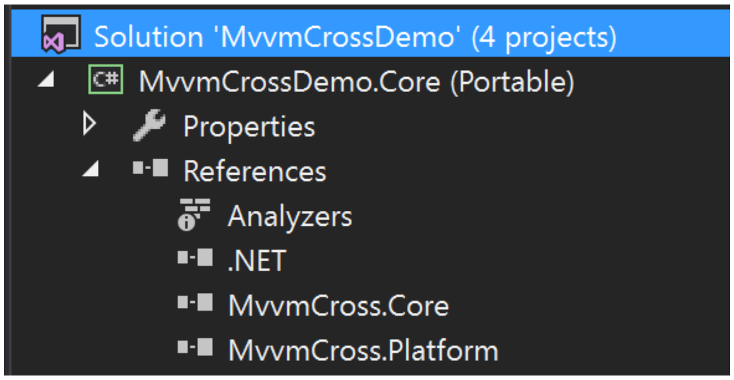


Search: mvvm cross core
Package source: nuget.org
MvvmCross.Core by MvvmCro v5.0.1
MvvmCross is the .NET MVVM framework for cross-platform solu...
MvvmCross.Core
Version: Latest stable 5.0.1

Figure 11 – MVVM Cross setup for Core project

Once NuGet is added you should see two references added:

- MvvmCross.Core
- MvvmCross.Platform



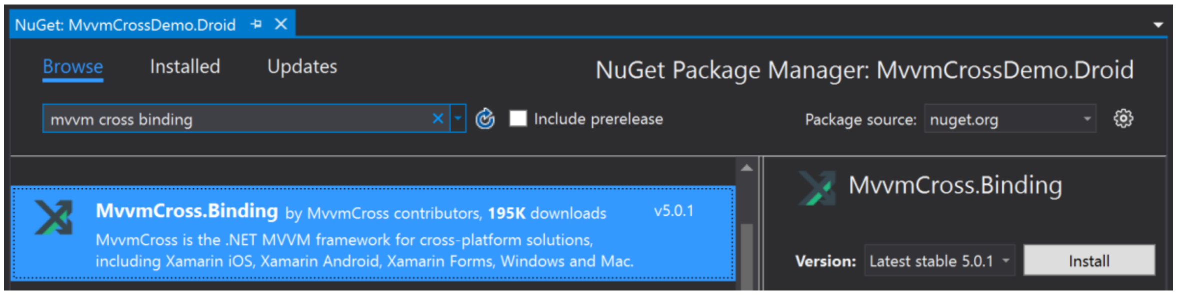
Solution 'MvvmCrossDemo' (4 projects)
MvvmCrossDemo.Core (Portable)
Properties
References
Analyzers
.NET
MvvmCross.Core
MvvmCross.Platform

Figure 12 - Core project MVVM Cross references

3.1.2. Xamarin.Android project setup

To configure MvvmCross for Xamarin.Android project below NuGet packages are required:

- NuGet package to add: MvvmCross.Binding


NuGet: MvvmCrossDemo.Droid
Browse
Installed
Updates
NuGet Package Manager: MvvmCrossDemo.Droid
Include prerelease
Install

mvvm cross binding

Package source: nuget.org

MvvmCross.Binding by MvvmCross contributors, **195K** downloads v5.0.1
MvvmCross is the .NET MVVM framework for cross-platform solutions, including Xamarin iOS, Xamarin Android, Xamarin Forms, Windows and Mac.

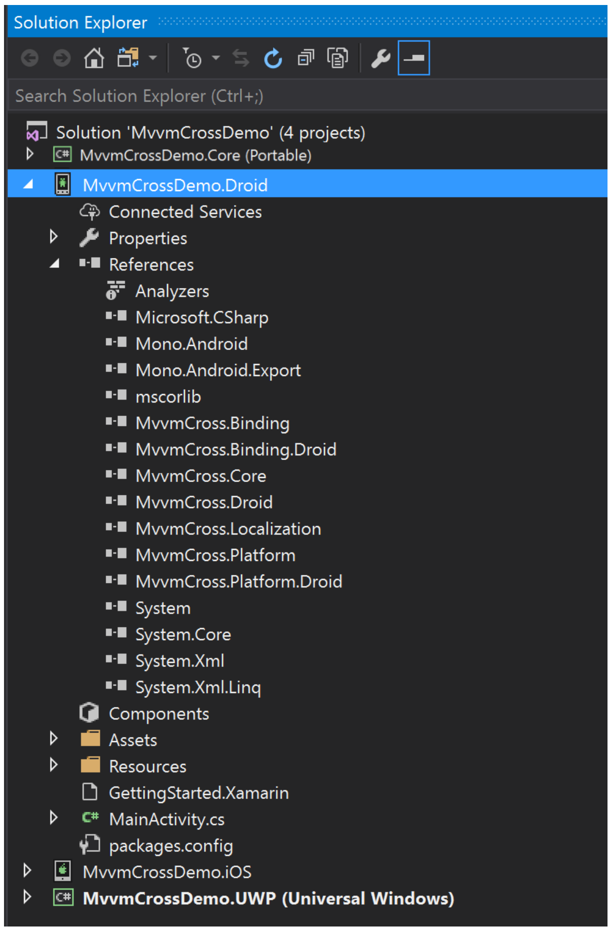
MvvmCross.Binding

Version: Latest stable 5.0.1

Figure 13 - MVVM Cross setup for Xamarin.Android project

Once NuGet is added you should see below references added:

- MvvmCross.Binding
- MvvmCross. Binding.Droid
- MvvmCross.Core
- MvvmCross.Core.Droid
- MvvmCross.Localization
- MvvmCross.Platform
- MvvmCross.Platform.Droid



Solution 'MvvmCrossDemo' (4 projects)
MvvmCrossDemo.Core (Portable)
MvvmCrossDemo.Droid
Connected Services
Properties
References
Analyzers
Microsoft.CSharp
Mono.Android
Mono.Android.Export
mscorlib
MvvmCross.Binding
MvvmCross.Binding.Droid
MvvmCross.Core
MvvmCross.Droid
MvvmCross.Localization
MvvmCross.Platform
MvvmCross.Platform.Droid
System
System.Core
System.Xml
System.Xml.Linq
Components
Assets
Resources
GettingStarted.Xamarin
MainActivity.cs
packages.config
MvvmCrossDemo.iOS
MvvmCrossDemo.UWP (Universal Windows)

Figure 14 - Xamarin.Android project MVVM Cross references

We need to add some additional NuGet packages listed below:

- Xamarin.Android.Support.v4
- Xamarin.Android.Support.v7.AppCompat
- MvvmCross.Droid.Support.V4
- MvvmCross.Droid.Support.V7.AppCompat

3.1.3. Xamarin.iOS project setup

To configure MvvmCross for Xamarin.iOS project below NuGet packages are required:

- NuGet package to add: MvvmCross.Binding

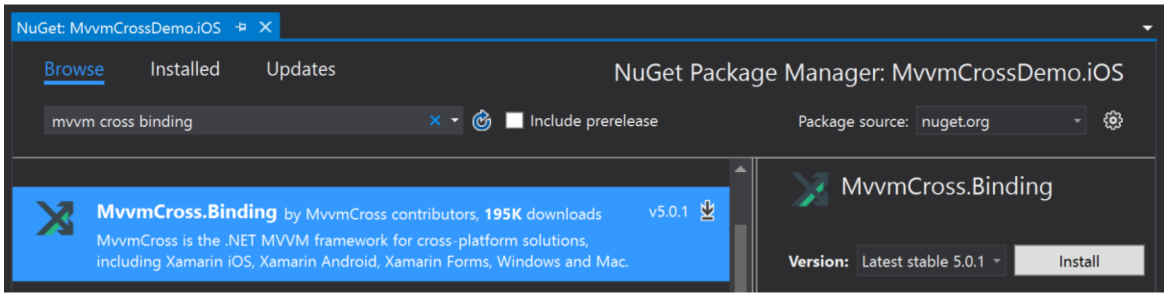
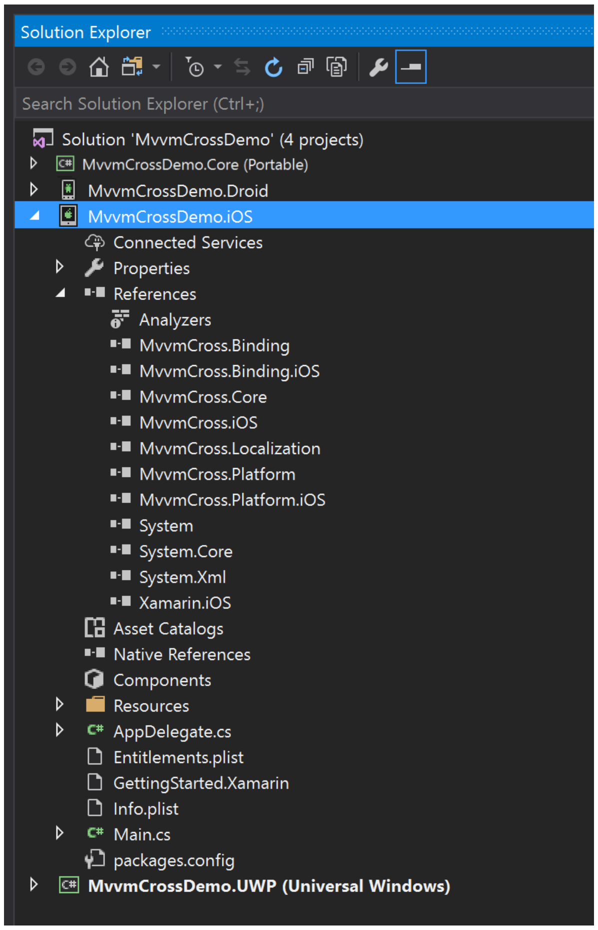

NuGet: MvvmCrossDemo.iOS
Browse
Installed
Updates
NuGet Package Manager: MvvmCrossDemo.iOS
mvvm cross binding
Include prerelease
Package source: nuget.org
MvvmCross.Binding by MvvmCross contributors, 195K downloads
v5.0.1
MvvmCross is the .NET MVVM framework for cross-platform solutions, including Xamarin iOS, Xamarin Android, Xamarin Forms, Windows and Mac.
MvvmCross.Binding
Version: Latest stable 5.0.1
Install

Figure 15 - MVVM Cross setup for Xamarin.iOS project

Once NuGet is added you should see below references added:

- MvvmCross.Binding
- MvvmCross. Binding.iOS
- MvvmCross.Core
- MvvmCross.iOS
- MvvmCross.Localization
- MvvmCross.Platform
- MvvmCross.Platform.iOS



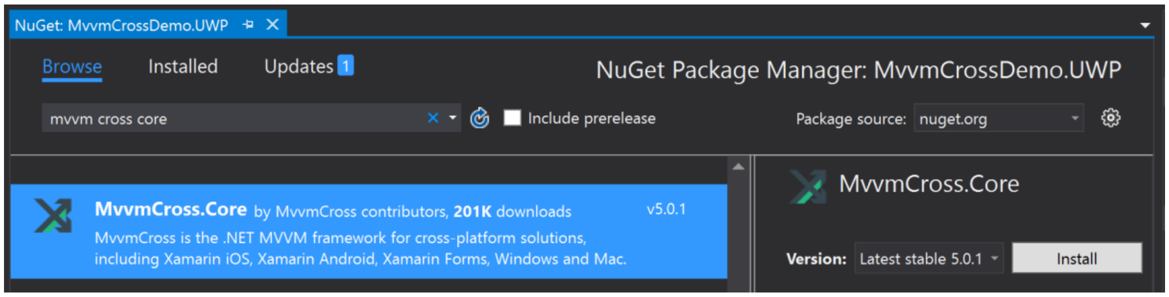
```
Solution 'MvvmCrossDemo' (4 projects)
  MvvmCrossDemo.Core (Portable)
  MvvmCrossDemo.Droid
  MvvmCrossDemo.iOS
    Connected Services
    Properties
    References
      Analyzers
      MvvmCross.Binding
      MvvmCross.Binding.iOS
      MvvmCross.Core
      MvvmCross.iOS
      MvvmCross.Localization
      MvvmCross.Platform
      MvvmCross.Platform.iOS
      System
      System.Core
      System.Xml
      Xamarin.iOS
    Asset Catalogs
    Native References
    Components
    Resources
    AppDelegate.cs
    Entitlements.plist
    GettingStarted.Xamarin
    Info.plist
    Main.cs
    packages.config
  MvvmCrossDemo.UWP (Universal Windows)
```

Figure 16 - Xamarin.iOS project MVVM Cross references

3.1.4. Universal Windows Application (UWP) project setup

To configure MvvmCross for UWP project below NuGet packages are required:

- NuGet package to add: MvvmCross.Core



NuGet: MvvmCrossDemo.UWP

Browse Installed Updates 1

NuGet Package Manager: MvvmCrossDemo.UWP

mvvm cross core

Include prerelease

Package source: nuget.org

MvvmCross.Core by MvvmCross contributors, **201K** downloads v5.0.1
MvvmCross is the .NET MVVM framework for cross-platform solutions, including Xamarin iOS, Xamarin Android, Xamarin Forms, Windows and Mac.

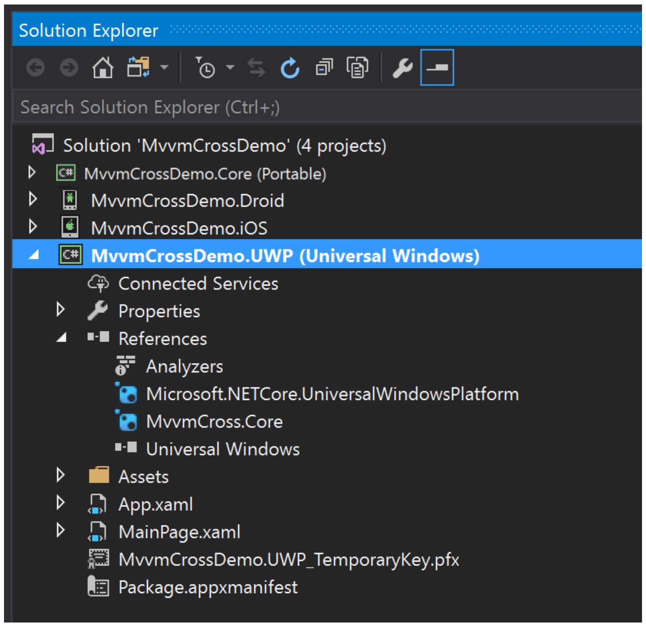
MvvmCross.Core

Version: Latest stable 5.0.1 Install

Figure 17 - MVVM Cross setup for UWP project

Once NuGet is added you should see below references added:

- MvvmCross.Core



Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'MvvmCrossDemo' (4 projects)
  MvvmCrossDemo.Core (Portable)
  MvvmCrossDemo.Droid
  MvvmCrossDemo.iOS
  MvvmCrossDemo.UWP (Universal Windows)
    Connected Services
    Properties
    References
      Analyzers
      Microsoft.NETCore.UniversalWindowsPlatform
      MvvmCross.Core
      Universal Windows
    Assets
    App.xaml
    MainPage.xaml
    MvvmCrossDemo.UWP_TemporaryKey.pfx
    Package.appxmanifest
```

Figure 18 - UWP project MVVM Cross references

3.2. Application user interface

Sample application will consist of three pages:

- Login page – where user can type username, password and sign in
- Main page – where user name is displayed with option to navigate to details page
- Details page – With simple label and image

We will also use below three graphic assets in the application:

Figure 19 - xamarin monkey graphic asset

Figure 20 - MVVM Cross graphics asset

Figure 21 – In'saneLab graphic asset

For each application, we also have splash screen image:

Figure 22 - Splash Screen image

3.2.1. Xamarin.Android application user interface implementation

Let’s start from creating Xamarin.Android application user interface. Structure is exactly the same like for ordinary Android application – all views are kept in Resources-layout folder.

Below you can find screenshots of each screen from the Xamarin.Android application.

- SplashScreen view will look like on figure below:

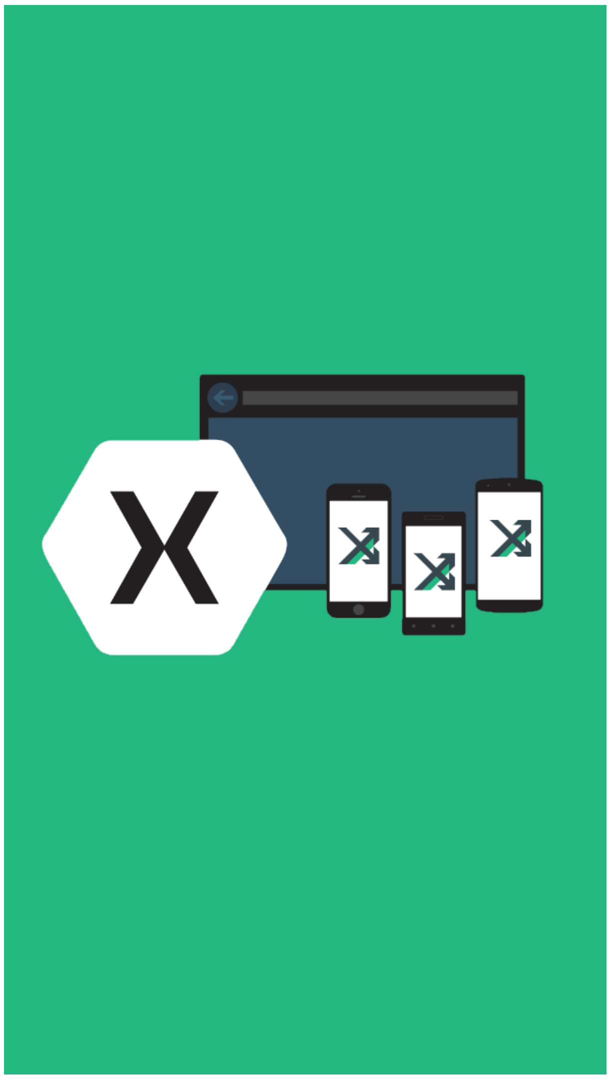


Figure 23 - Xamarin.Android Splash Screen

- LoginActivity view will look like on figure below:

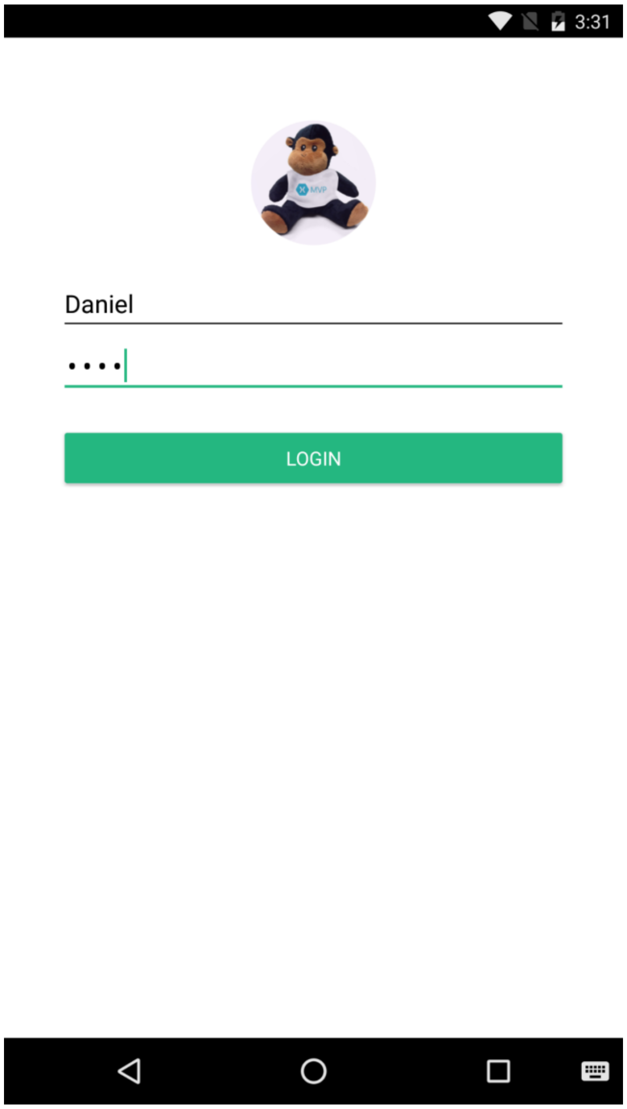


Figure 24 - Xamarin.Android Login Activity

- HomeFragment view will look like on figure below:

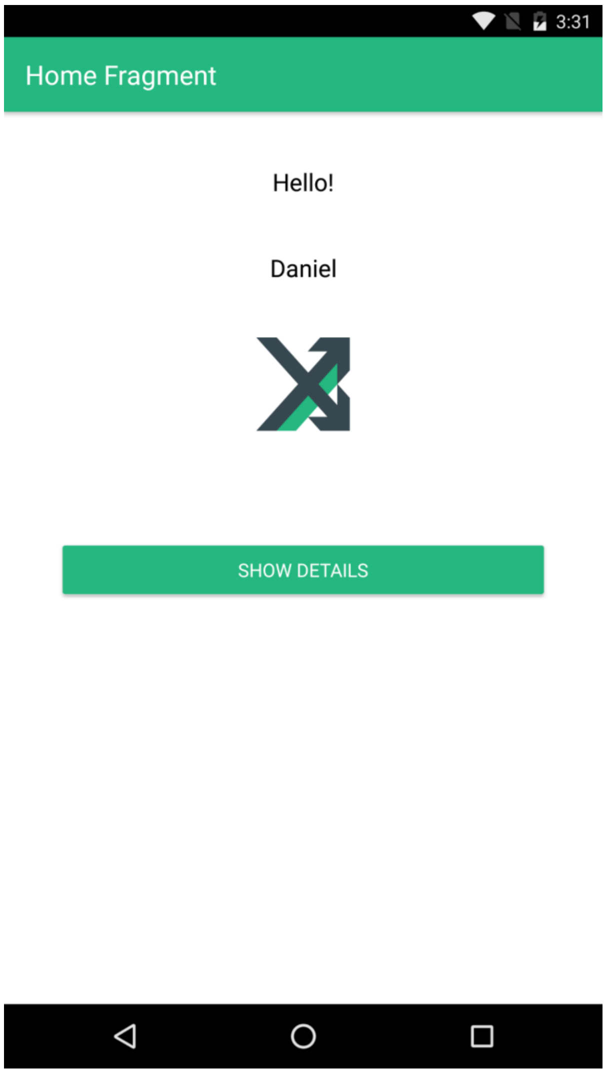
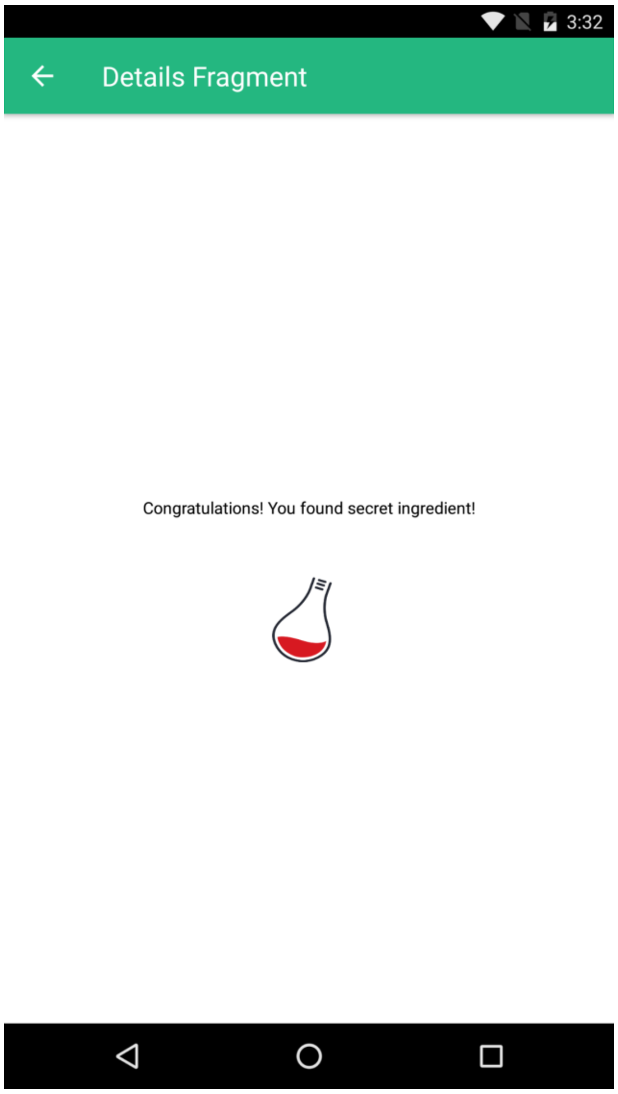


Figure 25 - Xamarin.Android Home Fragment

- DetailsFragment view will look like on figure below:



Congratulations! You found secret ingredient!

Figure 26 - Xamarin.Android Details Fragment

3.2.2. Xamarin.iOS application user interface implementation

To create iOS application user interface we will use FluentLayout library - fluent API for creating constraint-based layouts in Xamarin.iOS. Official GitHub page provides information and samples: https://github.com/FluentLayout/Cirrious.FluentLayout

- NuGet package to add Cirrious.FluentLayout: https://www.nuget.org/packages/Cirrious.FluentLayout

Below you can find screenshots of each screen from the Xamarin.iOS application.

- Splash Screen view will look like on figure below:

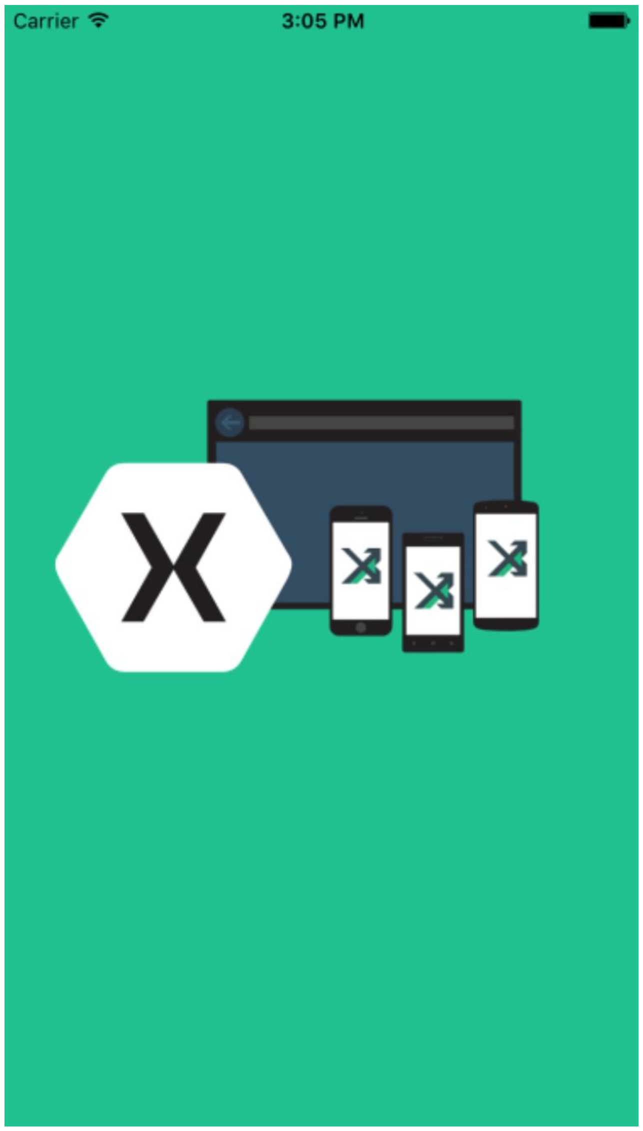


Figure 27 - Xamarin.iOS application Splash Screen

- LoginViewController view will look like on figure below:

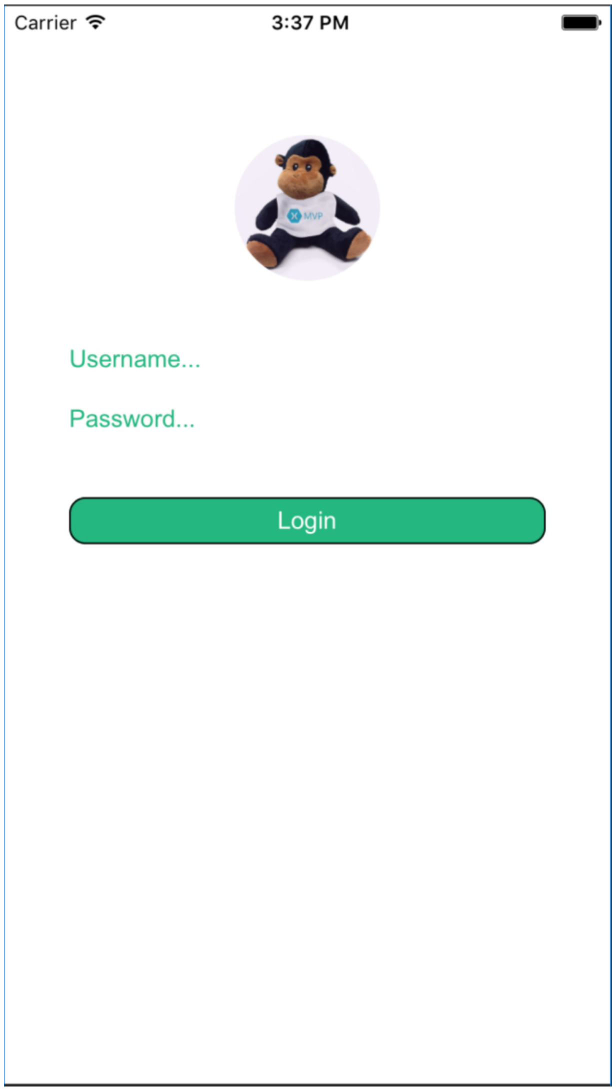


Figure 28 - Xamarin.iOS application Login ViewController

- HomeViewController view will look like on figure below:

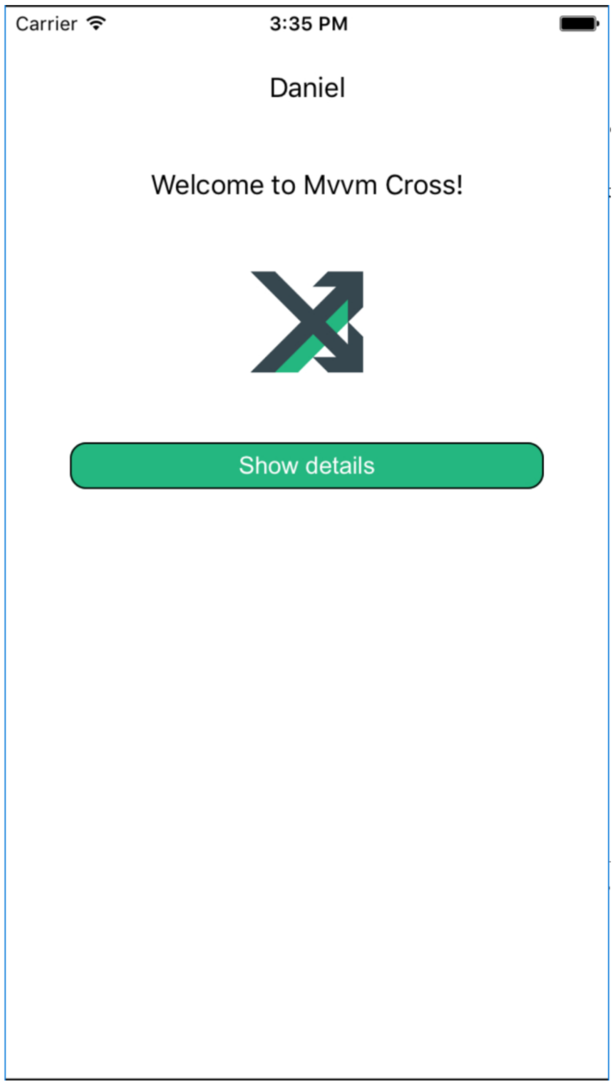


Figure 29 - Xamarin.iOS application Home ViewController

- DetailsViewController view will look like on figure below:

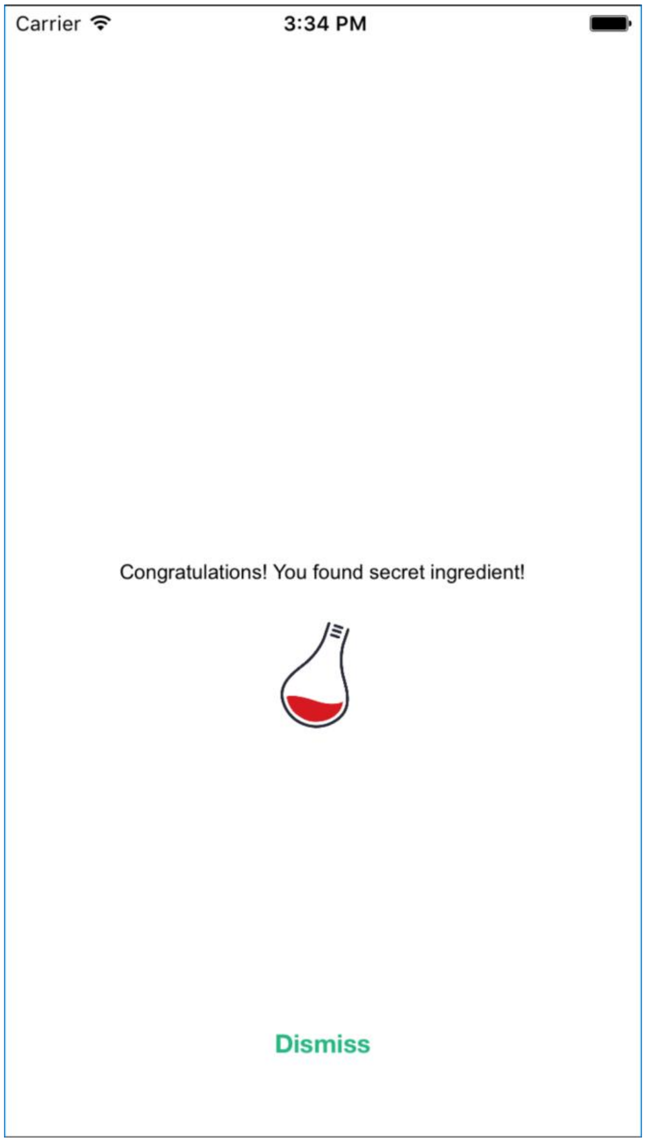


Figure 30 - Xamarin.iOS application Details ViewController

3.2.3. Xamarin.UWP application user interface implementation

Below you can find screenshots of each screen from the UWP application.

- SplashScreen will look like on figure below:

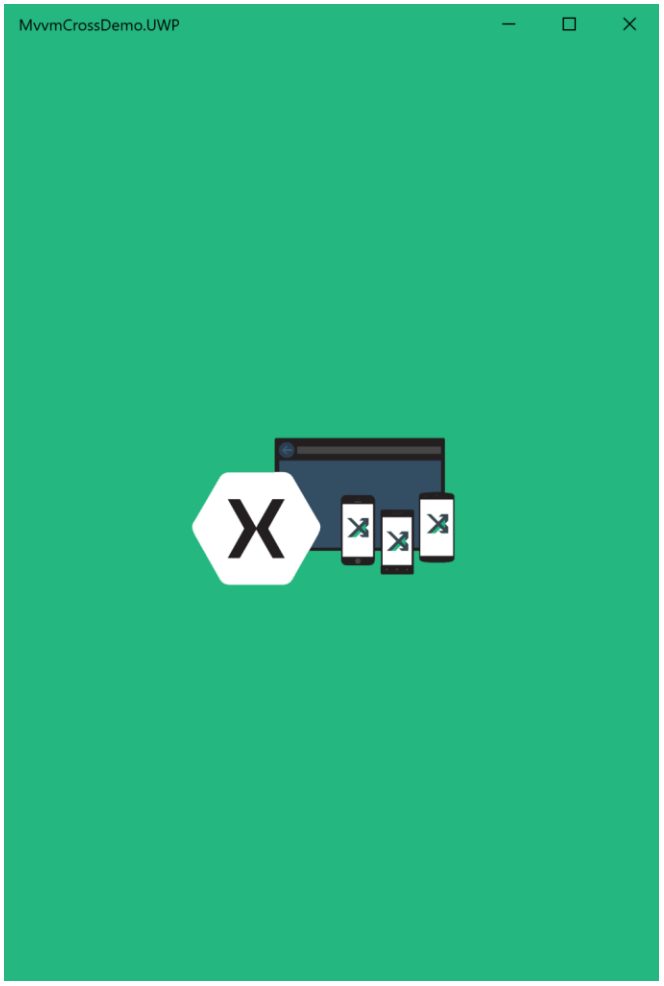


Figure 31 - UWP application splash screen

- LoginPage view will look like on figure below:

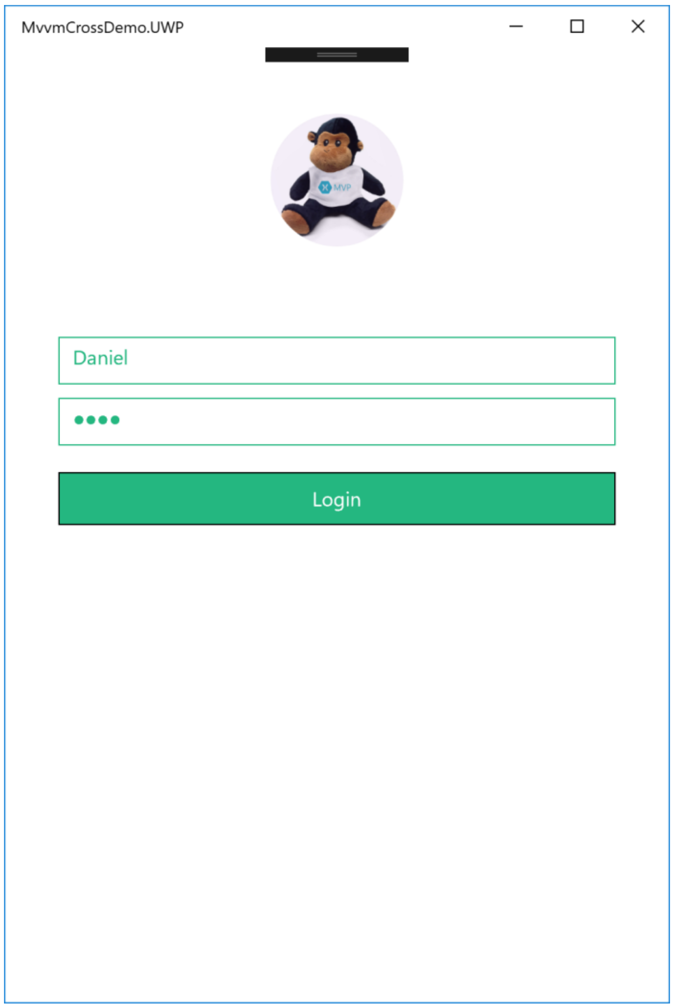


Figure 32 - UWP application Login Page

- HomePage view will look like on figure below:

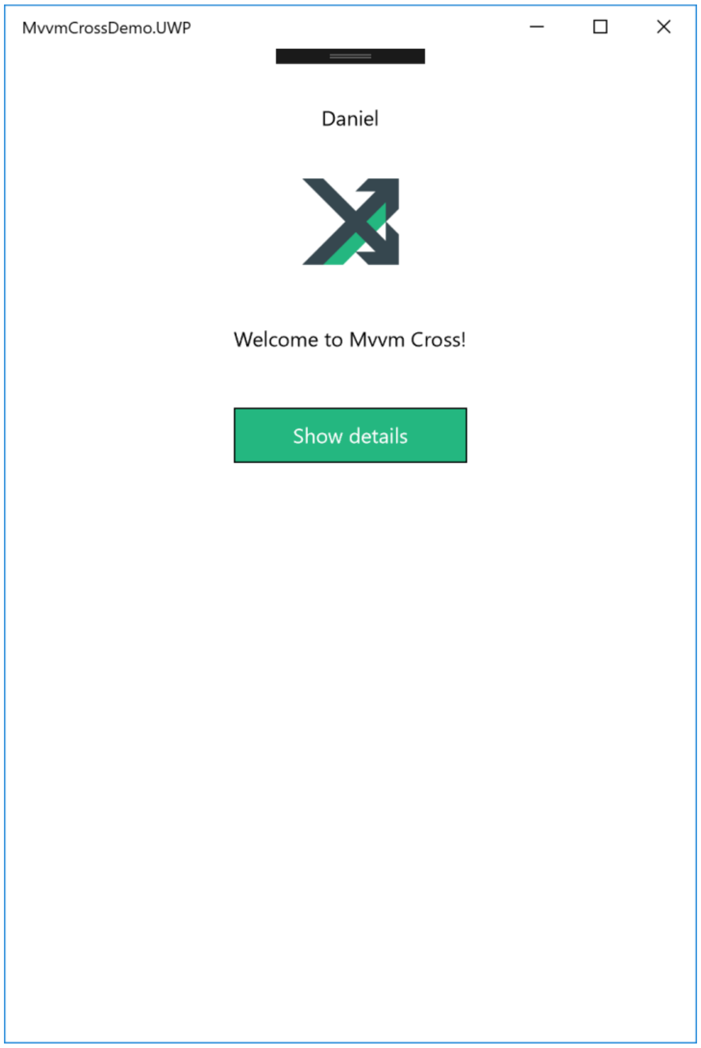
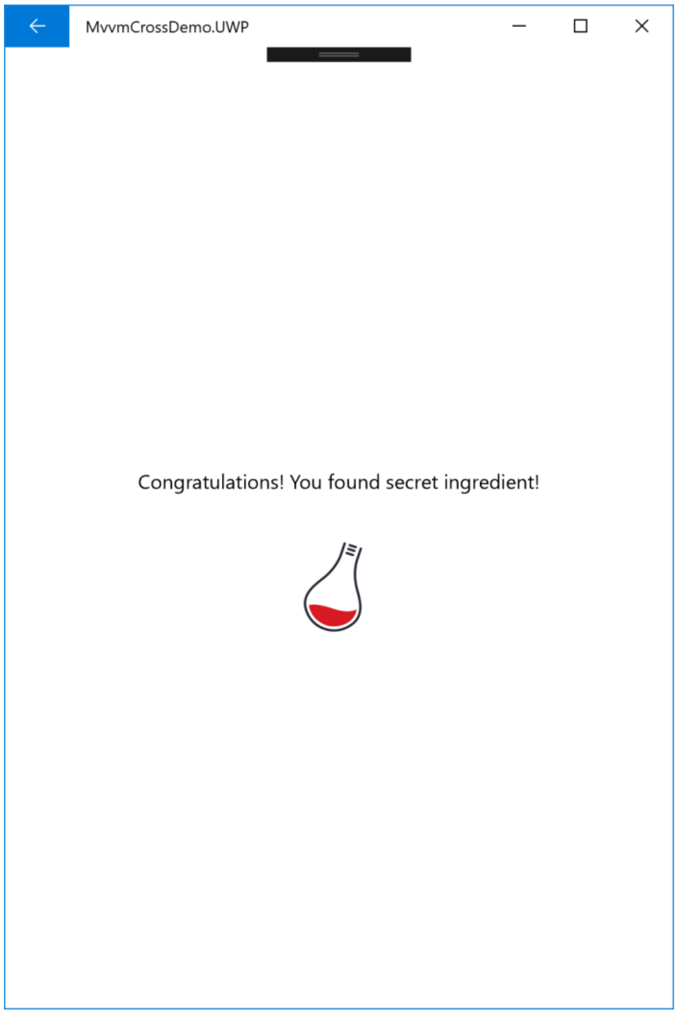


Figure 33 - UWP application Home Page

- DetailsPage view will look like on figure below:



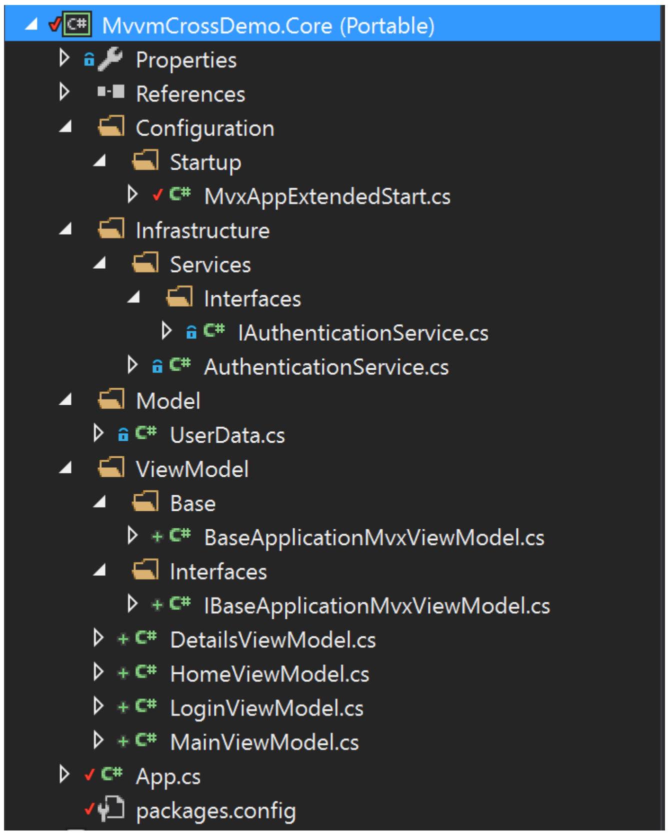
Congratulations! You found secret ingredient!

Figure 34 - UWP application Details Page

3.3. Core project structure

In this section I would like to discuss Core project structure and key elements.

On the figure below you can see how project structure template looks like. It has some crucial elements which we will discuss in this section.



```
MvvmCrossDemo.Core (Portable)
  Properties
  References
  Configuration
    Startup
      MvxAppExtendedStart.cs
  Infrastructure
    Services
      Interfaces
        IAuthenticationService.cs
      AuthenticationService.cs
  Model
    UserData.cs
  ViewModel
    Base
      BaseApplicationMvxViewModel.cs
    Interfaces
      IBaseApplicationMvxViewModel.cs
    DetailsViewModel.cs
    HomeViewModel.cs
    LoginViewModel.cs
    MainViewModel.cs
  App.cs
  packages.config
```

Figure 35 - Core project structure

Let me walk through all these folders so you can understand what is happening here.

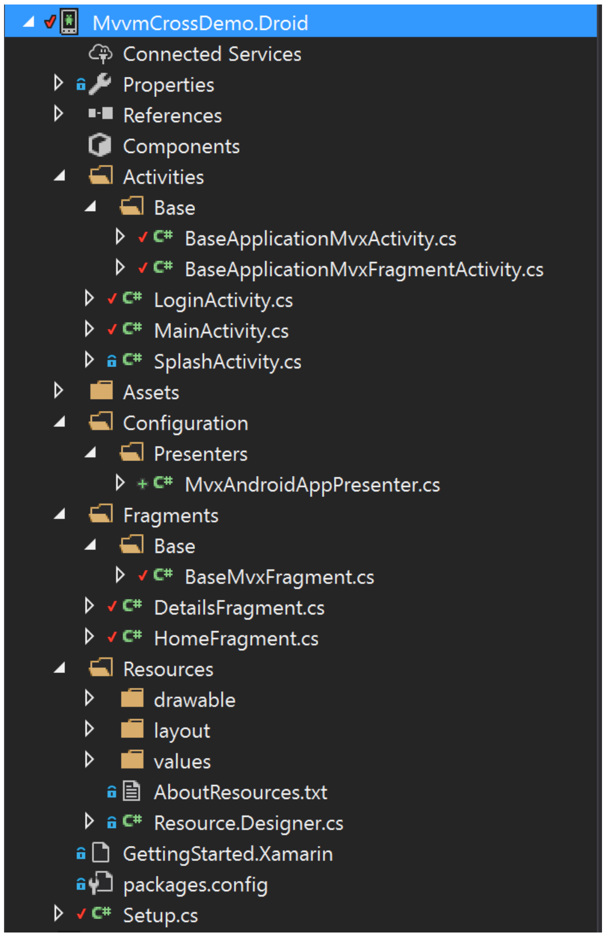
- Configuration
    - Startup – this folder contains class responsible for application startup. In this class you have chance to decide which ViewModel will be displayed first and in which order
- Infrastructure
    - Services – this folder contains all services you use in the application. Of course if you have more complex application you can create your own structure. In our case Services folder contains class called AuthenticationService responsible for handling user authentication
        - Interfaces – this folder contains all services interfaces used in the application so in our case it will be IAuthenticationService
- Model – this folder contains model you will use in the application so in our case this is for instance UserData class. Here you can add many different classes you use to map your objects in the application
- ViewModel – this folder is crucial in the project structure. It contains all ViewModels you use in your application. As you can see there is LoginViewModel responsible for user login operations or HomeViewModel connected with main screen in the application
    - Base – this folder contains base ViewModel for all other ViewModels you will create
    - Interfaces – this folder contains base interfaces which will be implemented by other ViewModels
- App.cs – this class is hear of the application. Inside it you can register dependencies in Mvx IoC container including MvxAppExtendedStart class object

Please open my sample from GitHub and go through the structure and classes. I added comments to explain the code.

3.4. Xamarin.Android project structure

In this section I would like to discuss Xamarin.Android project structure and key elements.

On the figure below you can see how project structure template looks like. It has some crucial elements which we will discuss in this section.



```
MvvmCrossDemo.Droid
    Connected Services
    Properties
    References
    Components
    Activities
        Base
            BaseApplicationMvxActivity.cs
            BaseApplicationMvxFragmentActivity.cs
        LoginActivity.cs
        MainActivity.cs
        SplashActivity.cs
    Assets
    Configuration
        Presenters
            MvxAndroidAppPresenter.cs
    Fragments
        Base
            BaseMvxFragment.cs
        DetailsFragment.cs
        HomeFragment.cs
    Resources
        drawable
        layout
        values
        AboutResources.txt
        Resource.Designer.cs
    GettingStarted.Xamarin
    packages.config
    Setup.cs
```

Figure 36 - Xamarin.Android project structure

Let me walk through all these folders so you can understand what is happening here.

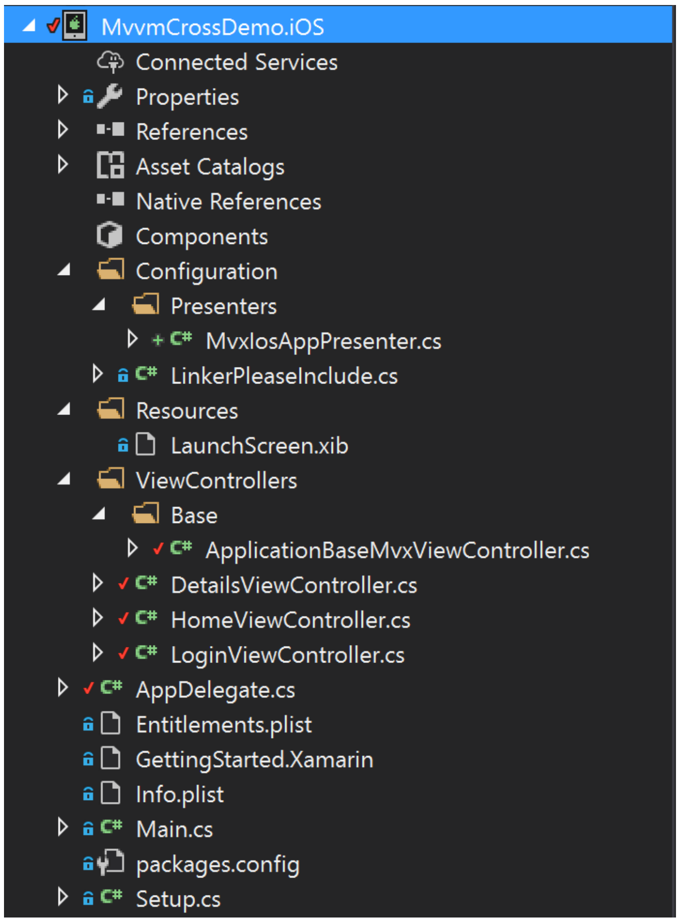
- Activities – this folder contains Activities code-behind classes
    - Base – this folder contains base classes for all activities used in the application
- Assets – default assets folder automatically created
- Configuration
    - Presenters – this folder contains View Presenter for Android platform (discussed earlier in MvxViewPresenter section)
- Fragments – this folder contains Fragments code-behind classes
    - Base – this folder contains base class for each fragment used in the application
- Resources (drawable, layout and values) – standard Android folders
- Setup.cs – Class which initializes View Presenter for Android application and using App.cs class from Core project to register dependencies. Setup class should be always included in the project

Please open my sample from GitHub and go through the structure and classes. I added comments to explain the code.

3.5.Xamarin.iOS project structure

In this section I would like to discuss Xamarin.iOS project structure and key elements.

On the figure below you can see how project structure template looks like. It has some crucial elements which we will discuss in this section.



```
MvvmCrossDemo.iOS
    Connected Services
    Properties
    References
    Asset Catalogs
    Native References
    Components
    Configuration
        Presenters
            MvxIosAppPresenter.cs
        LinkerPleaseInclude.cs
    Resources
        LaunchScreen.xib
    ViewControllers
        Base
            ApplicationBaseMvxViewController.cs
        DetailsViewController.cs
        HomeViewController.cs
        LoginViewController.cs
    AppDelegate.cs
    Entitlements.plist
    GettingStarted.Xamarin
    Info.plist
    Main.cs
    packages.config
    Setup.cs
```

Figure 37- Xamarin.iOS project structure

Let me walk through all these folders so you can understand what is happening here.

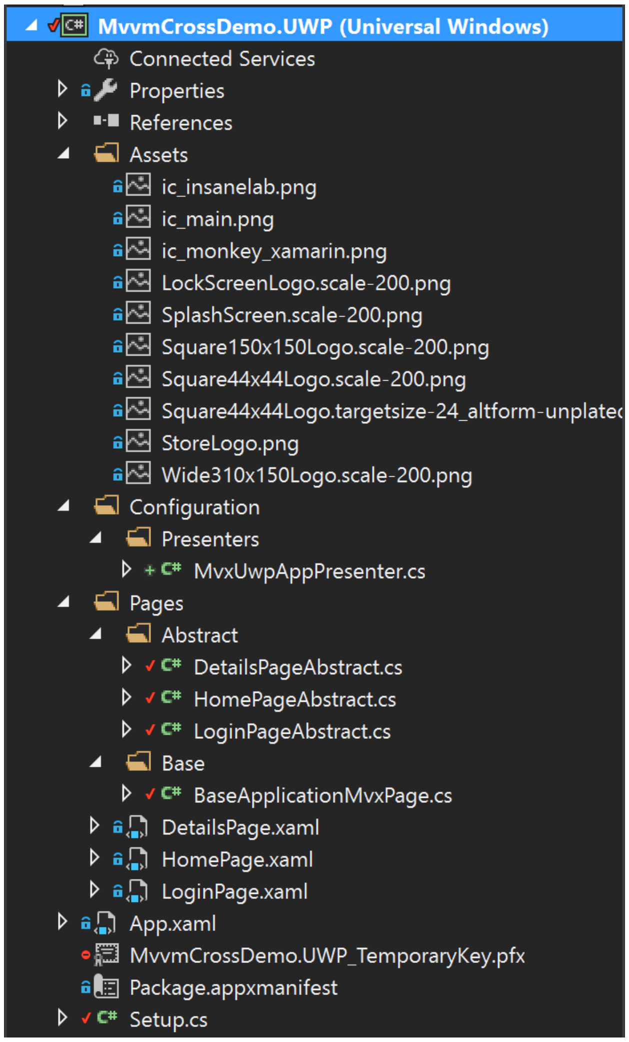
- Configuration
    - Presenters - this folder contains View Presenter for iOS platform (discussed earlier in MvxViewPresenter section)
    - LinkerPleaseInclude – this class is needed to give instructions for linker what should not be linked so application can work correctly
- Resources – standard resources folder created automatically
- ViewControllers – this folder contains all ViewControllers available in the application
    - Base – this folder contains base class for each View Controller used in the application
- AppDelegate.cs – class which is responsible for correct interaction with the system and with other apps
- Setup.cs – Class which initializes View Presenter for iOS application and using App.cs class from Core project to register dependencies. Setup class should be always included in the project

Please open my sample from GitHub and go through the structure and classes. I added comments to explain the code.

3.6. Xamarin.UWP project structure

In this section I would like to discuss Xamarin.UWP project structure and key elements.

On the figure below you can see how project structure template looks like. It has some crucial elements which we will discuss in this section.



```
MvvmCrossDemo.UWP (Universal Windows)
  Connected Services
  Properties
  References
  Assets
    ic_insanelab.png
    ic_main.png
    ic_monkey_xamarin.png
    LockScreenLogo.scale-200.png
    SplashScreen.scale-200.png
    Square150x150Logo.scale-200.png
    Square44x44Logo.scale-200.png
    Square44x44Logo.targetsize-24_altform-unplatec
    StoreLogo.png
    Wide310x150Logo.scale-200.png
  Configuration
    Presenters
      MvxUwpAppPresenter.cs
  Pages
    Abstract
      DetailsPageAbstract.cs
      HomePageAbstract.cs
      LoginPageAbstract.cs
    Base
      BaseApplicationMvxPage.cs
    DetailsPage.xaml
    HomePage.xaml
    LoginPage.xaml
  App.xaml
  MvvmCrossDemo.UWP_TemporaryKey.pfx
  Package.appxmanifest
  Setup.cs
```

Figure 38 - Xamarin.UWP project structure

Let me walk through all these folders so you can understand what is happening here.

- Assets – folder contains graphic assets for the application (like splash screen)
- Configuration
    - Presenters - this folder contains View Presenter for UWP platform (discussed earlier in MvxViewPresenter section)
- Pages – this folder contains all pages implementations (code behind) available in the application
    - Abstract – this folder contains abstractions of each page in the application
    - Base – this folder contains base class for each page used in the application
- Setup.cs – Class which initializes View Presenter for UWP application and using App.cs class from Core project to register dependencies. Setup class should be always included in the project

4. SAMPLE APPLICATION CODE REVIEW

In this section I would like to discuss the most important fragments of the code in the sample application I prepared. I assume that you are somehow familiar with MVVM pattern and have good knowledge of C# language. Whole sample is available on my GitHub. This section contains only selected and the most important fragments of code that should be explained.

4.1. Core project code review

In this section we will discuss code from the core project so you can understand how application works.

4.1.1. MvxAppExtendedStartup class

This class is responsible for app startup. Developer has chance to decide which ViewModel will be displayed once application is launched. In our case we will display either home page or login page depends if user is authenticated or not. MvxAppExtendedStartup class looks like below:

```
// Class has to implement IMvxAppStart interface so we will register it in IoC container:
    public class MvxAppExtendedStart: IMvxAppStart
    {
        private readonly IAuthenticationService _authenticationService;
        private readonly IMvxNavigationService _navigationService;

        // Initialize authentication service and navigation service using constructor
dependency injection:
        public MvxAppExtendedStart(IAuthenticationService authenticationService,
IMvxNavigationService navigationService)
        {
            _authenticationService = authenticationService;
            _navigationService = navigationService;
        }

        public void Start(object hint = null)
        {
            // If user is not authenticated display LoginViewModel:
            if (!_authenticationService.IsLoggedIn())
                _navigationService.Navigate<LoginViewModel>();
            // else display HomeViewModel:
            else
                _navigationService.Navigate<HomeViewModel>();
        }
    }
```

4.1.2. App class

This class is a plaace where dependencies can be registered in Mvx IoC container. There we also should register AppStart object. Please look on the code below:

```
// McxApplication should be extended:
  public class App : MvxApplication
  {

      private void ConfigureIoC()
      {
          Mvx.RegisterType<IAuthenticationService, AuthenticationService>();
      }

      public override void Initialize()
      {
          base.Initialize();

          ConfigureIoC();

          // We can change login state to see control navigation flow:
          Mvx.Resolve<IAuthenticationService>().LoggedIn = false;
          //Register extended app startup in IoC container:
          Mvx.ConstructAndRegisterSingleton<IMvxAppStart, MvxAppExtendedStart>();
          var appStart = Mvx.Resolve<IMvxAppStart>();
          // register the appstart object:
          RegisterAppStart(appStart);
      }
  }
```

4.1.3. BaseApplicationMvxViewModel

This is the base class for each ViewModel you create. It contains some methods written in generic way so you can use them with different types.

BaseApplicationMvxViewModel contains:

- Constructor with NavigationService dependency so we can navigate between ViewModels
- Navigate method which is responsible for navigation to other ViewModel without passing parameter
- Navigate method which enables navigation to other ViewModel with option to pass a parameter
- Close methods – responsible for closing selected ViewModel
- PresentationChanged method – to handle Presentation Hints when navigating to other ViewModel

Code looks like below:

```
//Base class for each ViewModel without initial parameter:
    public abstract class BaseApplicationMvxViewModel : MvxViewModel,
IBaseApplicationMvxViewModel
    {
        protected readonly IMvxNavigationService _navigationService;

        //Dependency injection to get navigation service:
        public BaseApplicationMvxViewModel(IMvxNavigationService navigationService)
        {
            _navigationService = navigationService;
        }

        //Method to navigate to other ViewModel without passing any parameters:
        public async Task Navigate<TViewModel>() where TViewModel : IMvxViewModel
        {
            await _navigationService.Navigate<TViewModel>();
        }


        //Method to navigate to other ViewModel with option to pass a parameter:
        public async Task Navigate<TViewModel, TParameter>(TParameter param)
            where TViewModel : IMvxViewModel<TParameter>
            where TParameter : class
        {
            await _navigationService.Navigate<TViewModel, TParameter>(param);
        }

        // Method responsible for closing ViewModel:
        public async Task Close()
        {
            await _navigationService.Close(this);
        }
        //Method responsible for handling Presentation Hints during navigation:
        public bool PresentationChanged(MvxPresentationHint presentationHint)
        {
            return ChangePresentation(presentationHint);
        }
    }
```

```
   //Base class for each ViewModel which requires initial parameter (it extends 
BaseApplicationMvxViewModel class written above):
    public abstract class BaseApplicationMvxViewModel<TInitParams> : 
BaseApplicationMvxViewModel, IMvxViewModel<TInitParams> where TInitParams : class
    {
        public BaseApplicationMvxViewModel(IMvxNavigationService navigationService) : 
base(navigationService)
        {
        }

        // Access passed parameter:
        public virtual Task Initialize(TInitParams parameter)
        {
            return Task.FromResult(true);
        }
    }
```

4.2. Xamarin.Android project code review

In this section we will discuss code from the Xamarin.Android project so you can understand how application works.

4.2.1. Setup class

This class is crucial and should be always added to the project. Class which initializes View Presenter and using App.cs class from Core project to register dependencies.

It looks like below:

```
 // MvxAndroidSetup should be extended:
    public class Setup: MvxAndroidSetup
    {
        public Setup(Context applicationContext) : base(applicationContext)
        {
        }

        // Create new App instance and register dependencies:
        protected override IMvxApplication CreateApp()
        {
            return new App();
        }

        // Create View Presenter for Android application:
        protected override IMvxAndroidViewPresenter CreateViewPresenter()
        {
            //Create Android View Presenter object:
            var mvxFragmentsPresenter = new
MvxAndroidAppPresenter(AndroidViewAssemblies);
            Mvx.RegisterSingleton<IMvxAndroidViewPresenter>(mvxFragmentsPresenter);
            return mvxFragmentsPresenter;
        }

        // This code is required to be able to use some bindings with AppCompat:
        protected override void FillTargetFactories(IMvxTargetBindingFactoryRegistry
registry)
        {
            MvxAppCompatSetupHelper.FillTargetFactories(registry);
            base.FillTargetFactories(registry);
        }
    }
```

4.2.2. MvxAndroidAppPresenter

As mentioned before View Presenters are a key object in the MVVM Cross architecture. You can imagine that View Presenter is like glue between ViewModel and View. Each platform has its own View Presenter. If you check in the code class MvxAndroidAppPresenter extends MvxFragmentsPresenter and this class derives from MvxAndroidViewPresenter class.

You can either use default Android View Presenter or create new class as it is in my sample to provide custom implementation. I use fragments in sample application that is why I extended MvxFragmentPresenter class. It looks like below:

```
public class MvxAndroidAppPresenter : MvxFragmentsPresenter
    {
        //Extend default constructor which register all Android views available in the
application:
        public MvxAndroidAppPresenter(IEnumerable<Assembly> androidViewAssemblies) :
base(androidViewAssemblies) { }

        // This method helps configure some system preferences like animaton when
navigating from splash screen to login screen:
        protected override Intent CreateIntentForRequest(MvxViewModelRequest request)
        {
            var intent = base.CreateIntentForRequest(request);

            if (request.ViewModelType == typeof(LoginViewModel) && Activity.GetType() ==
typeof(SplashActivity))
                intent.AddFlags(ActivityFlags.NoAnimation);

            if (request.ViewModelType == typeof(MainViewModel) || request.ViewModelType
== typeof(LoginViewModel))
                intent.AddFlags(ActivityFlags.ClearTask);

            return intent;
        }
    }
```

4.3. Xamarin.iOS project code review

In this section we will discuss code from the Xamarin.iOS project so you can understand how application works.

4.3.1. Setup class

This class is crucial and should be always added to the project. Class which initializes View Presenter and using App.cs class from Core project to register dependencies.

It looks like below:

```
// MvxIosSetup should be extended:
    public class Setup : MvxIosSetup
    {
        public Setup(MvxApplicationDelegate appDelegate, IMvxIosViewPresenter presenter)
        : base(appDelegate, presenter)
        {

        }

        protected override IMvxApplication CreateApp()
        {
            return new App();
        }
    }
```

For iOS application there is different way for setup initialization. Please open AppDelegate class and go to FinishedLaunching method:

```
public override bool FinishedLaunching(UIApplication application, NSDictionary
launchOptions)
        {
            // create a new window instance based on the screen size
            Window = new UIWindow(UIScreen.MainScreen.Bounds);

            // Create iOS View Presenter object:
            var presenter = new MvxIosAppPresenter(this, Window);
            var setup = new Setup(this, presenter);
            // Initialize app start with view presenter and dependencies:
            setup.Initialize();

            var startup = Mvx.Resolve<IMvxAppStart>();
            startup.Start();

            Window.MakeKeyAndVisible();

            return true;
        }
```

4.3.2. MvxIosAppPresenter

As mentioned before View Presenters are a key object in the MVVM Cross architecture. You can imagine that View Presenter is like glue between ViewModel and View. Each platform has its own View Presenter. If you check in the code class MvxIosAppPresenter extends MvxIosViewPresenter.

You can either use default iOS View Presenter or create new class as it is in my sample to provide custom implementation. It looks like below:

```
// Extend MvxIosViewPresenter so you can so custom implementation:
    public class MvxIosAppPresenter : MvxIosViewPresenter
    {
        public MvxIosAppPresenter(IUIApplicationDelegate applicationDelegate, UIWindow
window) : base(applicationDelegate, window)
        {
        }
    }
```

4.4. Xamarin.UWP project code review

In this section we will discuss code from the Xamarin.iOS project so you can understand how application works.

4.4.1. Setup class

This class is crucial and should be always added to the project. Class which initializes View Presenter and using App.cs class from Core project to register dependencies.

It looks like below:

```
    // MvxWindowsSetup should be extended:
    public class Setup : MvxWindowsSetup
    {
        public Setup(Frame rootFrame) : base(rootFrame)
        {
        }

        protected override IMvxApplication CreateApp()
        {
            return new MvvmCrossDemo.Core.App();
        }

        // Create View Presenter for UWP application:
        protected override IMvxWindowsViewPresenter CreateViewPresenter(IMvxWindowsFrame
rootFrame)
        {
            //Create UWP View Presenter object:
            var mvxUwpPresenter = new MvxUwpAppPresenter(rootFrame);
            return mvxUwpPresenter;
        }
    }
```

For UWP application there is different way for setup initialization. Please open App.xaml.cs class and go to OnLaunched method and see below fragment:

```
if (e.PrelaunchActivated == false)
            {
                if (rootFrame.Content == null)
                {
                    SystemNavigationManager.GetForCurrentView().BackRequested += 
OnBackRequested;

                    var setup = new Setup(rootFrame);
                    // Initialize app start with view presenter and dependencies:
                    setup.Initialize();

                    var start = Mvx.Resolve<IMvxAppStart>();
                    start.Start();
                }
                // Ensure the current window is active
                Window.Current.Activate();
            }
```

4.4.2. MvxUwpAppViewPresenter

As mentioned before View Presenters are a key object in the MVVM Cross architecture. You can imagine that View Presenter is like glue between ViewModel and View. Each platform has its own View Presenter. If you check in the code class MvxUWPAppPresenter extends MvxWindowsViewPresenter.

You can either use default UWP View Presenter or create new class as it is in my sample to provide custom implementation. It looks like below:

```
// Extend MvxWindowsViewPresenter so you can so custom implementation:
  public class MvxUwpAppPresenter : MvxWindowsViewPresenter
  {
      public MvxUwpAppPresenter(IMvxWindowsFrame rootFrame) : base(rootFrame)
      {
      }
  }
```

4.5. Binding

Binding is mechanism which enables communication between ViewModel and View. Implementation looks different on each platform and that is why I decided to include this section.

4.5.1. Binding in Xamarin.Android application

Binding in Android application is quite straightforward. Let me explain it using Login Activity.

Open LoginActivity.axml file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:local="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/LayoutRoot"
    android:background="@color/activityBackgroundColor">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:paddingTop="60dp"
        android:paddingLeft="40dp"
        android:paddingRight="40dp">
        <ImageView
            android:src="@drawable/ic_monkey_xamarin"
            android:layout_width="90dp"
            android:layout_height="90dp"
            android:id="@+id/imageView1"
            android:scaleType="fitXY"
            android:fitsSystemWindows="true"
            android:layout_marginBottom="20dp"
            android:layout_gravity="center_horizontal" />
        <EditText
            style="@style/MaterialEditText"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/editText1"
            android:lineSpacingExtra="8dp"
            android:hint="Username..."
            local:MvxBind="Text Username"
            android:inputType="textEmailAddress" />
        <EditText
            style="@style/MaterialEditText"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/editText2"
            android:hint="Password..."
            android:inputType="textPassword"
            local:MvxBind="Text Password" />
        <Button
            style="@style/Widget.AppCompat.Button.Colored"
            android:text="Login"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/button1"
            android:layout_marginTop="20dp"
            local:MvxBind="Click Login; Enabled CanSignIn" />
    </LinearLayout>
</android.support.design.widget.CoordinatorLayout>
```

As you can see there is local namespace added:

```
xmlns:local="http://schemas.android.com/apk/res-auto"
```

This enables using MvxBind in the layout code so we can setup bindings.

Now let's look on EditText for username. We want to pass value provided by user to LoginViewModel. To do it we have to provide information which property of the control will be bound to ViewModel property:

```
local:MvxBind="Text Username"
```

```
<EditText
            style="@style/MaterialEditText"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:id="@+id/editText1"
            android:lineSpacingExtra="8dp"
            android:hint="Username..."
            local:MvxBind="Text Username"
            android:inputType="textEmailAddress" />
```

In this case we want to create binding between Text property of EditText control and Username property declared in LoginViewModel.

What about binding context? It is provided in code behind for this Activity:

```
public class LoginActivity: BaseApplicationMvxActivity<LoginViewModel>
```

We can of course use binding mechanism with Button to control when user clicks it:

```
local:MvxBind="Click Login; Enabled CanSignIn" />
```

As you can see we can in this case we want to create binding between Click property of Button control and Login command declared in LoginViewModel. What is more we can decide when button should be enabled – that is why Enabled property is bound to CanSignIn property from LoginViewModel.

4.5.2. Binding in Xamarin.iOS application

Binding in iOS application looks different than for Android app. In this case we provide whole binding information in the C# code. As you know editing Storyboards code is not convenient so it is much easier to implement binding in the code.

Let me explain it using Login ViewController.

Open LoginViewController.cs file and go to SetupBindings method:

```
private void SetupBindings()
        {
            var bindingSet = this.CreateBindingSet<LoginViewController,
LoginViewModel>();

            bindingSet.Bind(_loginTextField)
                .To(x => x.Username);

            bindingSet.Bind(_passwordTextField)
                .To(x => x.Password);

            bindingSet.Bind(_loginButton)
                .To(x => x.Login);

            bindingSet.Bind(_loginButton)
                .For(x => x.Enabled)
                .To(x => x.CanSignIn);

            bindingSet.Apply();
        }
```

As you can see we have to create BindingSet which contains information about ViewController and assigned ViewModel. Once BindingSet is created we can configure bindings. We can use example with UITextField. We want to pass value provided by user to LoginViewModel. To do it we have to provide information which property of the control will be bound to ViewModel property:

```
bindingSet.Bind(_loginTextField).To(x => x.Username);
```

In this case we want to create binding between Text property of UITextField control and Username property declared in LoginViewModel. By default UITextField.Text property is used. If you want to create binding for different property you should use For method:

```
bindingSet.Bind(_loginTextField).For(x=> x.Text).To(x => x.Username);
```

We can of course use binding mechanism with UIButton to control when user clicks it:

```
bindingSet.Bind(_loginButton).To(x => x.Login);

bindingSet.Bind(_loginButton).For(x => x.Enabled).To(x => x.CanSignIn);
```

As you can see we can in this case we want to create binding between Click property of Button control and Login command declared in LoginViewModel. What is more we can decide when button should be enabled – that is why Enabled property is bound to CanSignIn property from LoginViewModel.

4.5.3. Binding in Xamarin.UWP application

Binding in UWP application is not complicated and has its default format.

Let me explain it using LoginPage.xaml file:

```
<abstract:LoginPageAbstract
    x:Class="MvvmCrossDemo.UWP.Pages.LoginPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:abstract="using:MvvmCrossDemo.UWP.Pages.Abstract"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="White">
        <Grid.RowDefinitions>
            <RowDefinition Height="97*"/>
            <RowDefinition Height="223*"/>
        </Grid.RowDefinitions>

        <Image HorizontalAlignment="Center" VerticalAlignment="Top" Margin="0,50,0,0"
Source="ms-appx:///Assets/ic_monkey_xamarin.png" Width="100" Height="100"/>

        <StackPanel HorizontalAlignment="Stretch" Grid.Row="2" Margin="40,0,40,0">
            <TextBox Foreground="#FF24B780" Background="{x:Null}" BorderBrush="#FF24B780"
Height="36" Padding="10,4,0,0" PlaceholderText="Username..." Text="{Binding Username,
Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" SelectionHighlightColor="Black"
BorderThickness="1" Style="{StaticResource TextBoxCustomStyle}"/>
            <PasswordBox Margin="0,10,0,0" Background="{x:Null}" BorderBrush="#FF24B780"
Height="36" Foreground="#FF24B780" Padding="10,4,0,0" PlaceholderText="Password..."
Password="{Binding Password, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}"
SelectionHighlightColor="Black" BorderThickness="1" Style="{StaticResource
PasswordBoxCustomStyle}" />
            <Button Margin="0,20,0,0" HorizontalAlignment="Stretch" Height="40"
Content="Login" Background="#FF24B780" BorderBrush="Black" BorderThickness="1"
Foreground="White" FocusVisualPrimaryBrush="#FFF7F7F7" Command="{Binding Login}"
IsEnabled="{Binding CanSignIn, Mode=TwoWay}" />
        </StackPanel>
    </Grid>
</abstract:LoginPageAbstract>
```

First of all as you can see there is abstraction used for each page in the project. It is because we would like to provide generic way for binding between ViewModel and Page. That is why we have to create abstract classes for each page like below:

```
public abstract class LoginPageAbstract : BaseApplicationMvxPage<LoginViewModel>
```

Each abstract page class derives from BaseApplicationMvxPage with assigned ViewModel.

For LoginPage code behind looks like below:

```
public sealed partial class LoginPage : LoginPageAbstract
    {
        public LoginPage()
        {
            this.InitializeComponent();
        }
    }
```

Now let’s look on TextBox for username. We want to pass value provided by user to LoginViewModel. To do it we have to provide information which property of the control will be bound to ViewModel property:

```
Text="{Binding Username, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}"
```

In this case we want to create binding between Text property of TextBox control and Username property declared in LoginViewModel. There is also UpdateSourceTrigger property to update Username property each time user write anything in the TextBox control.

4.6. Navigation and presentation

MVVM Cross provides specific way to navigate and present views in the application. As mention in MvxViewPresenter section, each platform (Android, iOS and Windows) has its own View Presenter to display views.

4.6.1. Presentation and navigation in Xamarin.Android

MvxAndroidViewPresenter is object that controls presentation layer when navigating between ViewModels. There are two important Android Presenter types provided by MVVM Cross:

- MvxAndroidViewPresenter – enables navigation with Activities
- MvxFragmentsPresenter – enables navigation with Activities and Fragments

In my sample I am using MvxFragmentsPresenter to provide proper navigation:

```
public class MvxAndroidAppPresenter : MvxFragmentsPresenter
```

Now if we invoke Navigate method from ViewModel, Android Presenter is informed and knows how to display view for selected ViewModel:

```
await Navigate<HomeViewModel, UserData>(userData);
```

You can see how exactly MvxFragmentsPresenter class is implemented on GitHub: MvxFragmentsPresenter.cs.

4.6.2. Presentation and navigation in Xamarin.iOS

MvxIosViewPresenter is object that controls presentation layer when navigating between ViewModels. iOS Presenter is different than Android one and supports for the following navigation patterns:

- Tabs
- SplitView
- Modal
- Stack

Key functionality here is set of attributes (Presentation Attributes) which we can use to define how selected view will be displayed:

- MvxTabPresentationAttribute – for tabs
- MvxRootPresentationAttribute – to set ViewController as root
- MvxModalPresentationAttribute – to display ViewController modally
- MvxMasterSplitViewPresentationAttribute – for SplitView master controller
- MvxDetailSplitViewPresentationAttribute – for SplitView details controller

As you can see there are many attributes. Let's see how it looks in the code.

In my sample I am using MvxIosViewPresenter to provide proper navigation:

```
public class MvxIosAppPresenter : MvxIosViewPresenter
```

If I want to set root ViewController wrapped in NavigationController I can use Presenter attributes:

```
[MvxRootPresentation(WrapInNavigationController = true)]
    public class LoginViewController : ApplicationBaseMvxViewController<LoginViewModel>
```

If I want to display ViewController modally I can do it like below using MvxModalPresentation attribute:

```
[MvxModalPresentation]
    public class DetailsViewController :
ApplicationBaseMvxViewController<DetailsViewModel>
```

You can see how exactly MvxiOSViewPresenter class and attributes are implemented on GitHub: iOS Presenter.

4.6.3. Presentation and navigation in Xamarin.UWP

MvxWindowsViewPresenter is object that controls presentation layer when navigating between ViewModels. It enables switching between pages in UWP application within Frame.

In my sample I am using MvxWindowsViewPresenter to provide proper navigation:

```
public class MvxUwpAppPresenter : MvxWindowsViewPresenter
```

Now when navigating between ViewModels, Windows View Presenter controls Frame stack and displays specific pages.

You can see how exactly MvxWindowsViewPresenter class and attributes are implemented on GitHub: MvxWindowsViewPresenter.cs.

5. SUMMARY

I hope that I somehow helped you to understand basics of MVVM Cross framework and now you will be able to start creating cross-platform mobile applications with Xamarin. In this document I concentrated on MVVM architecture pattern, fundamentals of MVVM Cross like IoC container, base ViewModel or NavigationService. I also tried to show you how presentation layer is controlled with View Presenters.

This is only beginning. There are many more features and functionalities so I encourage you to visit official MVVM Cross documentation website.

Remember to review sample connected with this document on my GitHub.

Thank you!

6. About the author

Daniel Krzyczkowski

- Passionate about mobile technologies
- Blogger – mobileprogrammer.pl
- Microsoft & Xamarin Most Valuable Professional
- Find me on Twitter: @DKrzyczkowski