Xamarin BDD MVVM App Developer's Guide

Project Credentials

contact point: Reza Alemy (author), reza@alemy.net

User Story:

As a Mobile App developer
I want to have a boilerplate example of using BDD and MVVM-Light in Xamarin framework
So that I can develop multiplatform apps that use the BDD technique for testing and MVVM-light for component management.

Restrictions and Requirements (MoSCoW method)

- The boilerplate project MUST show how to use Visual Studio to set up the app codebase
- The boilerplate project MUST show how to add Specflow for BDD support to the app codebase
- The boilerplate project MUST show how to write Specflow steps in Xamarin.UITest framework
- The boilerplate project MUST show how to add pages and components using MVVM-Light framework

Acceptance Tests

Given: The repo is cloned

When: I read the documentation included

Then: I have a step by step explanation on how to setup and develop Xamarin apps using Specflow and MVVM.

Reference and Credits:

Rob Gibbens article for Specflow:
http://arteksoftware.com/bdd-tests-with-xamarin-uitest-and-specflow/

Install Development IDE

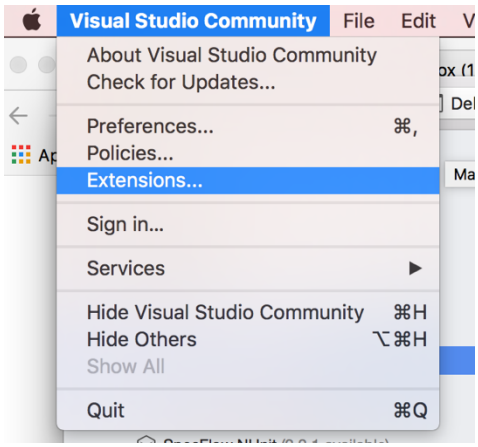
Selected Technology: Visual Studio, C# language on Xamarin framework

Presented Environment: Visual Studio for Mac
Downloaded from: https://www.xamarin.com/download
Version: 7.2.2 (build 11)

Install BDD Framework

Selected Technology: Gherkin BDD language
Presented as Visual Studio for Mac Extension: Specflow for mac
Downloaded from: https://github.com/straighteight/SpecFlow-VS-Mac-Integration/releases/tag/1.11.0.0

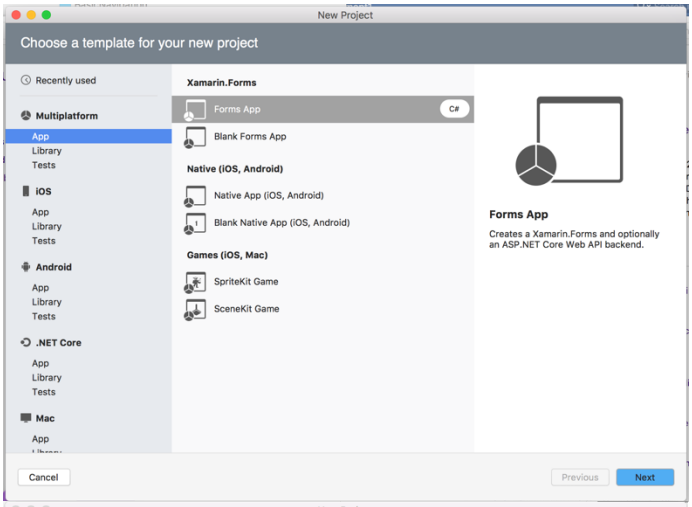
To install, go to 'Visual Studio.../Extensions...' click the button on the far left saying "Install from file..." and select the mpack file from your hard drive. The addin will install and you can now run and creat specflow tests

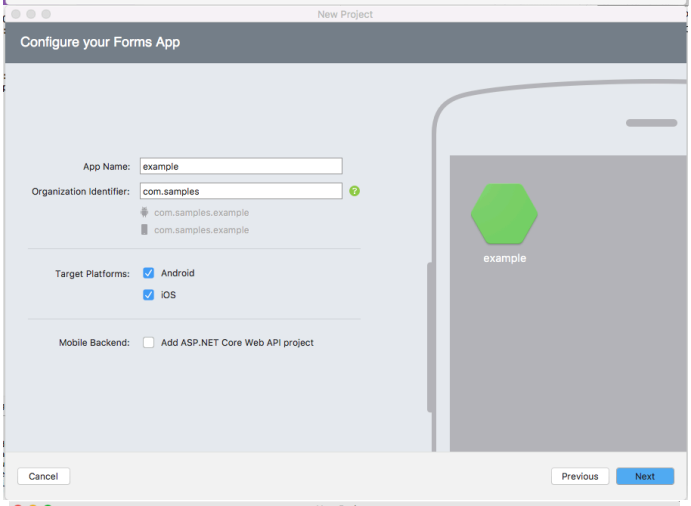


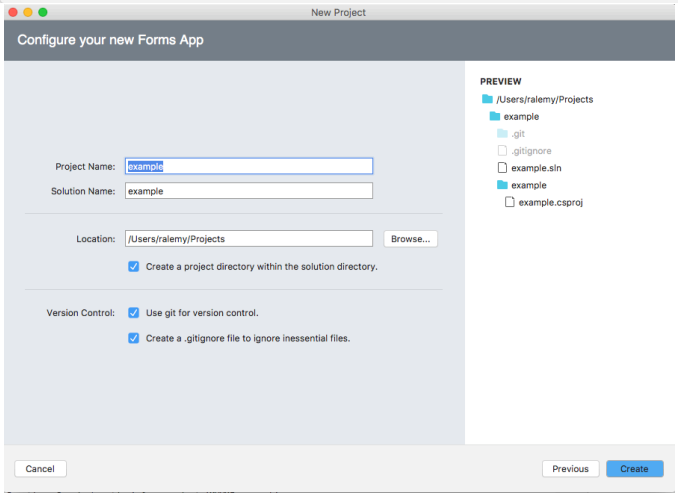
Restart Visual Studio for Mac

Create Main Solution:

Create a solution with Android and iOS.

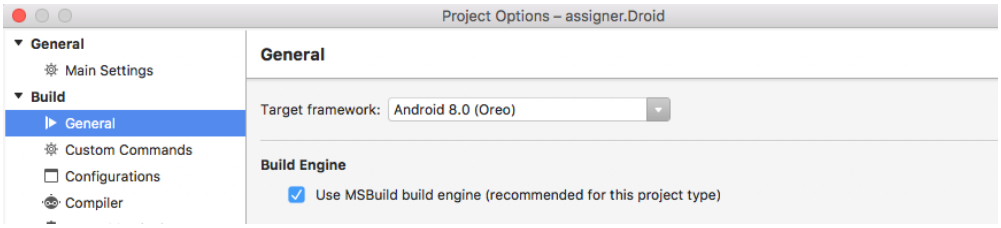




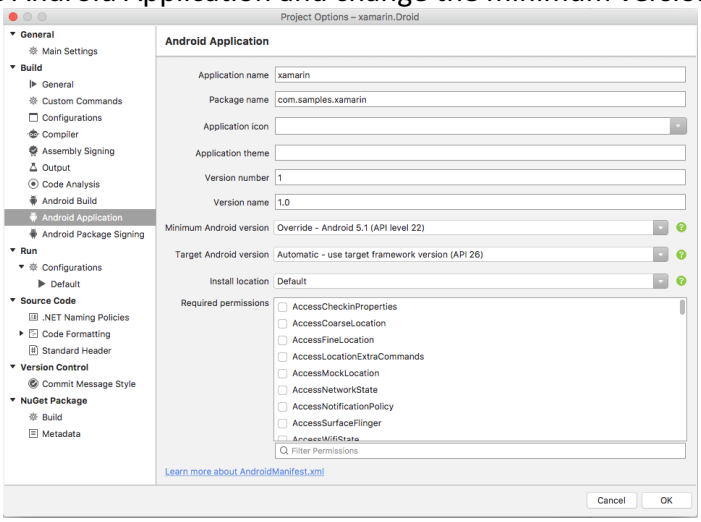


Target the latest Android framework for the Droid project (8.0 for this writing)

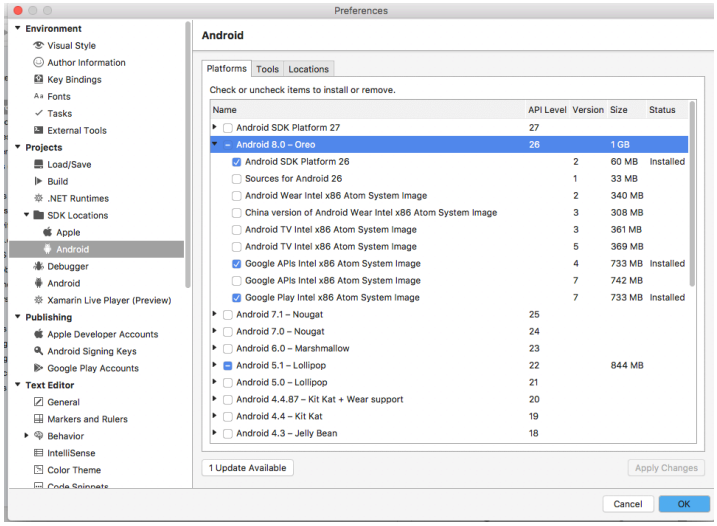
- Right click on Droid project and select options
- Go to Build->General->target framework



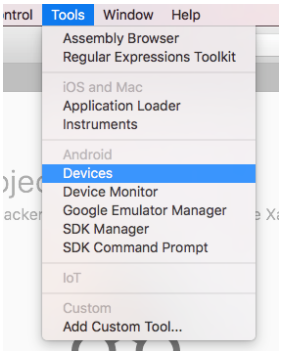
If you need, go to the Android Application and change the minimum version as well:



Install the latest Android SDK for emulator and solution in Tools->SDK Manager



While here, install any other emulator you need (e.g. lollipop). It is also a good time to define simulated devices using the Tools->Devices.



Update Packages for both Droid and iOS.

Add package MVVMLightLibs to both droid and ios frameworks.

Add package Xamarin.TestCloud.Agent to iOS project.
Note: at the time of this writing, there is a bug in TestCloud that crashes the Repl(). Because of this bug, the Xamarin.TestCloud.Agent should be downgraded to 0.21.3. hopefully the next versions would fix this bug.

In the iOSProject, find the AppDelegate.cs, and start Calabash:

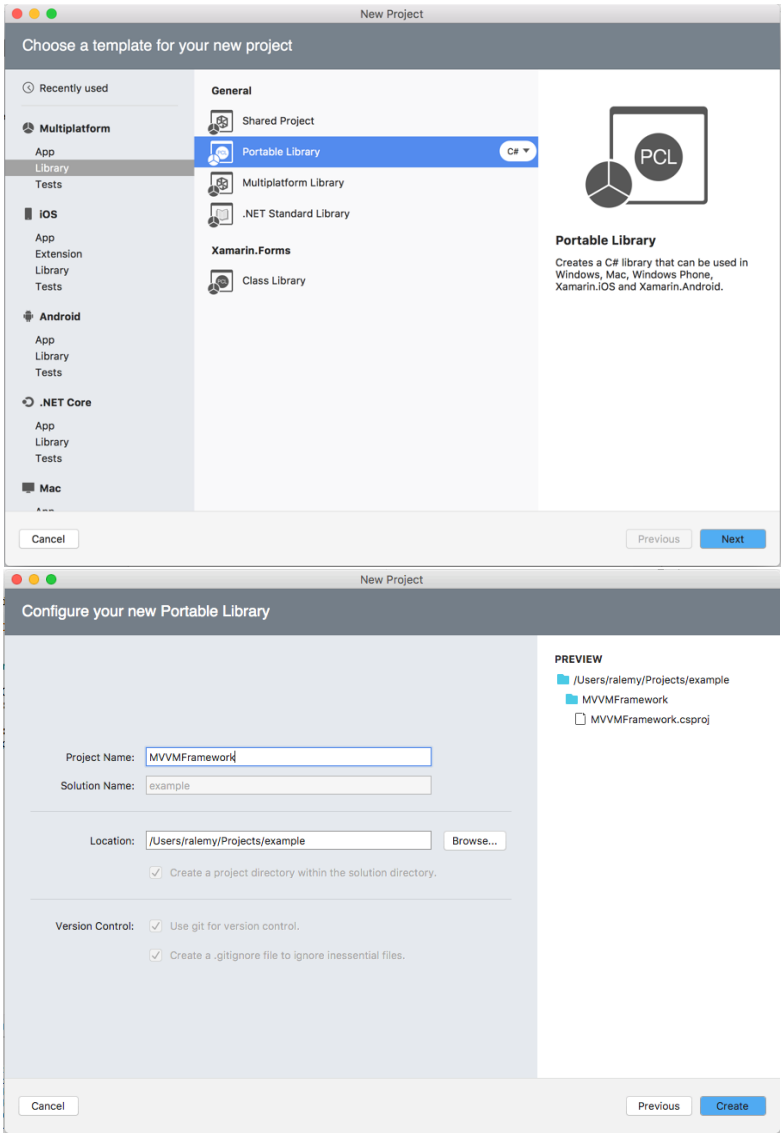
```
    public partial class AppDelegate : global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegat
e
    {
        public override bool FinishedLaunching(UIApplication app, NSDictionary options)
        {

#if ENABLE_TEST_CLOUD
            global::Xamarin.Calabash.Start();
#endif

            global::Xamarin.Forms.Forms.Init();

            LoadApplication(new App());

            return base.FinishedLaunching(app, options);
        }
    }
```

Since the ENABLE_TEST_CLOUD compiler directive is only defined in Debug mode the Calabash will not be present in the Release version, which would have resulted in rejection from Apple Store.

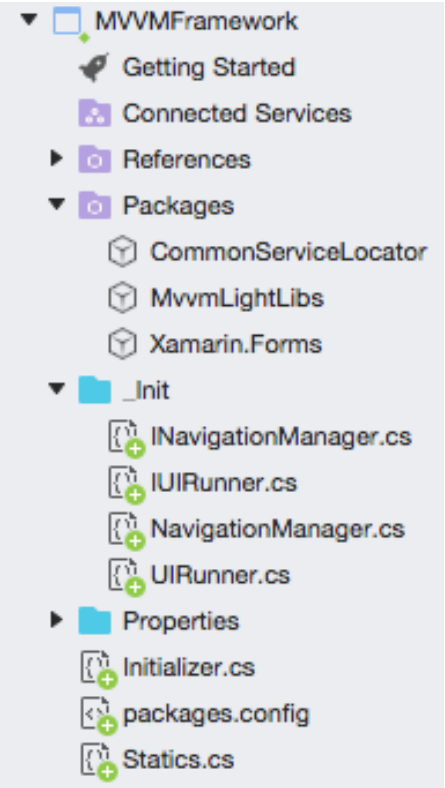
Right click on References directory in the Droid project and select Edit Preferences… Enable Mono.Android.Export so that backdoors would become available to the android project.

Setup the MVVMLight framework

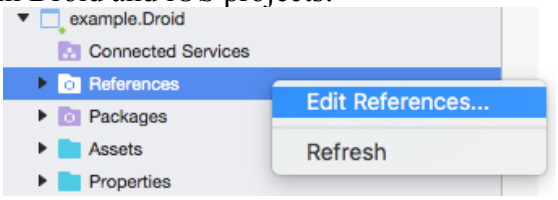
Add a new project for MVVM framework as a portable library

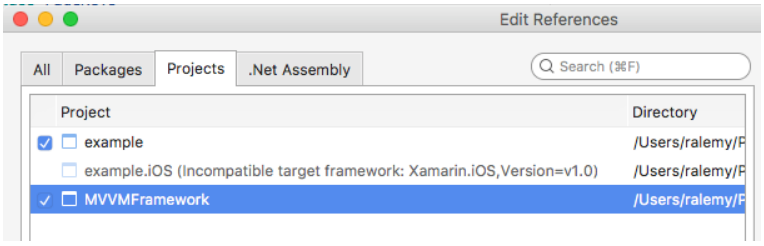


Add MVVMLightLibs and Xamarin.Forms package to this framework
Copy the MVVMFramework/_init directory from reference repo to this framework
Copy the MVVMFramework/Initializer.cs from reference repo to this framework
Copy the MVVMFramework/Statics.cs from reference repo to this framework
Delete MyClass.cs file from the project:



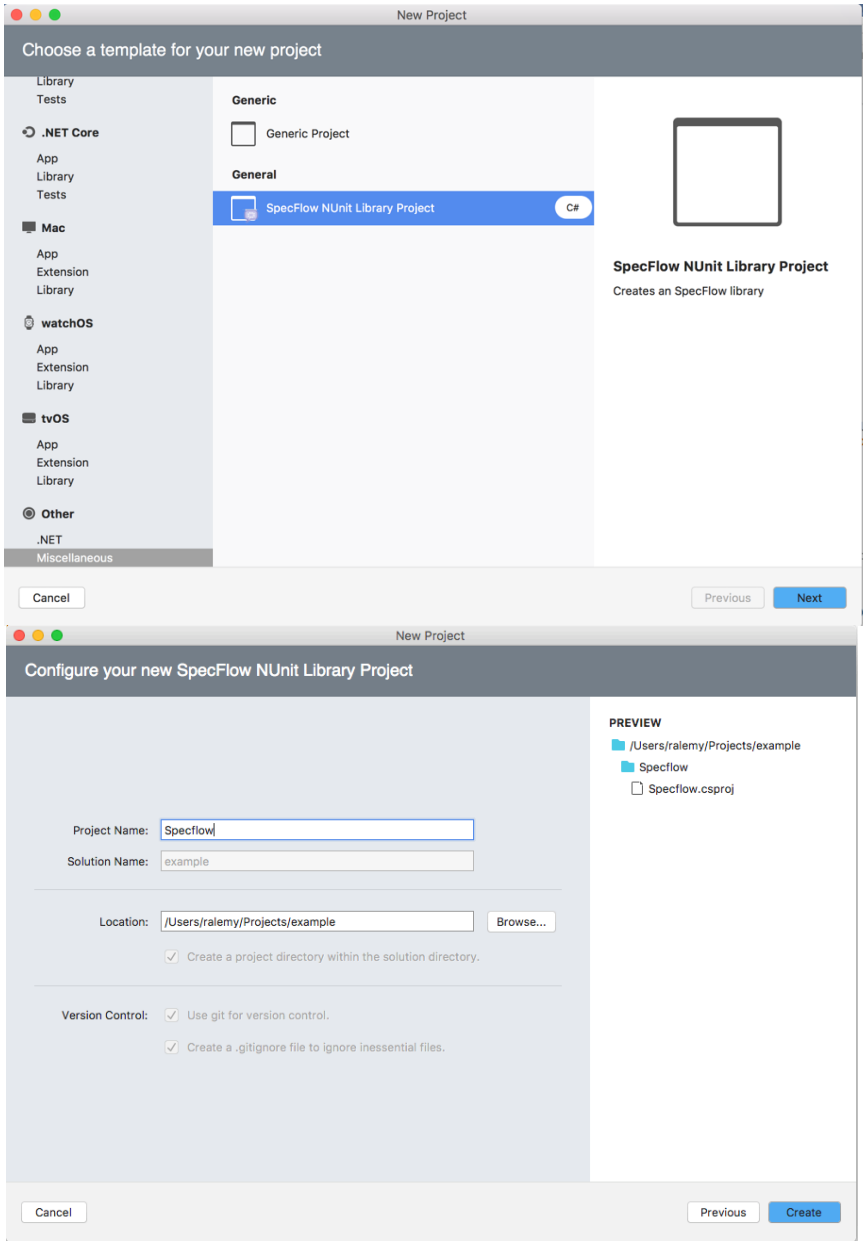
Reference this framework in Droid and iOS projects:





Add BDD Project

Add a new Specflow project from miscellaneous section



Update packages on Specflow project.
Add reference to MVVMFramework project.
Remove Specflow.NUnit and NUnit packages, update Specflow package
Add Xamarin.UiTest, NUnit (2.6.4), Docopt.Net, Grapevine, Restsharp, and Should packages to Specflow project.

- Please note: **based on Xamarin.UITest documentation at the time of this writing, NUnit 2.6.x is the only supported version. Do not install higher versions**.
- Also Note: at the time of this writing there is a bug that causes Repl() to crash. Until it is fixed, an older version of XamarinUITest (2.2.1) should be used.
- Adding Docopt.net will add a few files to the project which can be removed safely.

- The Test.feature and TestSteps.cs should both be deleted.

Requirements

Xamarin Test Cloud and Xamarin.UITest only support Android and iOS applications. The Windows Phone and Windows versions of Xamarin.Forms apps cannot be tested in Test Cloud.

It is important to understand the concepts described in the Introduction to Xamarin Test Cloud guide.

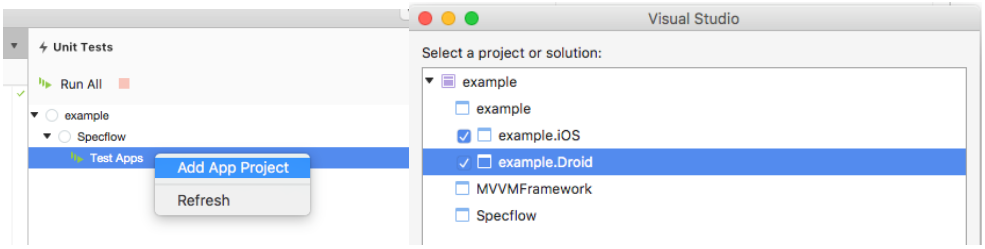
It is assumed that the most recent stable version of the Xamarin.UITest NuGet Package is installed in the UITest project. It is assume that iOS projects have the most recent version of the Xamarin Test Cloud Agent installed.

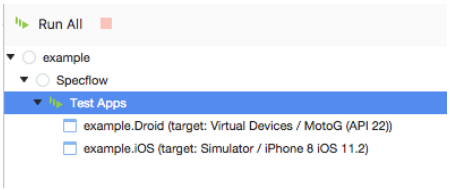
iOS devices must be configured with a valid development provisioning profile.

In order to run Xamarin.UITests with Visual Studio for Mac, the following dependencies must be met:

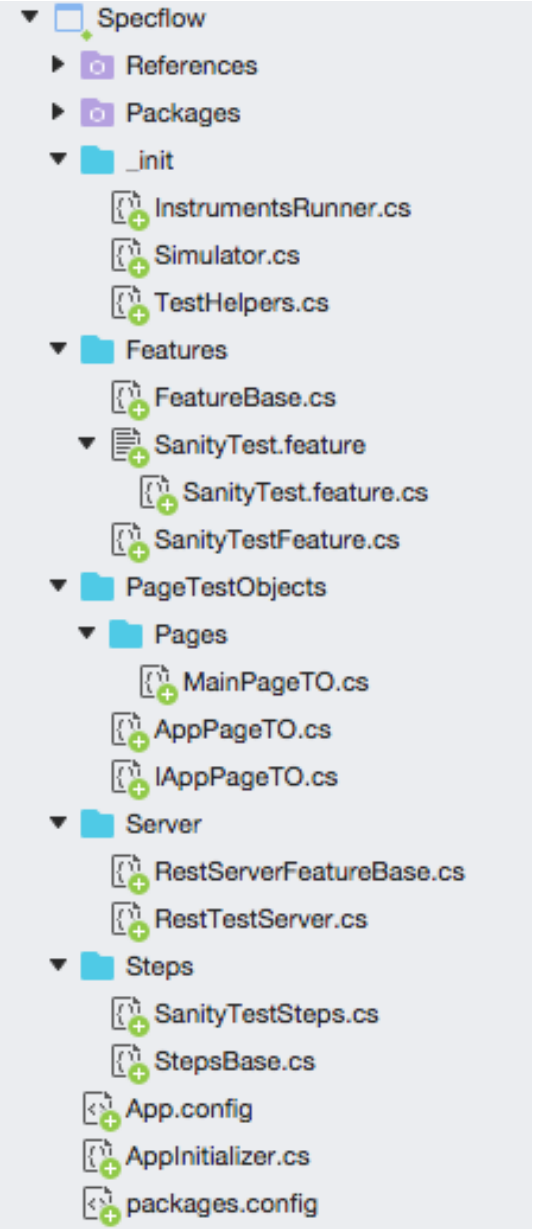
- **NUnit 2.6.x** – Xamarin.UITest is not compatible with NUnit 3.x.
- **Android SDK** – Only if testing Android apps.
- **Java Developers Kit** – Only if testing Android apps.
- **Xcode Command Line Tools** – Only for testing iOS apps.

Open the UnitTest pad from the View menu and add the droid and iOS projects as Test Apps





put the AppInitializer.cs in the Specflow project.
Copy _init directory in the Specflow project.
Copy Features directory from reference repo to Specflow.
Copy PageTestObjects directory from reference repo to Specflow.
Copy Server directory from reference repo to Specflow
Copy Steps directory from reference repo to Specflow.



Refactoring the Shared Project to MVVM model

From the shared project, remove all folders (Model, View, ViewModel) and files and just keep the App.xaml and App.xaml.cs. Add two new directories, Pages and ViewModels

Structure of an MVVM page

Each MVVM page consists of three components, a xaml file, a xaml.cs file and a ViewModel file.

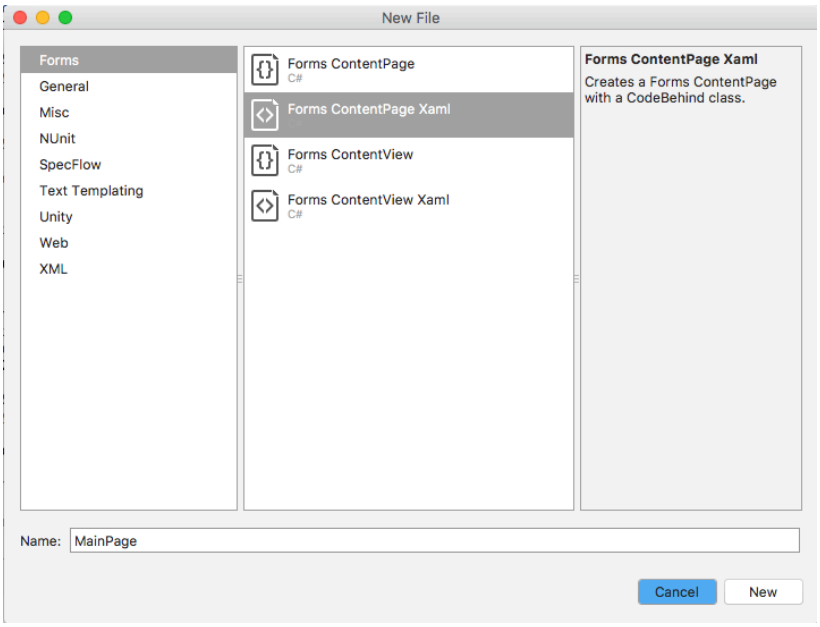
- The xaml file contains the markup for the page and should include an AutomationId for each element that is subject of UITests (more on this later)

- The xaml.cs file contains the code-behind, but in MVVM model it mainly serves two purposes
    - It contains a static PageKey attribute used for navigation
    - It assigns the BindingContext to the ViewModel file
- The ViewModel file contains all logic for the page, but is independent of UI and can be tested with Unit tests.

Adding the Main Page

Make sure that both the Droid and the iOS project reference the MVVMFramework directory.

Add a Content Page Xaml to Pages directory and call it MainPage



Add a Class to ViewModels directory and call it MainPageVM (it has to be a subclass of ViewModelBase)

```
public class MainPageVM :ViewModelBase
 {
     public MainPageVM()
     {
     }
 }
```

Replace the contents of the App.xaml.cs to the following:

```
public partial class App : Application
{
    public App()
    {
        InitializeComponent();
        RegisterDependencies();
        MainPage = RegisterPages(new NavigationPage(new MainPage()));
    }

    private Page RegisterPages(NavigationPage page)
    {
        var nav = Initializer.GetDependency<INavigationManager>();
        nav.SetMain(page);
        return page;
    }

    private void RegisterDependencies()
    {
        Initializer.SetupDI();
        Initializer.Register<MainPage>();
    }

}
```

In the MainPage.xaml.cs, set the binding context to the View model object, and the AutomationId to the PageKey, which comes from MVVMFramework Statics:

```
public partial class MainPage : ContentPage
{
    public const string PageKey = PageKeys.MainPage;
    public MainPage()
    {
        InitializeComponent();
        AutomationId = PageKey;
        BindingContext = Initializer.GetDependency<MainPageVM>();
    }
}
```

Put a simple component in MainPage.xaml.

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.microsoft.com/
winfx/2009/xaml" x:Class="associator.Pages.MainPage">
    <ContentPage.Content>
        <StackLayout Padding = "30">
                    <Label Text= "Boilerplate app"/>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Building the solution

Go to the directory of your project, remove the bin and obj directory under Droid and iOS project, and then select the build->Rebuild All option from the menu. Few warnings may happen that could be addressed.

For example, on a Mac there may be a warning for AutoGenerateBindingRedirects key to be added to the iOS project, or duplicate assembly warnings that would be resolved by removing reference to Microsoft.Csharp package.

Running The Sanity Tests

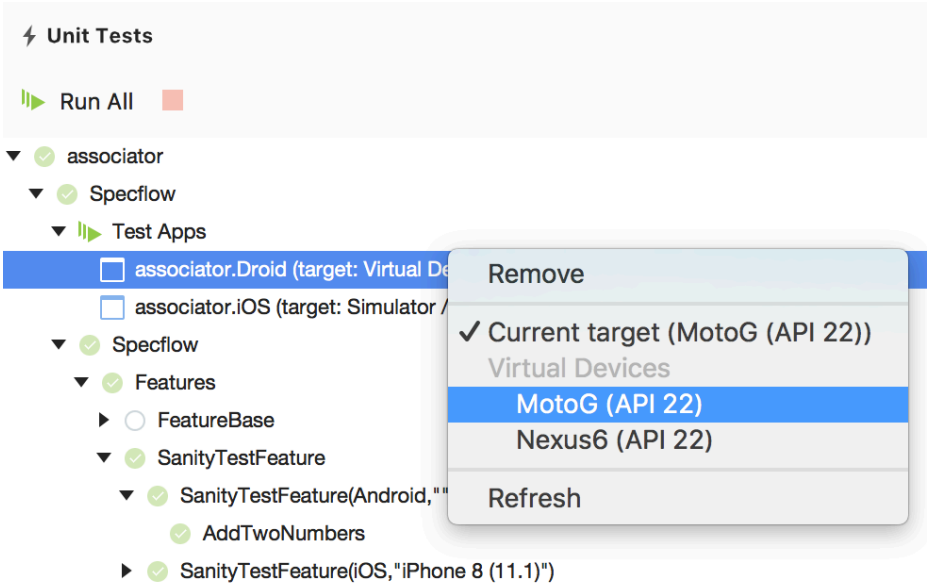
At this point, all configuration should be finished and Sanity tests should be able to run. Selecting on what device the tests will run is done through FeatureBase.cs file.

```
namespace Specflow.Features
{
    [TestFixture(Platform.Android, "", true)]
    [TestFixture(Platform.iOS, "iPhone 7 (11.2)", false)]
    //[TestFixture(Platform.iOS,"iPad Air (11.2)", false)]
    public class FeatureBase
    {
```

each [TextFixture()] line will result in one instance of FeatureBase being created. The first parameter is platform, the second parameter is name of iOS simulator (so, empty for Android) and the final one is a Boolean to decide whether to reset the simulator or not (set to false for now). For example, in the above sample code would result in the tests being created for both android and iPhone 7 (running iOS 11.2) platforms.

Selecting Android Emulator to run

In the UnitTest pad, right click on the android test and select the simulator you want



The FeatureBase file then needs to have a TextFixture annotation for running Android apps:

Selecting iOS Simulator to run

Selecting iOS Simulators are bit more complex. First, it depends on the Mac OSX machine and version. Assuming that XCode command line tools are installed (which they need to be), the list of available simulation devices can be obtained by executing the xcrun instruments command.

This command returns the name of the device followed by the GUID of the device and whether it is a device or a simulator. The output of this command should be used for selecting the device name, which is the second argument to the TextFixture annotation. For example, to run tests on iPhone X iOS 11.1 the command is executed as follows:

$ xcrun instruments -s devices

Apple TV (11.1) [D6652A16-8DC1-4E3B-9DE0-D7089A5EC65A] (Simulator)
Apple TV 4K (11.1) [8BD84528-1F8A-4C05-AF76-B0CE65CB0629] (Simulator)
Apple TV 4K (at 1080p) (11.1) [8423BB06-2729-49F0-8CCB-FE0A02958149] (Simulator)
Apple Watch - 38mm (4.1) [0851C081-A15B-49F1-B9F0-2250AB3AAE54] (Simulator)
Apple Watch - 42mm (4.1) [E3F400FD-E539-4FEC-8A9D-C1969032593B] (Simulator)
iPad (5th generation) (11.1) [BE1A2867-1952-48BF-B03B-0645B665270E] (Simulator)
iPad Air (11.1) [BAD66F4C-D752-4BA1-B8AA-472F79467ADC] (Simulator)
iPad Air 2 (11.1) [CE92903B-9F0F-44FB-94E0-9FFF1F075497] (Simulator)
iPad Pro (10.5-inch) (11.1) [499F044E-D4BC-41B1-B834-D492C9772E18] (Simulator)
iPad Pro (12.9-inch) (11.1) [F5334B38-AE95-4A31-A7A5-F509E0DBDA63] (Simulator)
iPad Pro (12.9-inch) (2nd generation) (11.1) [0E41A484-5652-493E-85C7-33D8AE81DCB3] (Simulator)
iPad Pro (9.7-inch) (11.1) [6632B986-CB51-4786-B8AA-C79DADB546FF] (Simulator)
iPhone 5s (11.1) [D37EF898-9DF8-446D-AC45-E911C86CB57F] (Simulator)
iPhone 6 (11.1) [010A136D-783F-4605-BEAF-803DB58A5CDD] (Simulator)
iPhone 6 Plus (11.1) [BFE74391-F2A6-48EB-BC7E-5533EBF2C1AA] (Simulator)
iPhone 6s (11.1) [0772CF98-67D6-423A-9D1E-782927FB41A1] (Simulator)
iPhone 6s Plus (11.1) [63EB8F74-3F8C-4486-AC99-60A72048E21A] (Simulator)
iPhone 7 (11.1) [B7D9A19B-D62E-40D8-9428-AFD1315A127B] (Simulator)
iPhone 7 (11.1) + Apple Watch Series 2 - 38mm (4.1) [491C61AE-B90A-45C5-8C8A-6C49543AB070] (Simulator)
iPhone 7 Plus (11.1) [1F52DCE5-191C-4424-B7DC-C4100DA51D98] (Simulator)
iPhone 7 Plus (11.1) + Apple Watch Series 2 - 42mm (4.1) [02AAB585-FE6F-494B-A747-E2039F9A56EB] (Simulator)
iPhone 8 (11.1) [E92C6E7A-7BA4-46B2-BC09-FABAB60A9013] (Simulator)
iPhone 8 (11.1) + Apple Watch Series 3 - 38mm (4.1) [15120BF0-E977-498A-A964-17FCA1061B3A] (Simulator)
iPhone 8 Plus (11.1) [15C369BE-69AF-4EC5-B7A7-F1FC22265EC0] (Simulator)
iPhone 8 Plus (11.1) + Apple Watch Series 3 - 42mm (4.1) [E1039D93-3B98-4DA7-8920-C61476A69CE9] (Simulator)
iPhone SE (11.1) [438728D5-AE0D-4DB5-ABD6-8345FD5202BC] (Simulator)
iPhone X (11.1) [3522DBD6-58EE-447E-A205-DBD5EAF2CDFB] (Simulator)

Thus, the simulator name will be “iPhone X (11.1)”. this can be put directly in the TextFixture annotation:

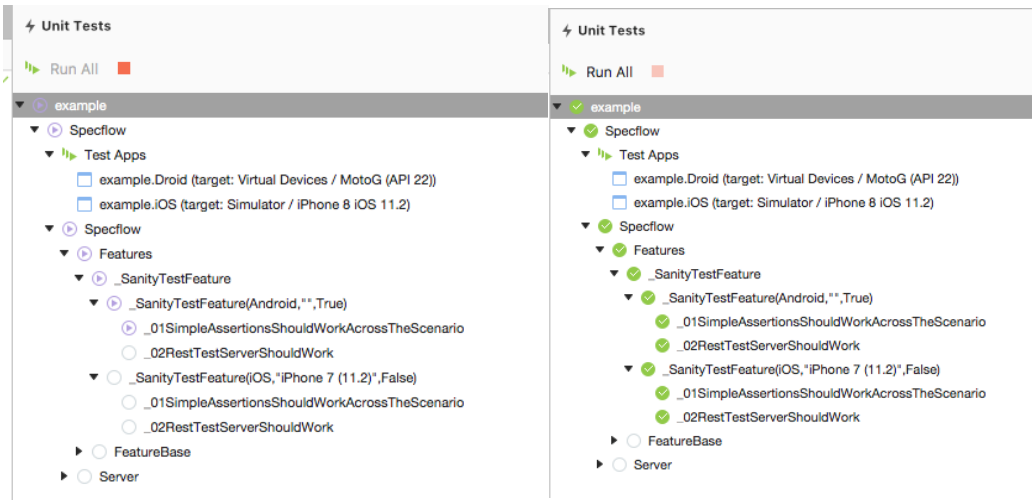
```
namespace Specflow.Features
{
    [TestFixture(Platform.Android, "", true)]
    [TestFixture(Platform.iOS,"iPhone X (11.1)", false)]
    public class FeatureBase
    {
```

Running tests on Xamarin Test Cloud

[To be added]

Running tests locally

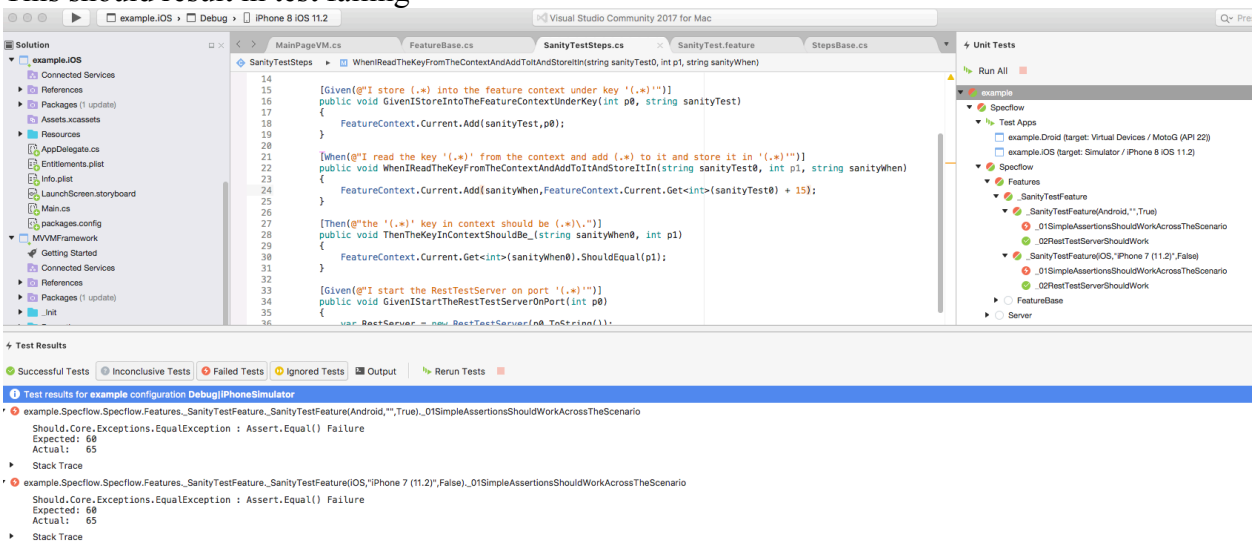
At this point, what is needed is to run the tests locally is to open the Unit Tests pad and click on Run All button:



to be sure, change the assertion in SanityTestSteps.cs file to look for something different than the expected value, for example, add 10 to it:

```
 [When(@"I read the key '(.*)' from the context and add (.*) to it and store
it in '(.*)'")]
        public void WhenIReadTheKeyFromTheContextAndAddToItAndStoreItIn
            (string sanityTest0, int p1, string sanityWhen)
        {
           FeatureContext.Current
            .Add(sanityWhen,FeatureContext.Current.Get<int>(sanityTest0)+15);
        }
```

This should result in test failing



This is the final part of the master branch. It is tagged as boilerplate and is used as the basis for more examples in the other branches.

The rest of this document contains general information on building apps using the BDD technique which applies to all branches and is a recommended reading.

Section 2- General Techniques in Developing the Application

The TDD/BDD protocol

Application is developed following the TDD protocol, i.e. first a test is written against the required behavior, which will fail. Then enough code will be added until the test passes, then the code is refactored to comply with coding standards. The cycle is then repeated, with test followed by code until test passes followed by refactor, until all the requirements and features are implemented.

The technique has a three nuances. First, one should stop writing test as soon as one can make the test fail. For example, once you notice you need an object that is not defined, that is a failing test. You should stop and go to the next step.

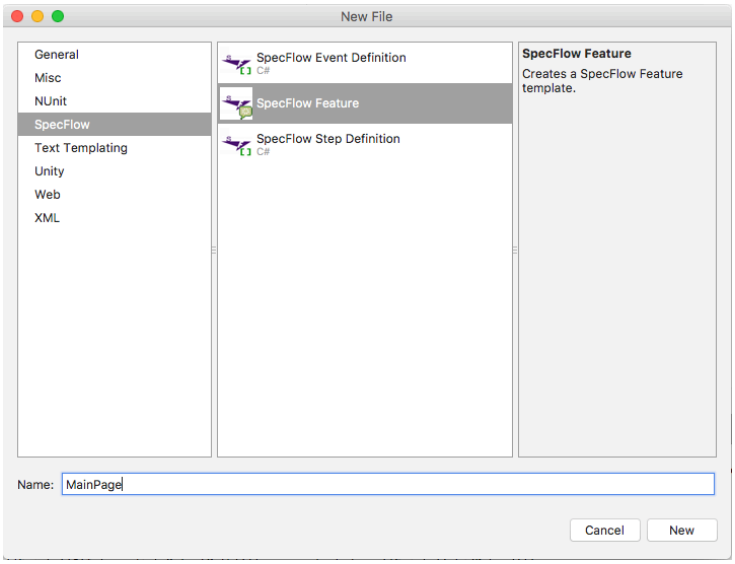
Second, one should stop writing code as soon as the test no longer fails. So once you define that object, you should not continue implementing all its method. You should go back to test writing until one of your tests fails because it needs a method on the object.

Third, once the test pass, both code and test code need to be refactored. It is important to apply refactoring only on the passing test scenario so that the codebase can be reset to the passing stage.

If you are using Git, that means committing at end of each of the above three steps.

Writing the smallest failing test:

If you don't already have a feature for this scenario, add a feature file to the features directory of the Specflow project.



Write the first scenario and assign a decorator to it.

```
Feature: Main Page
    I wish to ensure that the structure and behaviour of the main page is correct

@structure
Scenario: Should be able to start association
    Given I am running the app
    When I examine the main page
    Then It has a button marked as 'Associate'
```

Save the file. This will generate the first part of partial class for feature.
Create the second part of partial class for the feature in the feature directory

- This second part is a subclass of FeatureBase class.

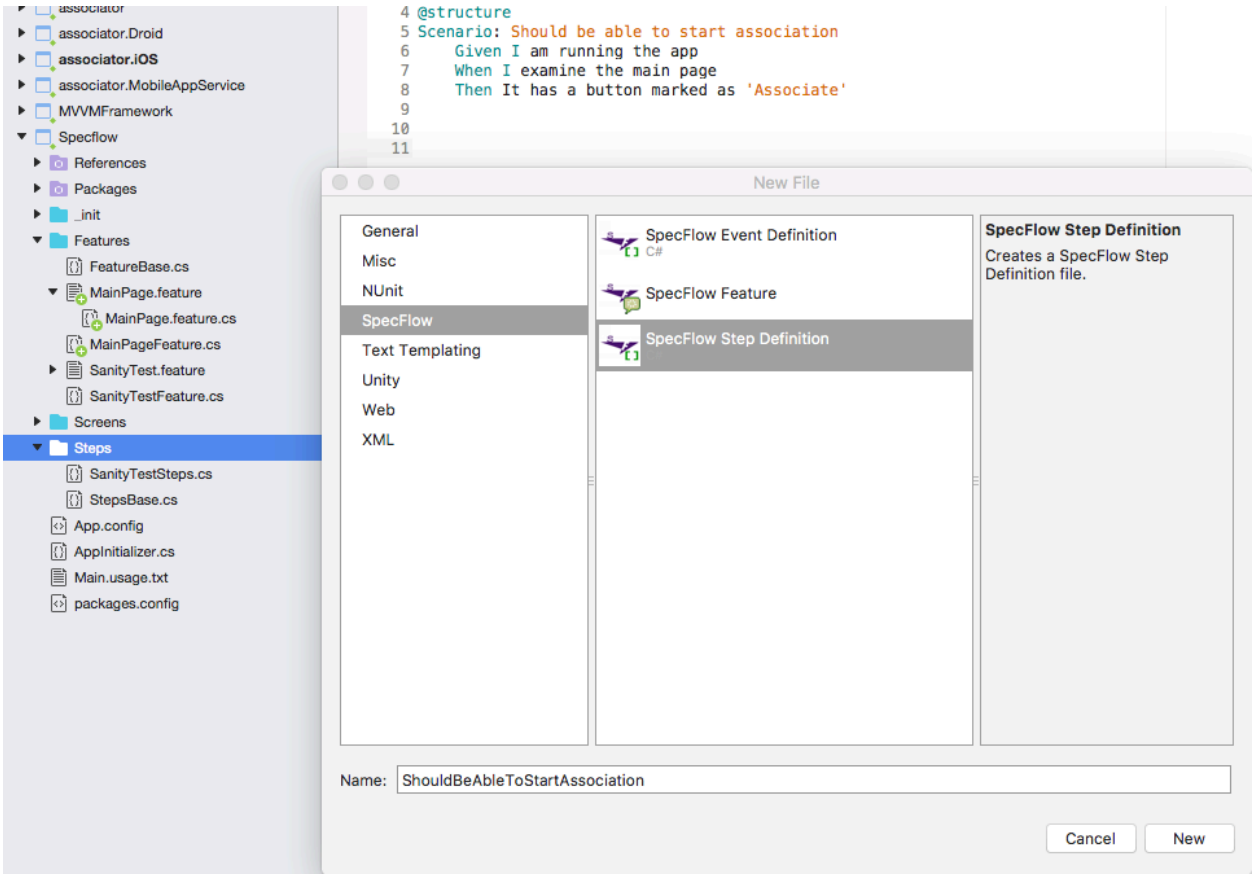
```
using System;
using Xamarin.UITest;

namespace Specflow.Features
{
    public partial class MainPageFeature : FeatureBase
    {
        public MainPageFeature(Platform p, string iOSSimulator, bool resetSim)
            :base(p,iOSSimulator,resetSim)
        {
        }
    }
}
```

Run the test. This will fail but in Application Output pane it will print useful templates for step functions.

```
Deploying to Device | Application Output - Unit Tests

Signing apk with Xamarin keystore.

Skipping installation: Already installed.

Given I am running the app

-> No matching step definition found for the step. Use the following code to create one:
        [Given(@"I am running the app")]
        public void GivenIAmRunningTheApp()
        {
            ScenarioContext.Current.Pending();
        }


When I examine the main page

-> No matching step definition found for the step. Use the following code to create one:
        [When(@"I examine the main page")]
        public void WhenIExamineTheMainPage()
        {
            ScenarioContext.Current.Pending();
        }


Then It has a button marked as 'Associate'

-> No matching step definition found for the step. Use the following code to create one:
        [Then(@"It has a button marked as '(.*)'")]
        public void ThenItHasAButtonMarkedAs(string associate)
        {
            ScenarioContext.Current.Pending();
        }
```

There, we have the smallest failing test. But it is failing because of test code, not production code, so let's continue with writing more test code.

Add a Steps file to the Steps directory. One convention to use is to set the name the same as the scenario or it's tag:



In the source code, set the superclass to StepsBase, and use the template of the test runner to create steps for the scenario.

```
using System;

using Should;
using Xamarin.UITest;
using TechTalk.SpecFlow;

namespace Specflow.Steps
{
    [Binding]
    public class ShouldBeAbleToStartAssociation : StepsBase
    {
        [Given(@"I am running the app")]
        public void GivenIAmRunningTheApp()
        {
           app.ShouldNotBeNull();
        }

        [When(@"I examine the main page")]
        public void WhenIExamineTheMainPage()
        {
            ScenarioContext.Current.Pending();
        }

        [Then(@"It has a button marked as '(.*)'")]
        public void ThenItHasAButtonMarkedAs(string associate)
        {
            ScenarioContext.Current.Pending();
        }
   }
}
```

This is now a failing step because of production code, since there is no way to check if the test is examining the main page. Very frequently the test needs to know about specific pages or components on the page.

Making the test pass with minimum code

Xamarin.UITest allows for components to have a “AutomationId” attribute, which the test can query to find or check for the component they need. We have added the AutomationId as part of the code when we created the MainPage.

```
public partial class MainPage : ContentPage
{
    public const string PageKey = PageKeys.MainPage;
    public MainPage()
    {
        InitializeComponent();
        AutomationId = PageKey;
        BindingContext = Initializer.GetDependency<MainPageVM>();
    }
}
```

As will be shown later, AutomationId can be added to the elements in the Xaml markup as well. But at this point, we can refactor the step to check if such element exists.

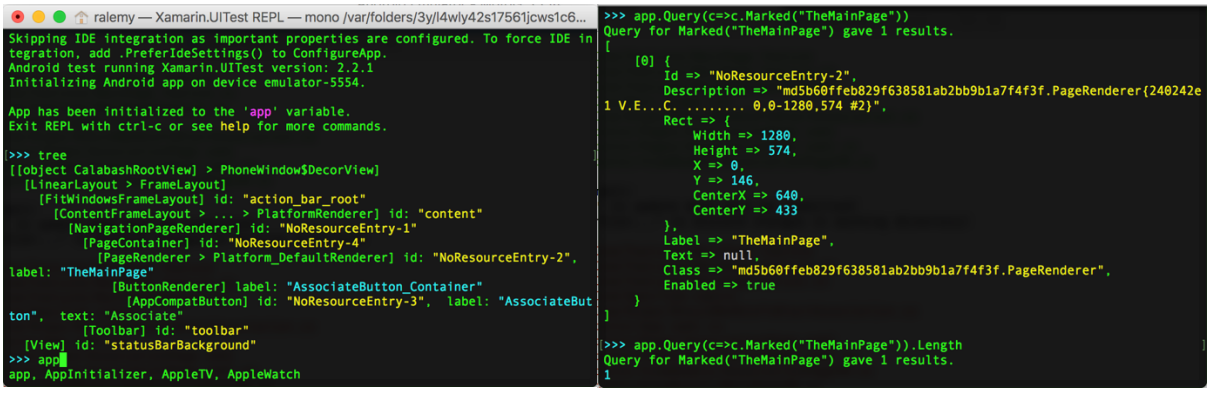
```
[When(@"I examine the main page")]
public void WhenIExamineTheMainPage()
{
    app.Query(c=>c.Marked(PageKeys.MainPage)).Length.ShouldBeGreaterThan(0);
}
```

Using app.Repl()

To detect the elements of the page and correct statement to apply for the tests, Xamarin allows for a very useful command called REPL. Essentially, when the test contains this statement, Xamarin stops and brings up a terminal for the user to type commands against the app and observe the responses so that the user can decide which command is best to detect the feature in which the user is interested. For example, the above test should have started like this:

```
[When(@"I examine the main page")]
public void WhenIExamineTheMainPage()
{
    app.Repl();
}
```

Once the execution reaches this line, Xamarin will stop and display a terminal that can be used to execute commands and see their results. This terminal supports command completion and copying history to clipboard, which are very useful to navigate complex structures.



This shows what the correct test should be. Now the user can exit the Repl by typing quit or exit.

Refactor, and Repeat.

We can then use the same technique to check for a button that is marked as expected:

```
[Then(@"It has a button marked as '(.*)'")]
public void ThenItHasAButtonMarkedAs(string associate)
{
    app.Query(c=>c.Marked("AssociateButton")).Length.ShouldEqual(1);
}
```

Which will fail, and we should go and add the button to the page.

```
<StackLayout Padding = "30">
    <Button AutomationId="AssociateButton" Text="Associate" />
</StackLayout>
```

At this point, the tests will all pass. We have proven that the button is there and it has the right label. Next, is to prove it will work.

Adding Button Actions to the application

To allow a button to execute a command, three components work with each other.

- In the xaml file, the command attribute of the button is set to a binding

```
<Button AutomationId="AssociateButton" Text="Associate" Command="{Binding GoForAssociation}"  />
```

- In the ViewModel file, a public command variable is declared which will receive the click or tap as part of binding.

```
public class MainPageVM : ViewModelBase
{
    public ICommand GoForAssociation { get; private set; }
```

- The ICommand is initialized in the constructor, and pointer to a method or action is sent to it. This method is where the code for the command exists, and it can be executed and tested with unit tests.

```
public class MainPageVM : ViewModelBase
{
    public ICommand GoForAssociation { get; private set; }

    public MainPageVM(){
        this.GoForAssociation = new RelayCommand(_GoForAssociation);
    }

    private void _GoForAssociation(){
           //Code for command
    }

}
```

Navigation to other pages

If the button is supposed to trigger navigation to another page, then the other page should be created as explained above:

- The xaml.cs file should have a public static PageKey set to a value defined in MVVMFramework.Statics so that it can be shared with the test base. The PageKey should be assigned as the AutomationId of the topmost element in the page constructor. The xaml.cs also file should have a BindingContext set to the ViewModel object retrieved from the dependency injector.

```
public partial class AssociationPage : ContentPage
{
    public const string PageKey = PageKeys.AssociationPage;
    public AssociationPage()
    {
        InitializeComponent();
        AutomationId = PageKey;
        BindingContext = Initializer.GetDependency<AssociationPageVM>();
    }
}
```

- The Xaml file for the page could add AutomationId to any other element that is subject of a test.

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage  xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.microsoft.com
/winfx/2009/xaml" x:Class="associator.Pages.AssociationPage">
    <ContentPage.Content>
        <StackLayout>
        <Label AutomationId="AssociationPageLabel" Text="This is the association page"/>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

- The ViewModel file must be a subset of ViewModelBase object.

```
public class AssociationPageVM : ViewModelBase
    {
        public AssociationPageVM()
        {
        }
    }
```

- Finally, in App.xaml.cs the Page object should be registered with the navigation manager under the PageKey and the ViewModel should be registered with the dependency injector.

```
 private Page RegisterPages(NavigationPage page)
        {
            var nav= Initializer.GetDependency<INavigationManager>();
            nav.SetMain(page);
            nav.Register(AssociationPage.PageKey, typeof(AssociationPage));
            return page;
        }
private void RegisterDependencies()
        {
            Initializer.SetupDI();
            Initializer.Register<MainPageVM>();
            Initializer.Register<AssocitionPageVM>();
        }
```

Once such page exists, the ViewModel that controls the command of the button can switch to that page. The navigator is received from the dependency injection system.

```
public MainPageVM(INavigationService navigator){
    this.navigator = navigator;
    this.GoForAssociation = new RelayCommand(_GoForAssociation);
}

private void _GoForAssociation(){
    navigator.NavigateTo(AssociationPage.PageKey);
}
```

Testing a new page

One of the more popular practices in BDD and end-to-end testing of multi-page applications is to create a Page object. The page object takes care of how elements are addressed and found. This technique has multiple advantages in maintainability and scalability. For example, if the text of the button changes, there would be no need to refactor all tests that are referring to that button.

Creating a Page Test Object

A page test object is a plain C# object that contains methods for identifying the page, making sure it has been loaded, and identifying pertinent components on the page. In the example of AssociationPage

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage  xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.microsoft.com
/winfx/2009/xaml" x:Class="associator.Pages.AssociationPage">
    <ContentPage.Content>
        <StackLayout>
        <Label AutomationId="AssociationPageLabel" Text="This is the association page"/>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Create a new C# class in Specflow.PageTestObjects.Pages to represent the Association Page. This should be a subclass of AppPageTO. This class overrides KeyForPage to return the static key for the association page, since both the shared project and the Specflow project have access to that key. It also needs to implement a method to show how to navigate to it from Main page.

```
using MVVMFramework.Statics;
using Xamarin.UITest;

namespace Specflow.PageTestObjects.Pages
{
    public class AssociationPageTO : AppPageTO
    {
        public AssicationPageTO(IApp app) : base(app)
        {
        }

        protected override string KeyOfPage => PageKeys.AssocaitionPage;

        public override void NavigateFromMain(AppPageTO main)
        {
        }
    }
}
```

The AppPageTO base class provides useful shared functions for testing pages. For example, IsAndroid() would determine at runtime of the test whether android or iOS device is tested. WaitForLoad() will delay testing until page is fully loaded, etc.

Scenario for a button navigating to page

As described above, a scenario is added to the MainPage feature file, after the scenario that makes sure the correct button exists.

```
@function
Scenario: Should navigate to association page
    Given I am in main page
    When I tap the 'Associate' button
    Then I go to the association page
```

Run the test once to generate the template and then add a Specflow Step file as explained above and write the test code. Here is how it should look like.

```
public class ShouldNavigateToAssociationPage: StepsBase
 {
     [Given(@"I am in main page")]
     public void GivenIAmInMainPage()
     {
         app.Query(c=>c.Marked(PageKeys.MainPage)).Length.ShouldBeGreaterThan(0);
          ScenarioContext.Current.Set<AssociationPageTO>(new AssociationPageTO(app));
     }

     [When(@"I tap the '(.*)' button")]
     public void WhenITapTheButton(string associate)
     {
         app.Tap(c=>c.Marked(associate));
     }

     [Then(@"I go to the association page")]
     public void ThenIGoToTheAssociationPage()
     {
         var page = ScenarioContext.Current.Get<AssociationPageTO>();
         page.WaitForLoad();
         app.Query(page.PageContainer).Length.ShouldBeGreaterThan(0);
     }

 }
```

See that the first test is not using a Page test object, and needs to know the exact id of the page container. But the second page is using a Page test object, so it can use code completion, and also in the future the way that the PageContainer is detected may change without requirement to refactor the tests.

It is important to remember that the test steps would be shared between scenarios when needed. This means that the Step Classes **Should Not** use public properties, because those will not be available to other step classes. Instead, they should use ScenarioContext to share data with other steps of the scenario. In the example above, the PageTestObject for the association page is stored in ScenarioContext in the the first step and retrieved in the last step. If steps need to share data across Scenarios, they can use FeatureContext.

Section 03: Next Steps

This repository contains branches that continue the demonstration to show how different features can be added to the app using the BDD approach and framework described here. For each of these branches there is a file in the Docs directory that explains steps to be taken to add such features.