Maintaining a WP7 and WP8 version of a same Silverlight application

This article provides some tricks and tips for maintaining a Windows Phone 8 version of a Silverlight application along with the legacy Windows Phone 7 version.

Note: The section #Adding WP8 features to a WP7 Application : Sample Application contains content merged from the article **Maintaining a WP7 and WP8 version of a same Silverlight application** by r2d2rigo

Note: This article was a winner in the Windows Phone 8 Wiki Competition 2012Q4.

Introduction

Now that Windows Phone 8 is available, a new challenge arises for Windows Phone developer: fragmentation. If you want to use the great new features of Windows Phone 8 but you still want to target your large Windows Phone 7 user base, you may need to maintain both a WP7 version and a WP8 version of your application. There is no "silver bullet" solution to this problem, but there are a number of approaches you can use to reduce the maintenance cost.

This article outlines a number of tricks and tips for managing your application versions, which you should use and combine depending on your needs:

- Sharing code files between your projects
- Adding async/await support to your Windows Phone 7 project
- Use of conditional symbols to add platform-specific pieces of code
- Set custom styles
- Using a custom URI mapper to redirect to specific XAML page depending on the platform
- Adding WP8 features to an existing WP7 Application

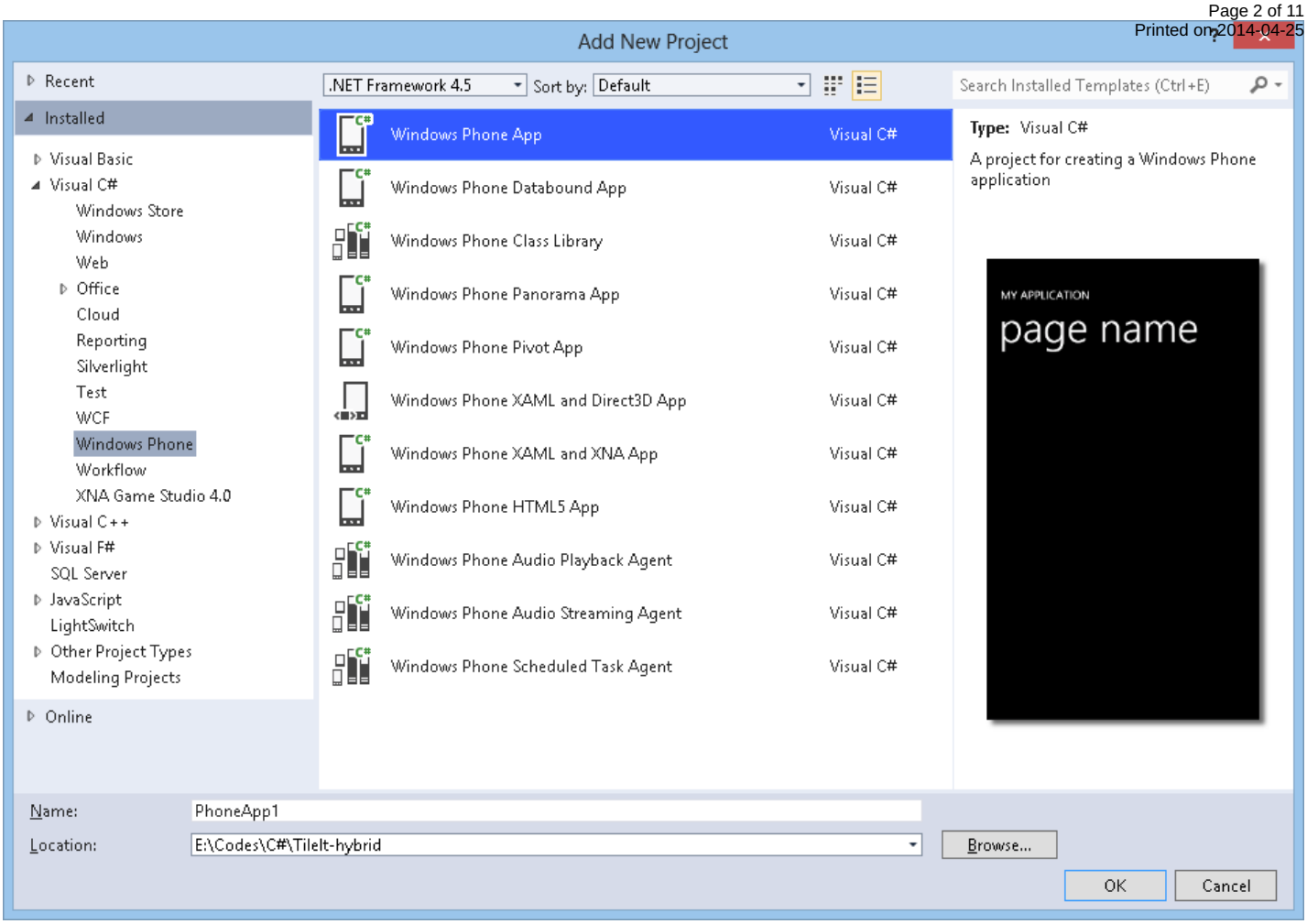
Setting up the projects and sharing code files

To build a Windows Phone 8 app, the first thing you need is... A WP8 project. The approach I use is having a WP7 and a WP8 project for the same application, sharing most of the source files. Then I try to develop as often as possible directly in the WP7 project, as it has the smallest subset of features. If something works on WP7, it'll most likely work on WP8.

So, how to create a Windows Phone 8 project out of your existing Windows Phone 7 project?

First step, create the actual Windows Phone 8 project. To do so, open your Windows Phone 7 solution, and add a new project targeting Windows Phone 8.

(C) Copyright Nokia 2014. All rights reserved.

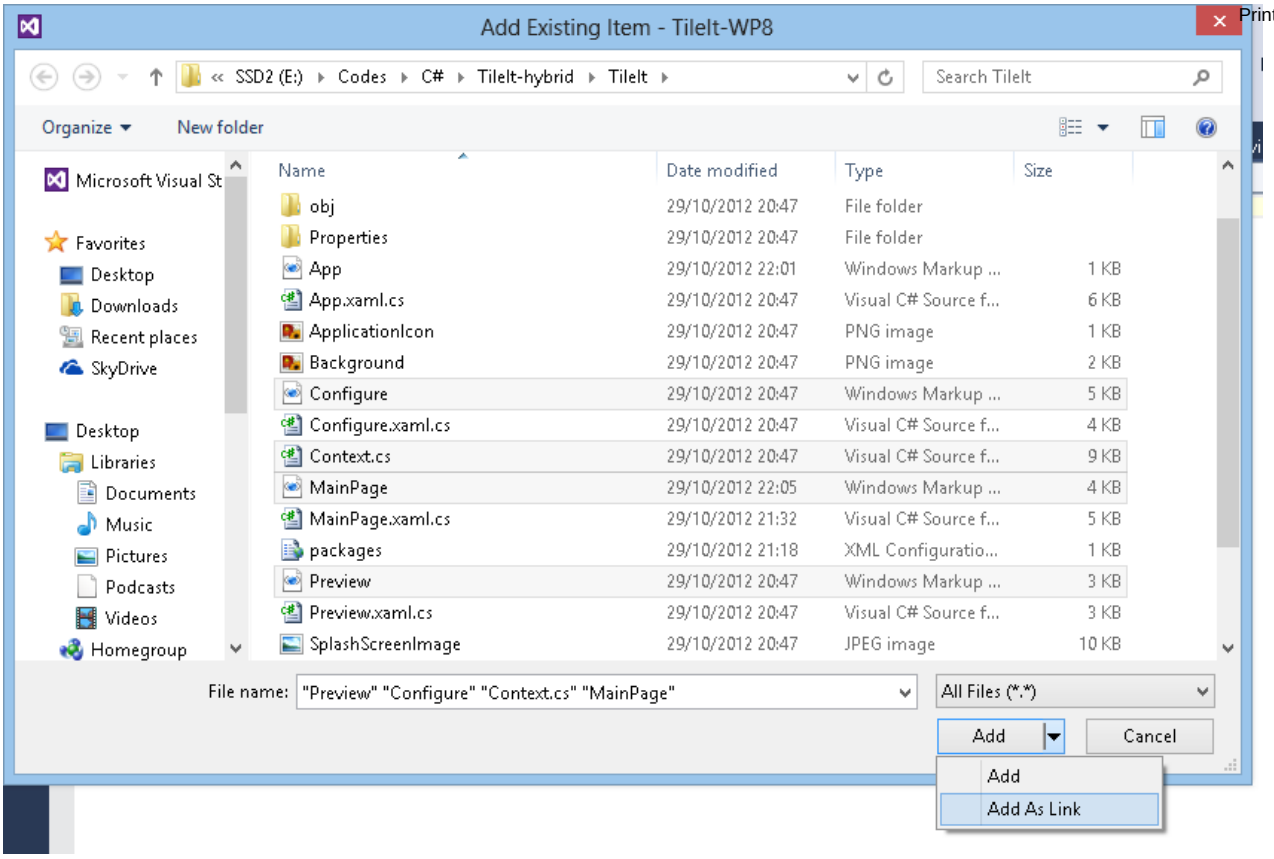


Remove the default **MainPage.xaml/cs** files from the project, then link the pages and classes from your WP7 project to your new project, as shown below:

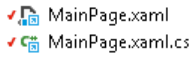
- Right-click on the project
- Choose **Add** then **Existing item**'
- Go to the WP7 project folder, select all the XAML and C# files you want to share
- Click on the small arrow on the **Add** button
- Select **Add as link**

You'll want to share most of the .cs and .xaml files of your WP7 project. The only ones that shouldn't be shared are those who contain a lot of platform specific code (in which case it may be easier to create a whole new file for WP8, as we'll see in the next sections of this article). Also, I recommend not sharing the **App.xaml** and **App.xaml.cs** files, as it may actually make some things easier to do, like customizing the styles.

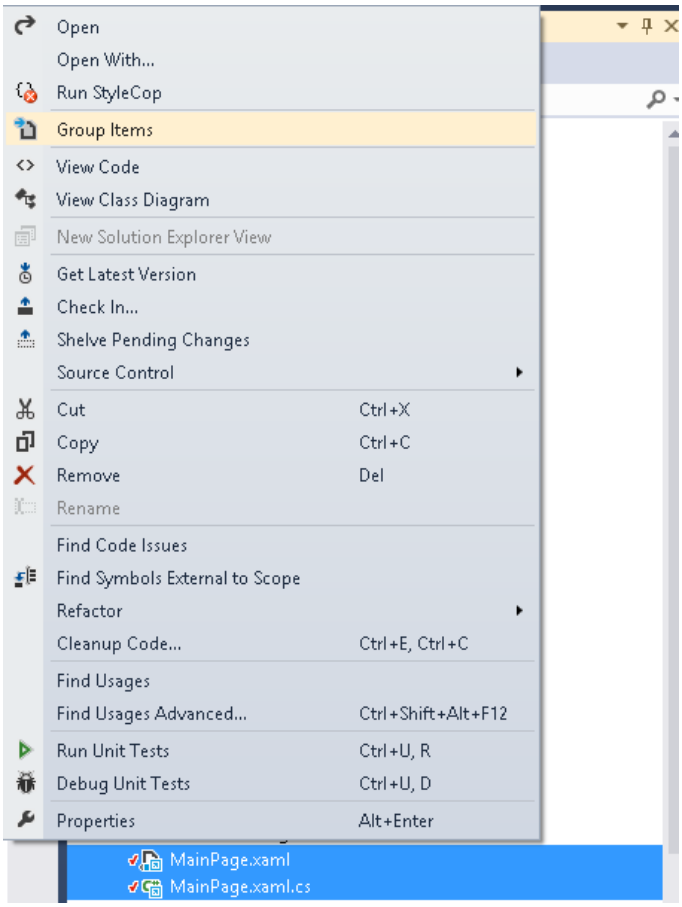
 (C) Copyright Nokia 2014. All rights reserved.



Tip: You will most likely want to share some XAML pages and their .cs code-behind file. If you add both files as link, they won't appear grouped in Visual Studio's solution explorer:



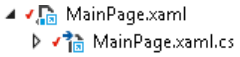
If you want them to be grouped, like they were in the original WP7 solution, you can either just add the .xaml file (Visual Studio will automatically add the .cs file), or use the free Visual Studio extension called *VSCommands*. It adds an option to the contextual menu that you can use to group your files. Just select the .xaml and .cs file, right-click, and click on **Group items**:



Then select the .xaml file as root item:

(C) Copyright Nokia 2014. All rights reserved.

Click on **Ok**, and the files are now grouped!



Now the physical files you've selected are shared between the two projects. It means that if you make changes to one of those files in a project, it'll be automatically updated in the other project (since it's actually the same file).

Up to this point, you should be able to compile the WP8 project (after adding the missing references and updating the manifest if needed) and run it. Now let's start adding specific features to the WP8 project.

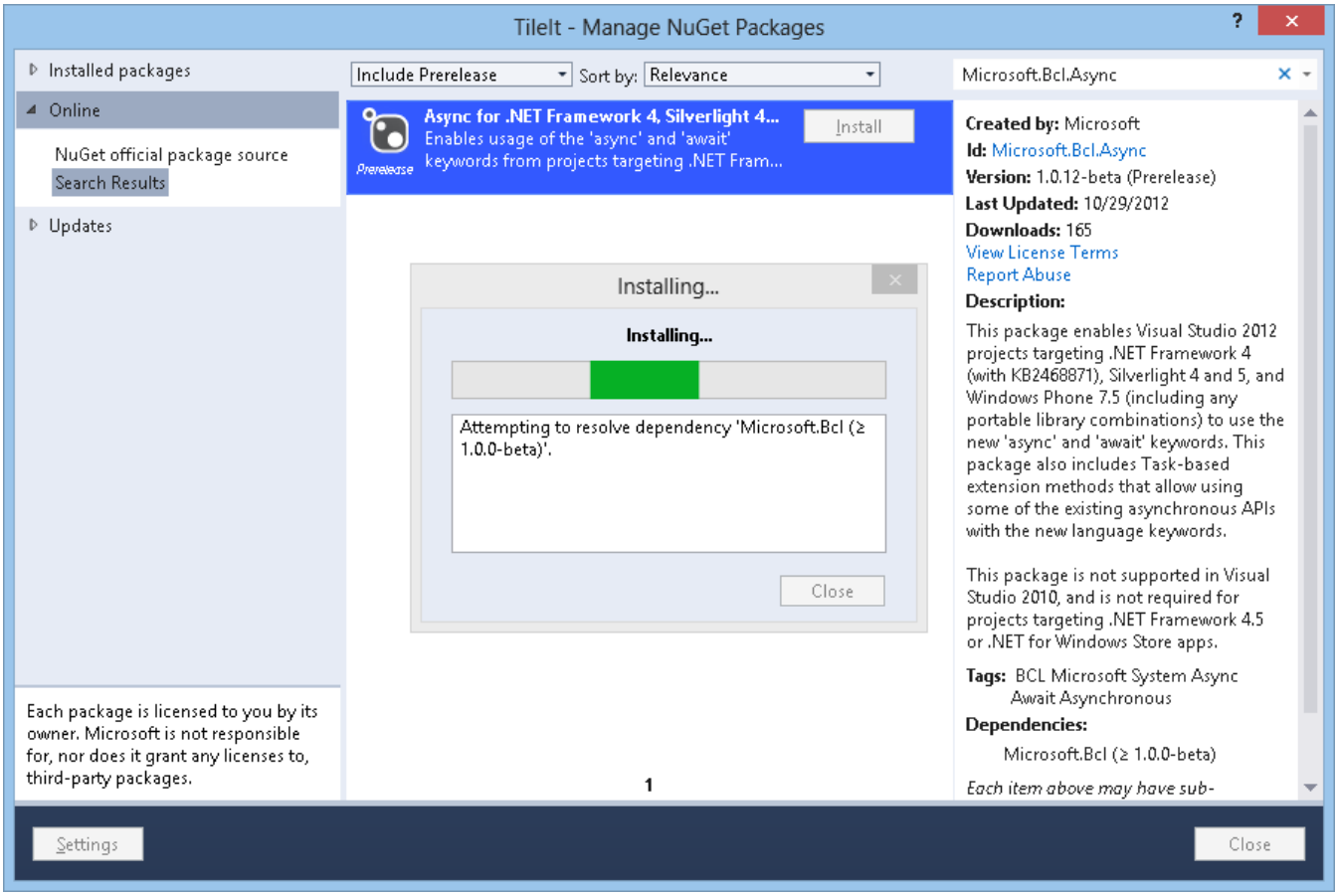
Adding async/await support to your Windows Phone 7 project

One of the best new feature of .NET 4.5 has been made available for Windows Phone 8: the async and await keywords. Thankfully, the BCL team has published a NuGet package to make those keywords available for WP7 as well.

Installation steps are detailed in this blog post.

Overall, all you need to do is to search *Microsoft.Bcl.Async* in NuGet, and install the *Async for .NET Framework 4, Silverlight 4 and 5, and Windows Phone 7.5* package. You can also install it directly from the NuGet console by typing:

```
install-package Microsoft.Bcl.Async -pre
```



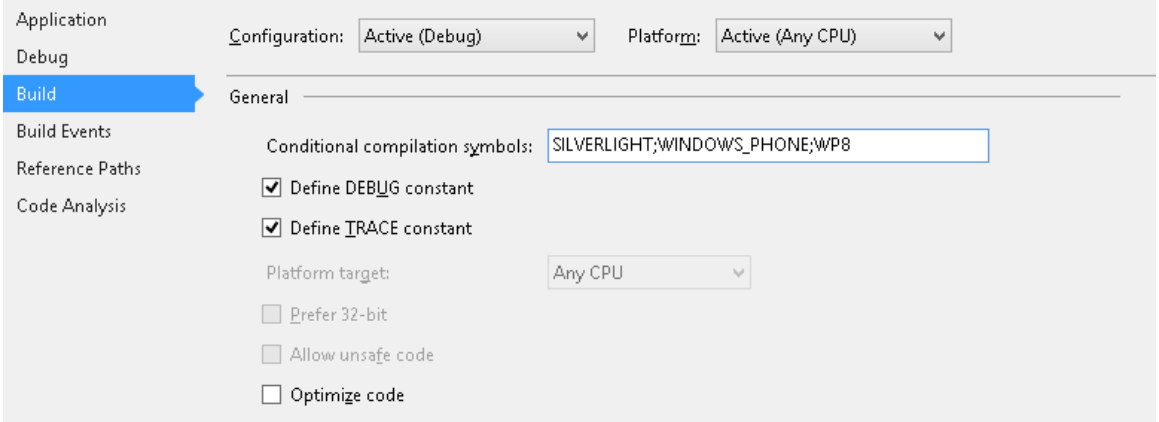
Of course, you need to add this package only to the WP7 project, as the WP8 one has already native async/await support. Once the package is installed, you can use tasks and the async/await keywords from both the WP7 and WP8 projects!

Use of conditional symbols for platform-specific code

 (C) Copyright Nokia 2014. All rights reserved.

It's no surprise that there will come a moment when you'll be using APIs that can't be back-ported to WP7. How do we call those new methods without creating compilation errors in the WP7 project? *By using preprocessor directives.*

Right-click on the WP8 project, go in the properties, then in the Build tab. There, you can see a 'Conditional compilation symbols' textbox, with per default "SILVERLIGHT;WINDOWS_PHONE". Edit it to add "WP8". The new complete value should then be "SILVERLIGHT;WINDOWS_PHONE;WP8".



Now, whenever you use a specific WP8 API, you can enclose your code using the directives `#IF WP8` and `#endif`. That way, those portions of code will be ignored when compiling the WP7 project!

For instance, let's say we want to call the new `Application.Terminate()` API only when executing the WP8 app. Then this code should do the trick:

```
#if WP8
Application.Current.Terminate();
#endif
```

Custom Styles

You may want to apply different styles to your XAML page depending on whether you're aiming WP7 or WP8, for instance to hide buttons. If your **app.xaml** files aren't shared, then you can simply declare your styles there in that project. However, if they are shared, you can reference an external resource dictionary.

On both projects:

- Right-click on the project
- Select **Add**, then **New Item**
- Select **Text file** and name it "Styles.xaml"

Then, in the solution explorer, select the file, look at the properties, and set the 'build action' to **Content**.

Open your two **Styles.xaml** files, and add the following XAML code:

```
<ResourceDictionary xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">

</ResourceDictionary>
```

Then, open the App.xaml file, and reference the newly created ResourceDictionary:

```
<Application.Resources>
 <ResourceDictionary Source="Styles.xaml" />
</Application.Resources>
```

And you're done! Now each project has its separate **Styles.xaml** file where you can put your custom styles.

How does it work? Let's say you've got a button that you want to style differently depending on whether you're executing the WP7 or the WP8 app.

First, apply the style to the button:

(C) Copyright Nokia 2014. All rights reserved.

```
<Button Style="{StaticResource TestStyle}" x:Name="ButtonContinue" Content="Continue" />
```

Then create the corresponding style in the **Styles.xaml**. For instance, in the WP7 one:

```
<ResourceDictionary xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Style x:Key="TestStyle"  TargetType="Button">
        <Setter Property="Foreground" Value="Blue" />
    </Style>
</ResourceDictionary>
```

Then in the WP8 one:

```
<ResourceDictionary xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Style x:Key="TestStyle"  TargetType="Button">
        <Setter Property="Foreground" Value="Red" />
    </Style>
</ResourceDictionary>
```

Execute the project, and you'll see that the button's text is displayed in blue when executing the WP7 app, and in red when executing WP8 app. It's a convenient way to display or hide UI elements depending on which application is running.

Specific pages and URI mapper

Sometimes, conditional symbols and custom styles just aren't enough, it may be easier to create a whole new XAML page for Windows Phone 8.

There are two main cases where you could require to use specific pages:

1. You're adding a completely new feature to your app, only for Windows Phone 8. For instance, if you want to add NFC sharing which isn't supported by Windows Phone 7. Then there's no point in sharing a common XAML file between WP7 and WP8 since the WP7 one won't be able to do anything.

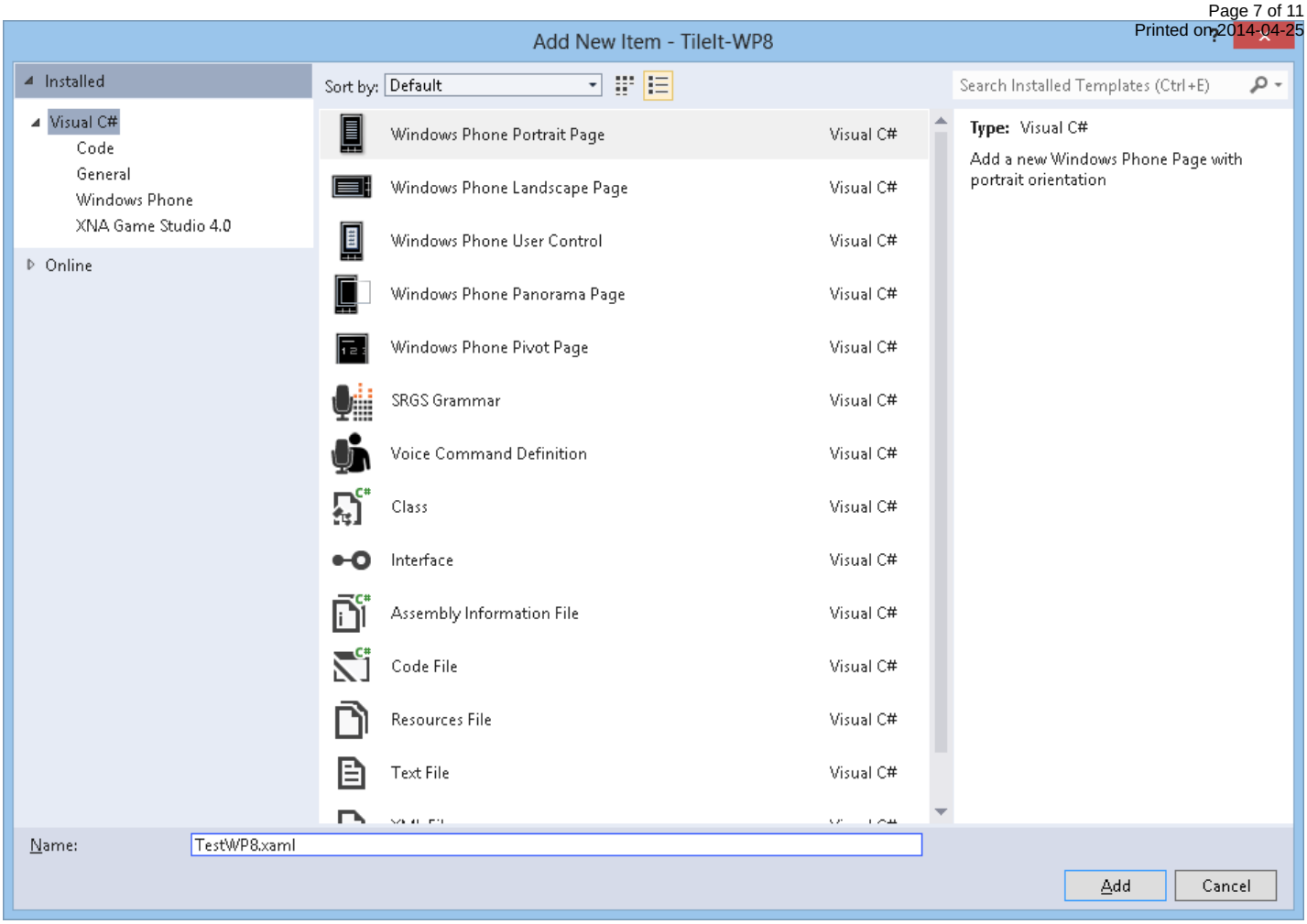
   In that case, it's rather straightforward: just add the new page to the WP8 project. Then, use conditional symbols or styles to add the button to navigate to that page.

2. You're dealing with features that exist in both version of your app, but the WP8 version requires a lot of specific code. For instance, if you're rewriting a page to use the brand new map control. In that case, you'd end up enclosing most of your page's code in conditional symbols, and it'll quickly become a nightmare to maintain. In that case, you're better out creating a complete new page.

The obvious way to do that is to remove the linked version of the file, and add a project-specific file (as you would do in a 'normal' project). However, you'll end up with files named the same way in your WP7 and WP8 projects, some of them being shared and the others not. To avoid any confusion, it's better to give them different names.

So let's say we have a page called **Test.xaml** in our WP7 project. We want to completely rewrite this page in the WP8 project. First, we add a new page called **TestWP8.xaml**.

 (C) Copyright Nokia 2014. All rights reserved.



Now, at some point in the WP7 project, you want to redirect the user to this page:

```
NavigationService.Navigate(new Uri("/Test.xaml", UriKind.Relative));
```

How to redirect to **TestWP8.xaml** in the WP8 version of the application? If you've read the previous chapters, you know you can do that using conditional symbols:

```
#if WP8
            NavigationService.Navigate(new Uri("/TestWP8.xaml", UriKind.Relative));
#else
            NavigationService.Navigate(new Uri("/Test.xaml", UriKind.Relative));
#endif
```

However, you may want to handle those specific redirections in a centralized way, especially if there are many ways in the app to reach the Test.xaml page. We can do that using a custom URI mapper.

What is a URI mapper? It's a class used by Silverlight to associate a page to an URI. When you call `NavigationService.Navigate`, the framework will use the URI mapper to which page is designated by your URI.

Making a custom URI mapper is quite simple. Add a new class to the WP8 project, and call it 'CustomUriMapper'. The class must inherit from `UriMapperBase`. Then we override the `MapUri'` method and we check which URI is called. If it's **Test.xaml**, then we redirect to our new **TestWP8.xaml**' page. Otherwise, we just return the original URI:

```
public class CustomUriMapper : System.Windows.Navigation.UriMapperBase
{
    public override Uri MapUri(Uri uri)
    {
        if (uri.OriginalString.StartsWith("/Test.xaml"))
        {
            return new Uri("/TestWP8.xaml", UriKind.Relative);
```


(C) Copyright Nokia 2014. All rights reserved.

```
        }

        return uri;
    }
}
```

Now, we just need to replace the original URI mapper by our custom one. In the **App.xaml.cs** file, in the constructor, search the line calling the `InitializePhoneApplication` method. Just after that line, replace the URI mapper of the root frame:

```
// Phone-specific initialization
InitializePhoneApplication();

// Replace the default URI mapper
RootFrame.UriMapper = new CustomUriMapper();
```

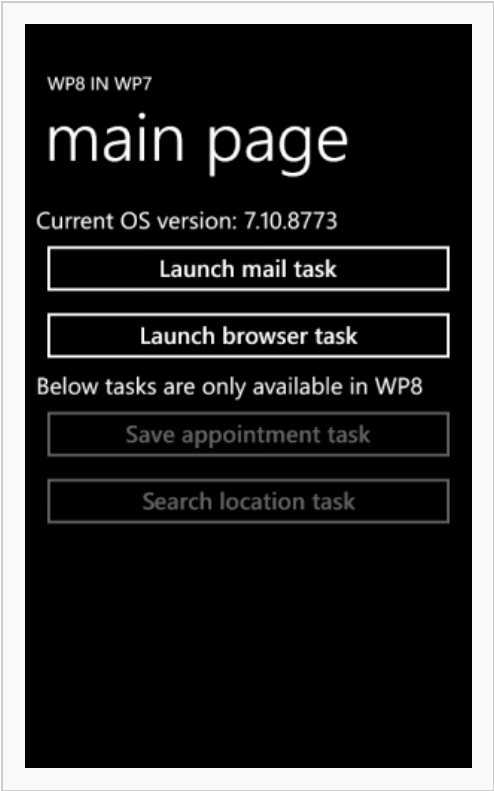
And we're done! Each time the application will try to navigate to Test.xaml, it'll be automatically redirected to TestWP8.xaml.

Adding WP8 features to a WP7 Application : Sample Application

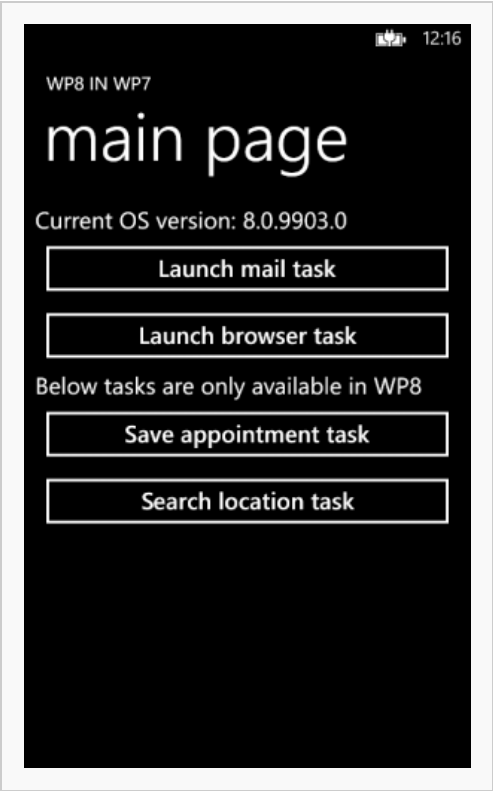
In this sample application we are going to learn how to add some Windows Phone 8 functionality in an existing Windows Phone 7 application through the use of reflection, and execute only when we are running under the correct operating system.

Chances are you already have a popular app published for Windows Phone 7 and want to add new Windows Phone 8 features to it without having to release a separate version, creating fragmentation and losing all you hardly earned positive user reviews. We can achieve this by checking the current operating system version at runtime and then loading the appropriate assembly using Reflection to search for the specific types we need to use.

For this demonstration we are going to create a very simple application that only has one page, a `TextBlock` to display the operating system's version and four buttons, two for tasks already available in Windows Phone 7 and rest of the two for new tasks introduced in Windows Phone 8.



Application running under WP7



Same application running under WP8

Specifying our viewmodel

The viewmodel (which is based on MVVM Light to simplify development) will have the following properties:

- `System.Version OsVersion`: returns the value of `System.Environment.OSVersion.Version`.

(C) Copyright Nokia 2014. All rights reserved.

- `bool IsWp8`: returns **true** if `OsVersion.Major` equals 8.
- `bool CanLoadTypes`: returns **true** if `IsWp8` is true and the internal field `wp8Assembly` is not null (meaning that we could load the assembly without problems).

To store a reference to the assembly, we also need a private field of type `System.Reflection.Assembly` called `wp8Assembly`. We will wrap some helper functions around it so we don't need to publicly expose it. Now, let's fill in the constructor:

```
public MainViewModel()
{
    // Load the assembly if we are in WP8.
    if (this.IsWp8)
    {
        try
        {
            wp8Assembly = Assembly.Load(Wp8AssemblyFullName);
        }
        catch (Exception e)
        {
            MessageBox.Show("Couldn't load '" + Wp8AssemblyFullName + "' assembly.");
        }
    }
}
```

The full name of the target assembly is stored in the constant `Wp8AssemblyFullName`, and its value is **Microsoft.Phone, Version=8.0.0.0, Culture=neutral, PublicKeyToken=24eec0d8c86cda1e**. We will only load it if we are running under Windows Phone 8, thus circumventing the prohibition of referencing an assembly of a superior .NET version in case we did it via the **Add Reference...** command in the editor.

Now, we only need two helper functions to instantiate types stored in the loaded assembly:

```
public Type GetAssemblyType(string fullName)
{
    return wp8Assembly.GetType(fullName);
}

public object CreateTypeInstance(Type type)
{
    return wp8Assembly.CreateInstance(type.FullName);
}
```

These will come in handy when we modify the code behind of the page.

Modifying the page's XAML

As stated before, create one TextBox and four buttons in the page, either using Blend or the editor integrated in Visual Studio. If you used MVVM Light and modified the default `MainViewModel` class as stated, your viewmodel should be already registered in the `ViewModelLocator` class, allowing us to databind to it as if it were a singleton instance. So, set the `Text` property's value of your TextBox to `{Binding Main.OsVersion}` to display the operating system when running the application.

Next up, create four buttons and add different functions to their `Click` events. To prevent the user for launching Windows Phone 8 specific tasks under older system versions, set the `IsEnabled` property of the corresponding buttons to `{Binding Main.IsWp8, Mode=OneWay}`. It would be a good addition to create a converter that transforms the value of `IsWp8` to a `System.Windows.Visibility` and bind it to the `Visibility` property to efficiently hide unavailable features to the end user.

Instantiating and launching the tasks

The final step is filling in the functions that will launch when each button is pressed. For the first two we are going to launch two tasks that have been available since Windows Phone 7, `EmailComposeTask` and `WebBrowserTask`:

 (C) Copyright Nokia 2014. All rights reserved.

```
private void MailTaskButton_Click(object sender, System.Windows.RoutedEventArgs e)
{
    EmailComposeTask composeTask = new EmailComposeTask();
    composeTask.Show();
}

private void BrowserTaskButton_Click(object sender, System.Windows.RoutedEventArgs e)
{
    WebBrowserTask browserTask = new WebBrowserTask();
    browserTask.Uri = new Uri("http://www.developer.nokia.com/");
    browserTask.Show();
}
```

As you can see, there's nothing exceptional in this code. Now comes the part where we must use Reflection to extract the types needed. We have to follow these steps in each function that needs this:

- First, we must check if `CanLoadTypes` is true, in case anything went wrong when initializing the viewmodel.
- Search for the required `Type` using the `GetAssemblyType` helper method, and specify the full name. For example, for `SaveAppointmentTask` it would be `Microsoft.Phone.Tasks.SaveAppointmentTask`.
- Instantiate it using `CreateTypeInstance`. Store the resulting value in a variable of type `object`.
- Search for the desired member method by calling `GetMethod` in the previosly searched `Type`. Its only parameter is the name of the function, in our case **Show**. Store the resulting `MethodInfo` value.
- In case you need to set the value of any property (like the case of `MapsTask`), call `GetProperty` in the `Type` using the property's name. This will return a `PropertyInfo` variable where you can call `SetValue` to assign its value.
- Call the method `Invoke` of the `MethodInfo` previously obtained.

```
private void SaveAppointmentButton_Click(object sender, System.Windows.RoutedEventArgs
e)
{
    // If we are in WP8
    if (this.viewModel.CanLoadTypes)
    {
        // Instantiate a SaveAppointmentTask and call its Show method.
        Type taskType = this.viewModel.GetAssemblyType(SaveAppointmentTaskTypeName);
        object taskObject = this.viewModel.CreateTypeInstance(taskType);
        MethodInfo showMethodInfo = taskType.GetMethod(ShowMethodName);
        showMethodInfo.Invoke(taskObject, null);
    }
}

private void SearchLocationButton_Click(object sender, System.Windows.RoutedEventArgs e)
{
    // If we are in WP8
    if (this.viewModel.CanLoadTypes)
    {
        // Instantiate a MapsTask, set its SearchTerm property and call its Show method.
        Type taskType = this.viewModel.GetAssemblyType(MapsTaskTypeName);
        object taskObject = this.viewModel.CreateTypeInstance(taskType);
        MethodInfo showMethodInfo = taskType.GetMethod(ShowMethodName);
        PropertyInfo termPropertyInfo = taskType.GetProperty(SearchTermPropertyName);
        termPropertyInfo.SetValue(taskObject, SearchTermValue, null);
        showMethodInfo.Invoke(taskObject, null);
    }
}
```

Now you can compile your project both from Visual Studio 2010 and Visual Studio 2012 and deploy it to Windows Phone 7 and Windows Phone 8 emulators/devices to test that it's running correctly.

(C) Copyright Nokia 2014. All rights reserved.

Limitations

We can only access new functions existing in the Microsoft.Phone assembly. Since the new Windows namespace is inside a WinRT component and these can't be reflected at the time being, accessing features like in-app purchases is impossible. If a method for doing this is discovered in the future, this wiki article will be accordingly updated.

Likewise, features that require new capabilities not available in Windows Phone 7 (like Wallet and removable storage) can't be used because the compiler will tell us that there is an unrecognised option specified in the application's manifest.

Summary

Maintaining a project for two different platforms will always be a tough task, but by using all those little tricks you should be able to make things easier.

Don’t forget to thoroughly test both the WP7 and WP8 version of your apps, as there may be slight behaviour changes on some common APIs. When you’re confident enough about the quality of your application, the last step is to publish your **.xap** file. The Windows Phone Dev Center allows you to independently update the WP7 version of your app even after you've published a WP8 version, so you always have the opportunity to publish fixes or whole new features for your legacy user-base.

Source Code

- You can find the source code of WP8 features in WP7 app on below link.
  File:WP8inWP7.zip

 (C) Copyright Nokia 2014. All rights reserved.