Building a Universal Windows Platform (UWP) Application (Part 2) – T4 and Strings

by **admin** | Oct 13, 2016

I just rolled off a client recently that needed to build a Universal Windows Platform (UWP) application as part of their hardware and software solution. For those who aren't yet familiar with UWP, you can check out this article by Tyler Whitney.

As many of you developers out there are aware, sometimes you have to build or bring with you a number of application infrastructure items before you can even get started with the core application logic. For example, you might need some helpers, services and base classes that make your job easier or allow you to start with your base patterns, such as, MVC, MVVM, etc.

So, I have decided to create a series of posts that build up a bunch of common, important parts of an application that you might want to have in place before you even start developing your core functionality. Check out the previous post about Using Template10. If you haven't already read the previous post, I recommend you do since they all build on each other.

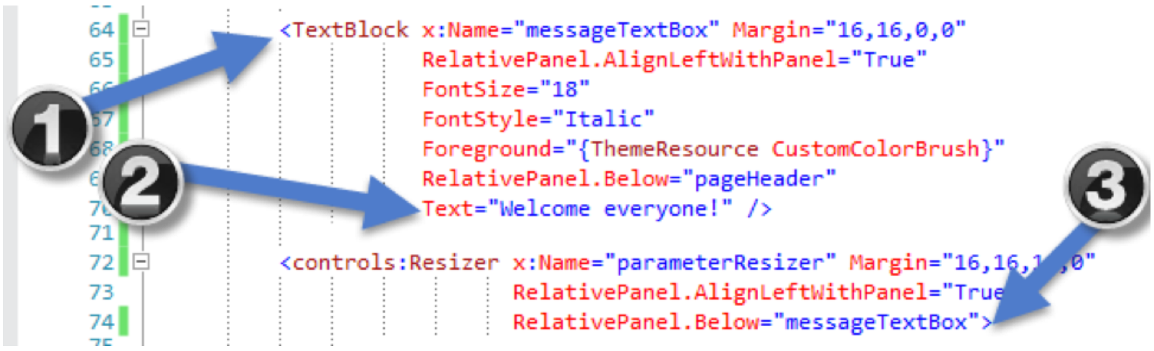
Today, we are going to discuss how we use strings in our application. Sounds easy doesn't it? I mean, just type in your text like this, "Hello World!", right? Well, not exactly.

What we really want to accomplish are a number of things:

- We want to support multiple languages
- We want to have Intellisense available to prevent runtime errors
- We want to be able to create and manage translations easily
- We want it all to be simple (can someone else do the work for us?) Those points being mentioned, let's start with the problem and move on to fix it.

How to Add a Message

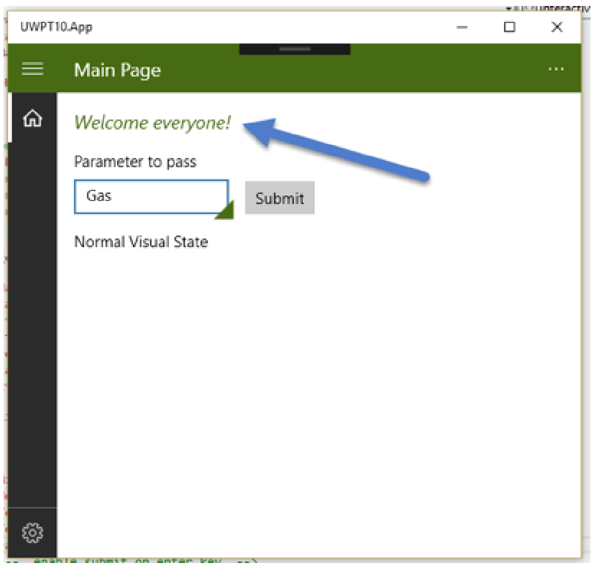
First, let's **open MainPage.xaml** and add a welcome message. As you can see below, I **added** a **TextBlock** that displays the message **"Welcome everyone!**".



```
<TextBlock x:Name="messageTextBox" Margin="16,16,0,0"
           RelativePanel.AlignLeftWithPanel="True"
           FontSize="18"
           FontStyle="Italic"
           Foreground="{ThemeResource CustomColorBrush}"
           RelativePanel.Below="pageHeader"
           Text="Welcome everyone!" />

<controls:Resizer x:Name="parameterResizer" Margin="16,16,1[illegible],0"
                  RelativePanel.AlignLeftWithPanel="True
                  RelativePanel.Below="messageTextBox">
```

Also, notice that I placed it **below** the **PageHeader** and changed the **Resizer** control **position** to be **below** our new **TextBlock**.

If I run the application now. We see our message.

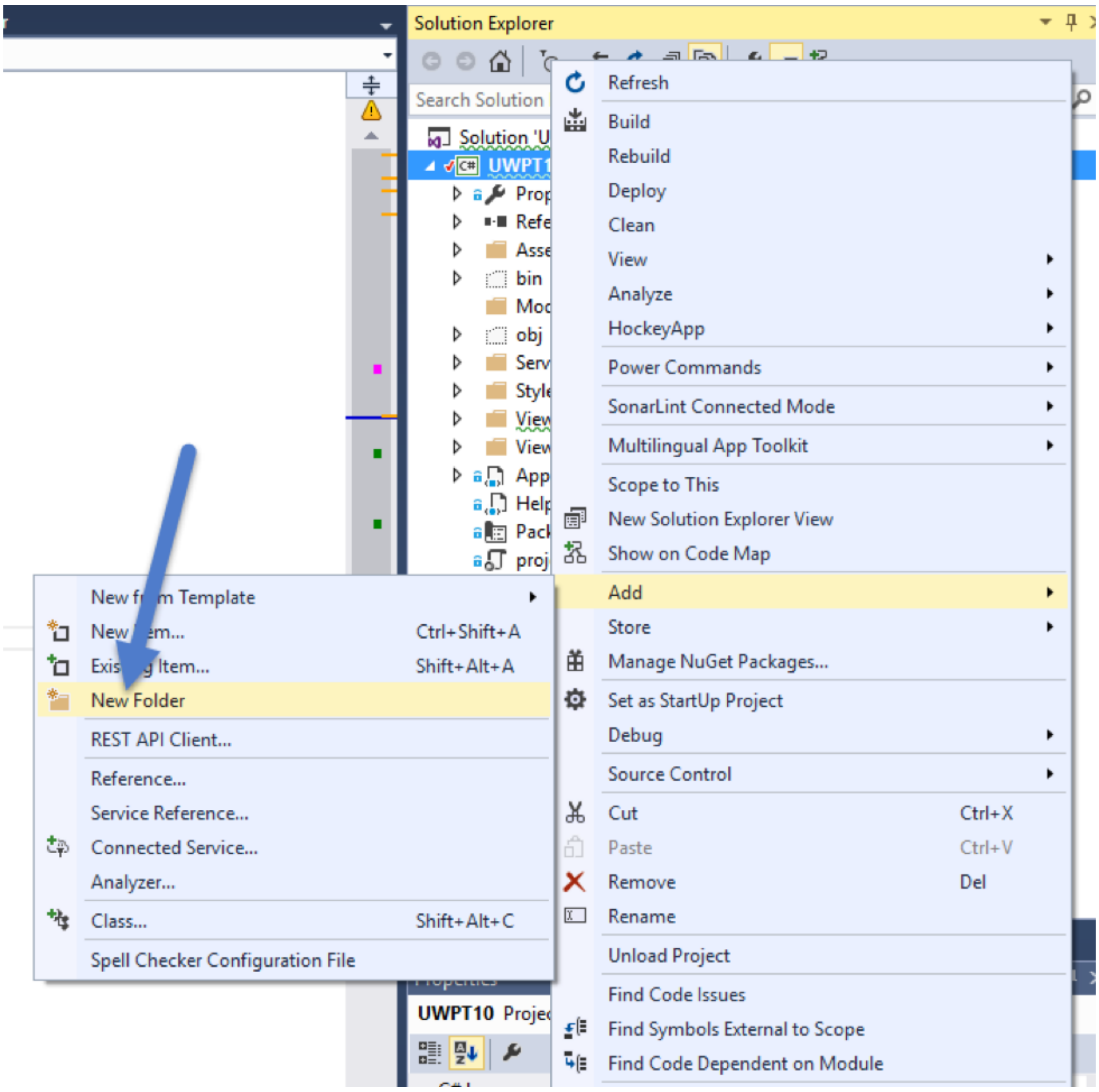


Okay, so, that was easy right? Well, what if we want someone to see the message in Spanish? What would we have to do. Well, we would have to change the message to Spanish text, rebuild the application and then send it to them. Acceptable? Uh, no.

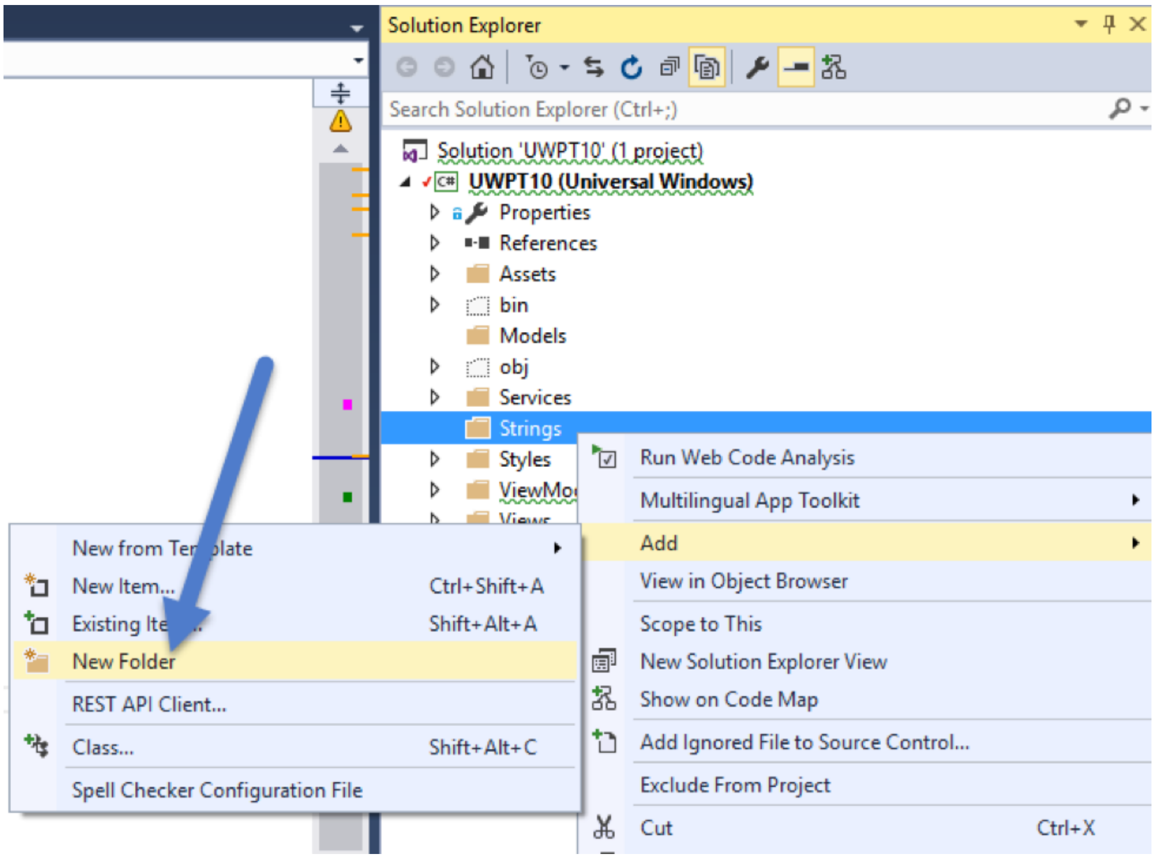
This issue has been around for a long time and one of the best ways to fix this is by using string resource files. This way, all we need are resource files for each language we wish to support.

The Easiest Way to Create a Default Language

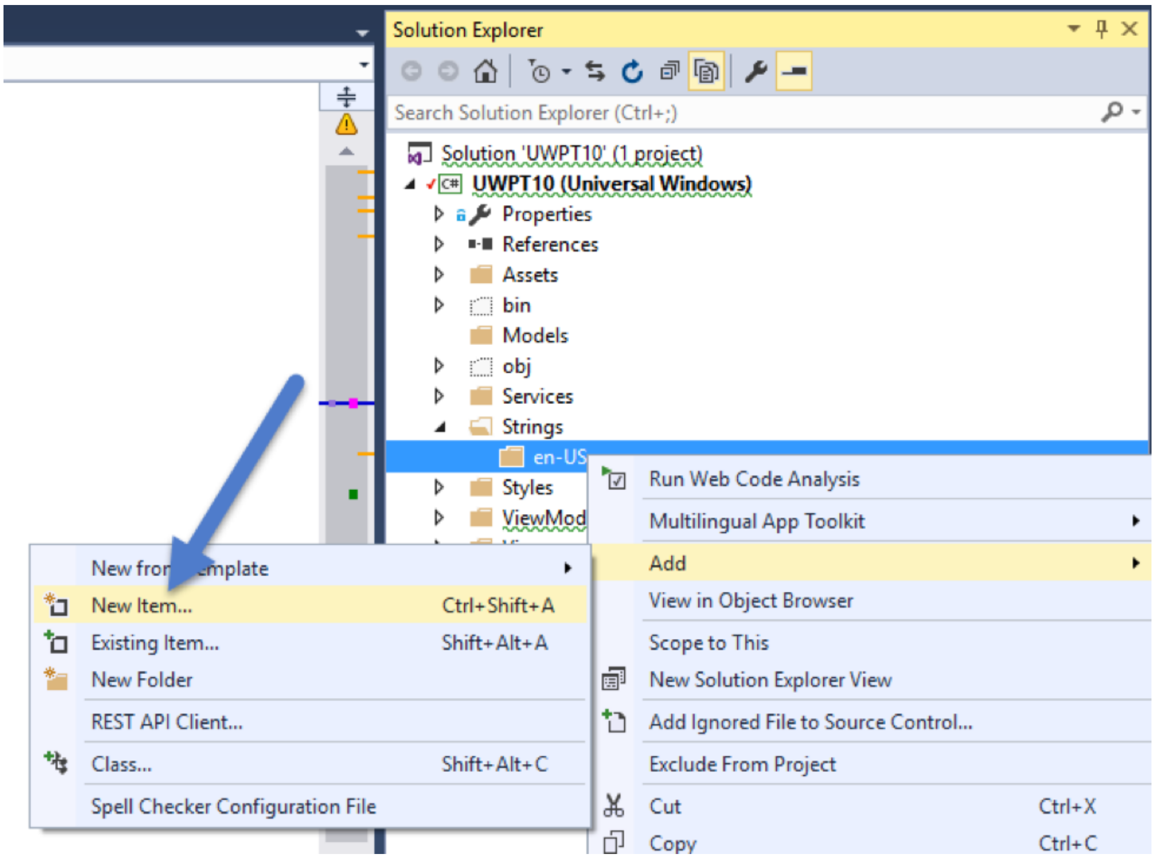
First we will need to add our default language. To do this, start by creating a Strings folder in the root of our application structure. **Right click** on your **project**, **select Add** then **select New Folder**. Name the folder **Strings**.



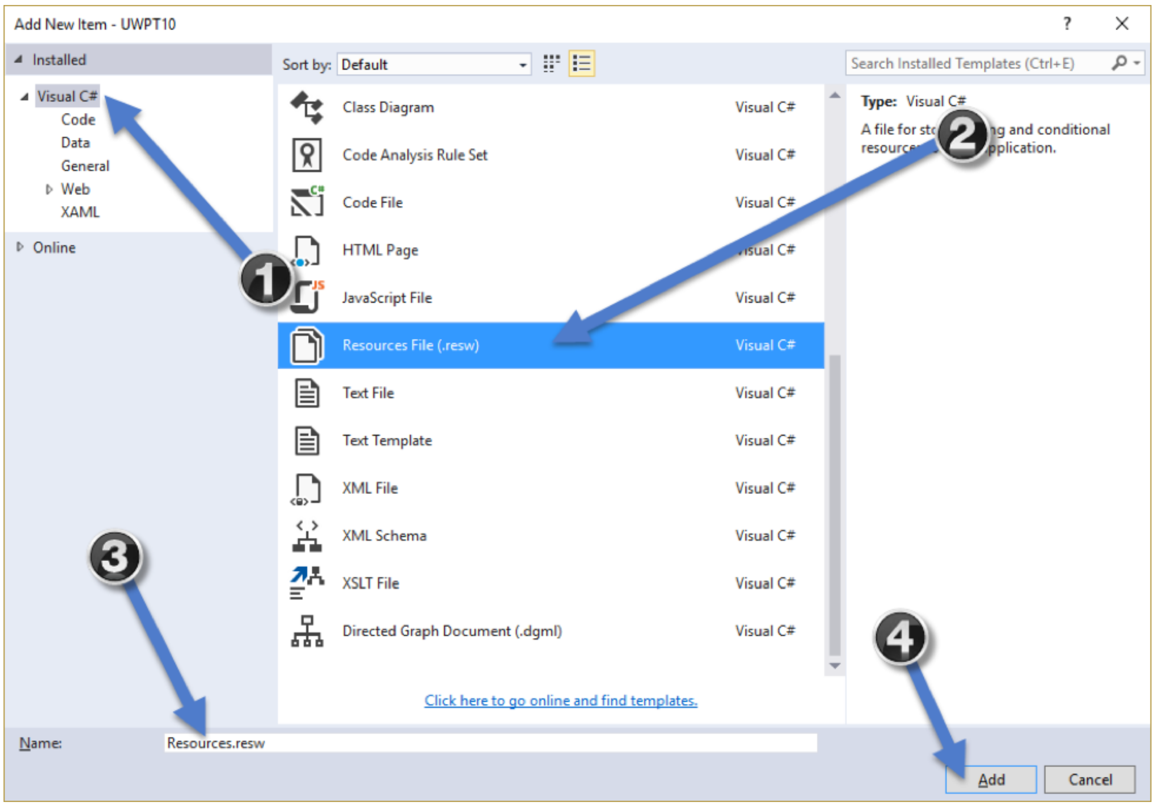
Now, from here, let's create our default language folder. In my case, I am going to make the default English. So, **right click** on the **Strings** folder, **select Add** then **select New Folder**. **Name** the folder **en-US**.



Next, we add a resource file. To do this, we **right click** on **en-US** folder, **select Add** and then **select New Item…**

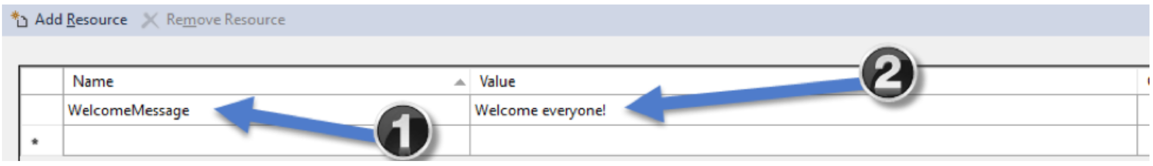


At this point you will be prompted to add a new item. From here, **select** the **Installed tab**, **select Visual C#**, **select Resources File (.resw)** and **keep** the default name **Resoures.resw**. **Select Add**.



Now, let's add a new string and some code to show the string on the home page.

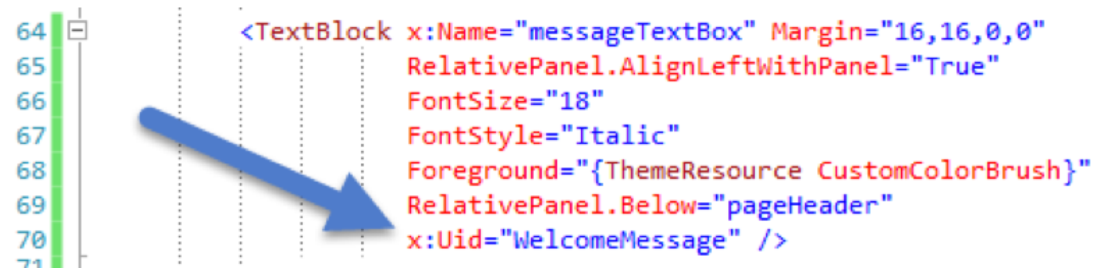
First, **open** the resources file, **Resource.resw** in the **en-US folder** and **add** a new string **WelcomeMessage**, with a value of "**Welcome everyone!".**



Name	Value
WelcomeMessage	Welcome everyone!

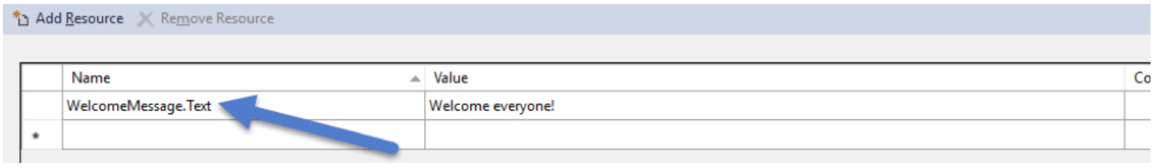
There are a few ways to add the string to your code in order to show it in your UI. I am going to choose to update my view via my view model, using a resource string so that I have the flexibility to change it on the fly if necessary. Let me explain.

So, there are two basic ways to access the strings. The first is to access the field directly in your XAML. **Open** up **MainPage.xaml**. **Add x:Uid="WelcomeMessage"** and **remove Text="Welcome everyone!"** to your **TextBlock**.



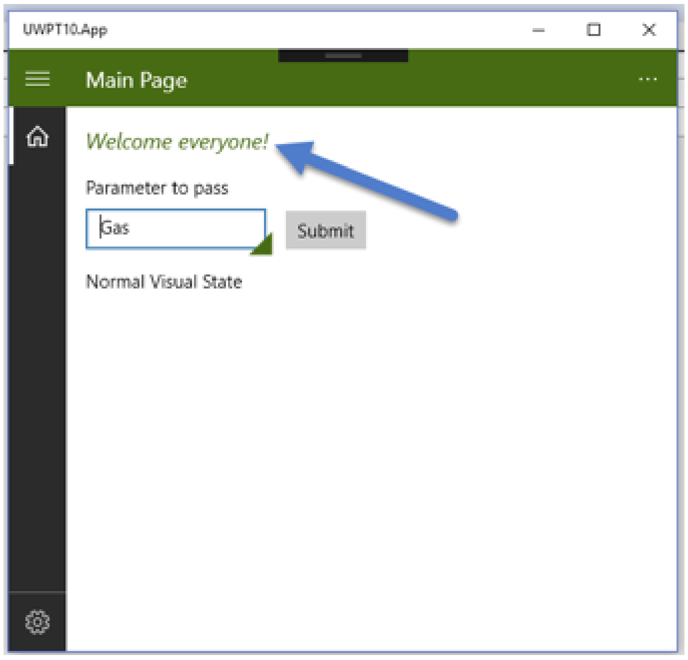
```
<TextBlock x:Name="messageTextBox" Margin="16,16,0,0"
           RelativePanel.AlignLeftWithPanel="True"
           FontSize="18"
           FontStyle="Italic"
           Foreground="{ThemeResource CustomColorBrush}"
           RelativePanel.Below="pageHeader"
           x:Uid="WelcomeMessage" />
```

Now, run your application. Did you see the welcome text? No, are you sure? Well, you are seeing it correctly, the message isn't displayed. Why? Well, there is one thing you have to do in order to get the string to display. In our strings resource file, you need to add the property to the end of the string name in order for the Uid to be used within the TextBlock.



	Name	Value
	WelcomeMessage.Text	Welcome everyone!
*		

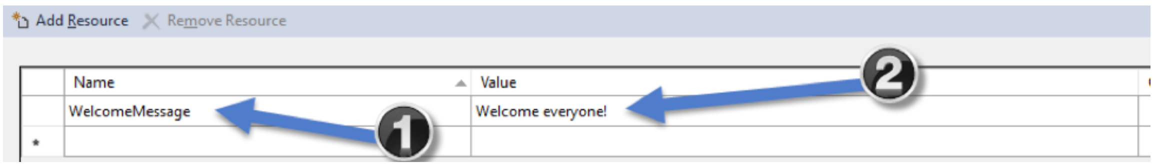
Now run it. What do you see?



Perfect, it shows up now. Okay, so this leads me to an issue I have with this approach. What if you have a text field that changes based on context or content? So, instead of "Welcome everyone!", you want it to say "Welcome John!", if John is the only one using the application? Besides that, the idea of adding another string literal id to my design. Seems like a simple place to type something wrong.

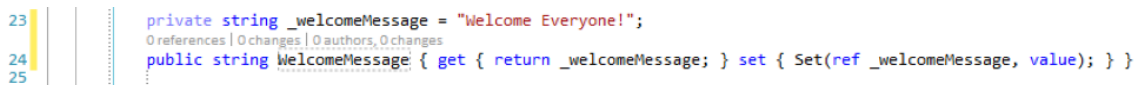
That leads me to another approach, what if we were able to bind the text via a property in the view model. That way, we can set it based on certain criteria?

First, let's change our string Name in Resources.resw back to WelcomeMessage.



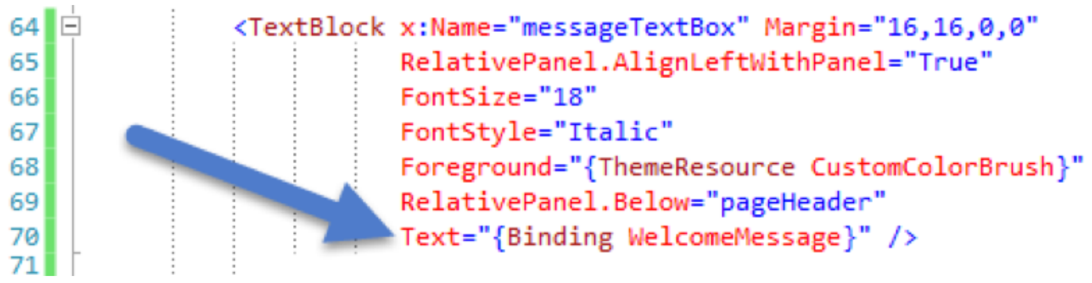
	Name	Value
	WelcomeMessage	Welcome everyone!
*		

Let's **open** up **MainPageViewModel.cs** and add a new property that will hold our message, called, well, yeah, **WelcomeMessage**.



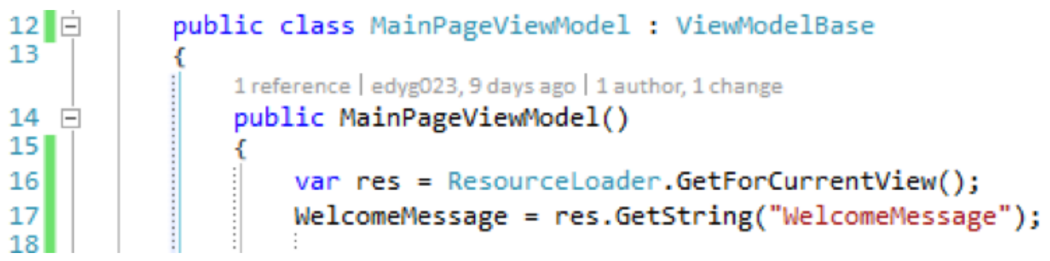
```
private string _welcomeMessage = "Welcome Everyone!";
0 references | 0 changes | 0 authors, 0 changes
public string WelcomeMessage { get { return _welcomeMessage; } set { Set(ref _welcomeMessage, value); } }
```

Now, in our view, **bind** our **WelcomeMessage** property to the **Text** property.



```
<TextBlock x:Name="messageTextBox" Margin="16,16,0,0"
           RelativePanel.AlignLeftWithPanel="True"
           FontSize="18"
           FontStyle="Italic"
           Foreground="{ThemeResource CustomColorBrush}"
           RelativePanel.Below="pageHeader"
           Text="{Binding WelcomeMessage}" />
```

For this example, let's **set** the **WelcomeMessage** in the **constructor** with the string resource. To do this, we need to get the current views resource loader. This is what does all the resource language magic for us. After we get the resource loader, we can make a call to the GetString() function with our string resource name, WelcomeMessage.



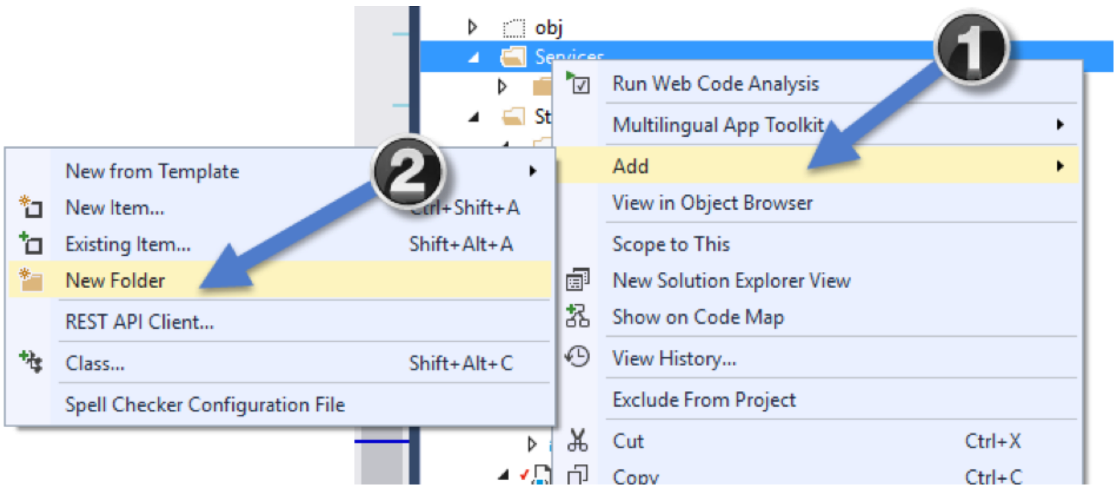
```
public class MainPageViewModel : ViewModelBase
{
    1 reference | edyg023, 9 days ago | 1 author, 1 change
    public MainPageViewModel()
    {
        var res = ResourceLoader.GetForCurrentView();
        WelcomeMessage = res.GetString("WelcomeMessage");
```

Run it. Pretty cool huh? But wait! What happens if you spell the resource name wrong? Truth is, we are kind of back where we were earlier. If you spell something wrong, we don't know until runtime and who knows if we ever actually navigate to a screen that displays the text? What do we do now?

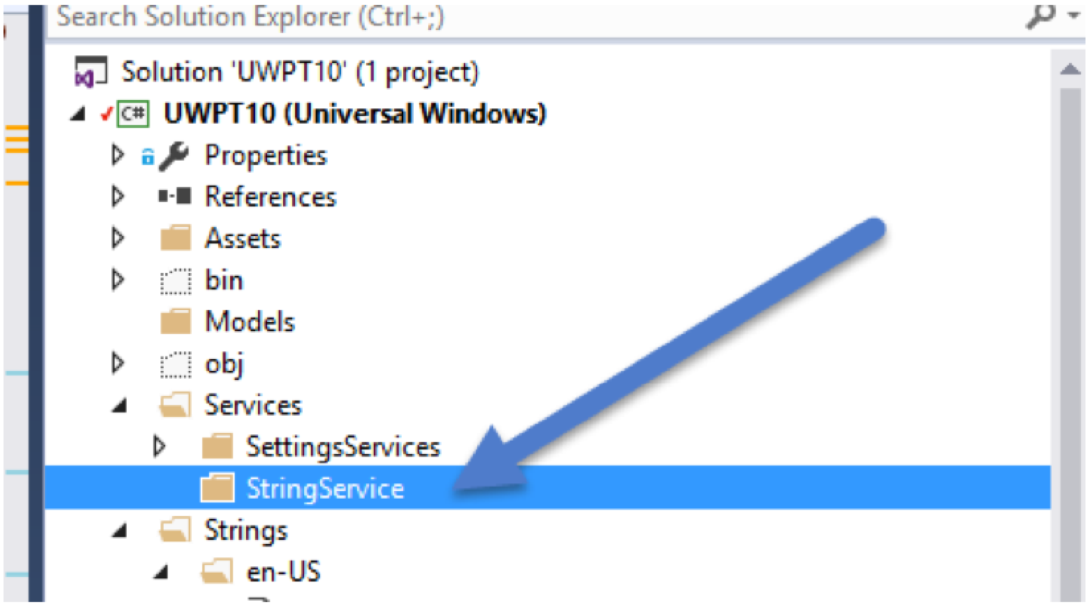
Replicate the Resource Loader Code by Creating a String Service

Well, first, I want to create a string service that I can call from anywhere in my code that replicates the resource loader code. That way I don't have to so all that messy work.

Let's add a new folder called *StringService*. **Right click** on the **Services** folder, **select Add** and then **select New Folder**.

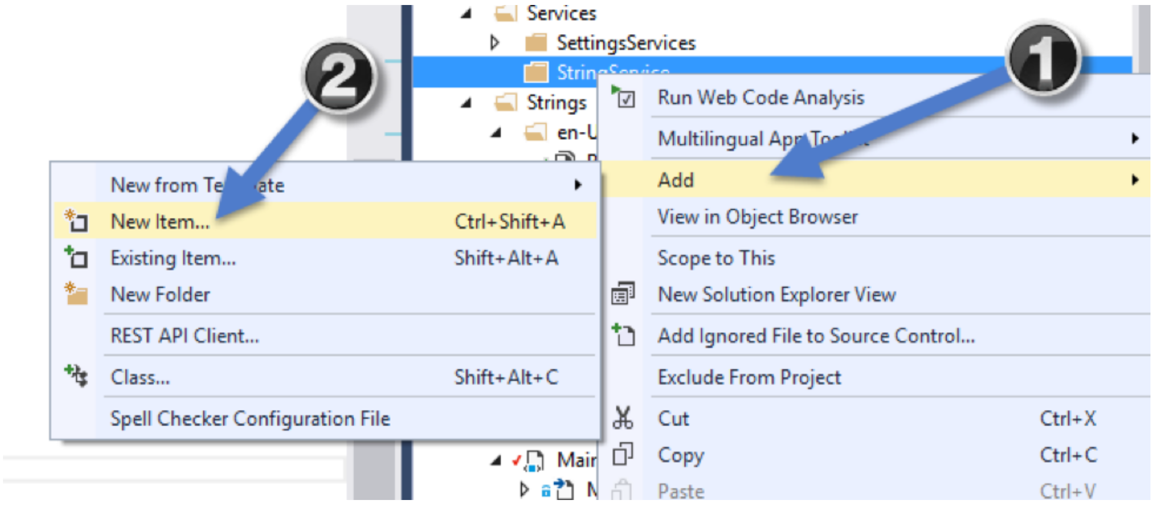


Name the folder **StringService**.

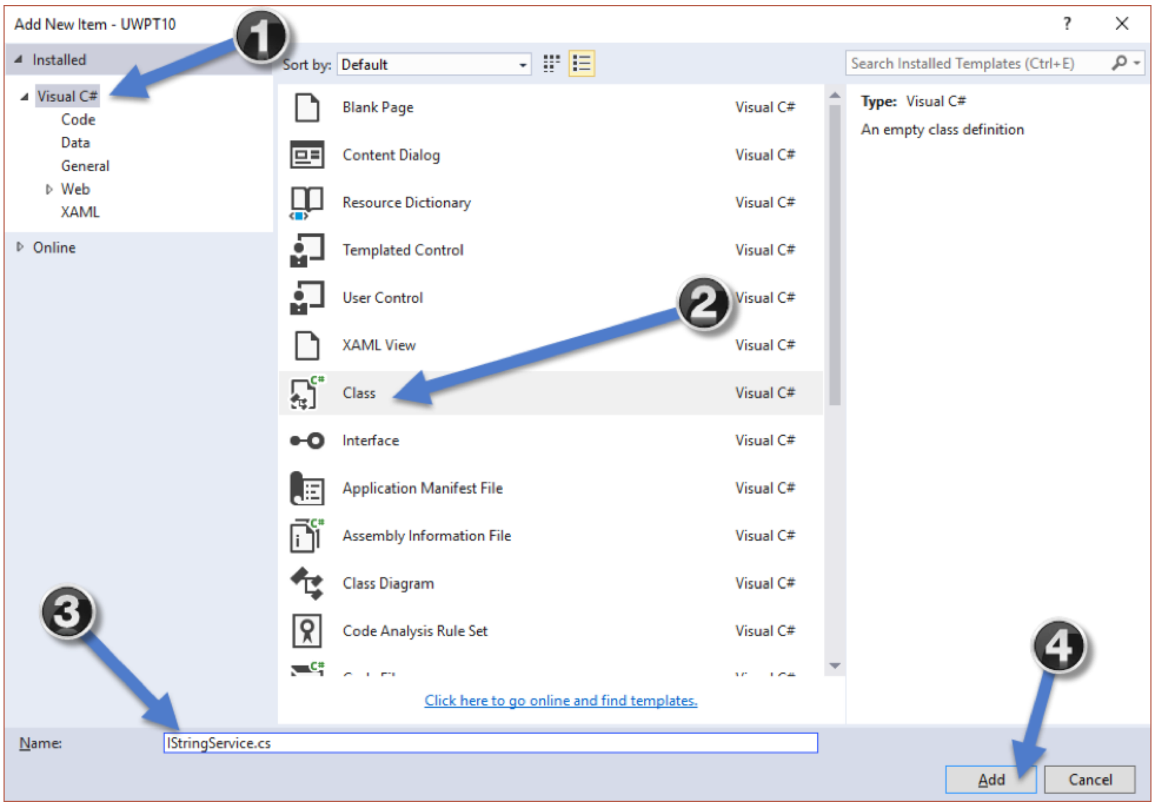

Search Solution Explorer (Ctrl+;)
Solution 'UWPT10' (1 project)
UWPT10 (Universal Windows)
Properties
References
Assets
bin
Models
obj
Services
SettingsServices
StringService
Strings
en-US

Now, let's add a couple of files to this folder. One is our interface, *IStringService.cs* and the other is our service code, *StringService.cs*.

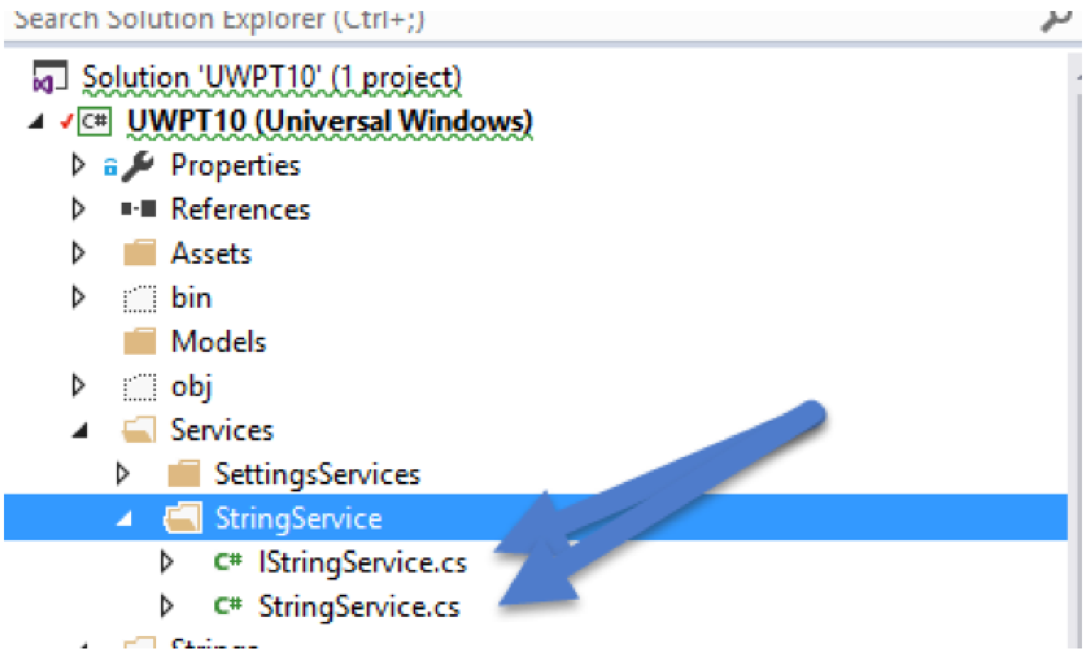
Right click on our StringService folder, select Add and then select Add Item…



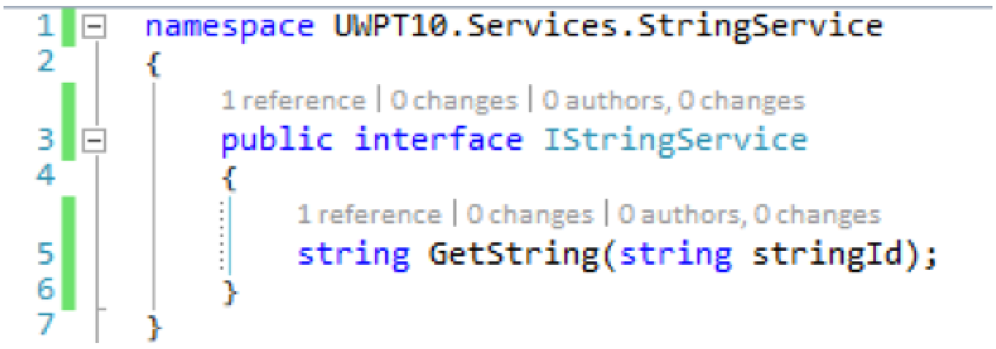
Select Visual C# section, select Class, enter IStringService.cs in the Name field and then select Add.



Do this again for StringService.cs. Once you have completed this, you should have both files under the StringService directory.

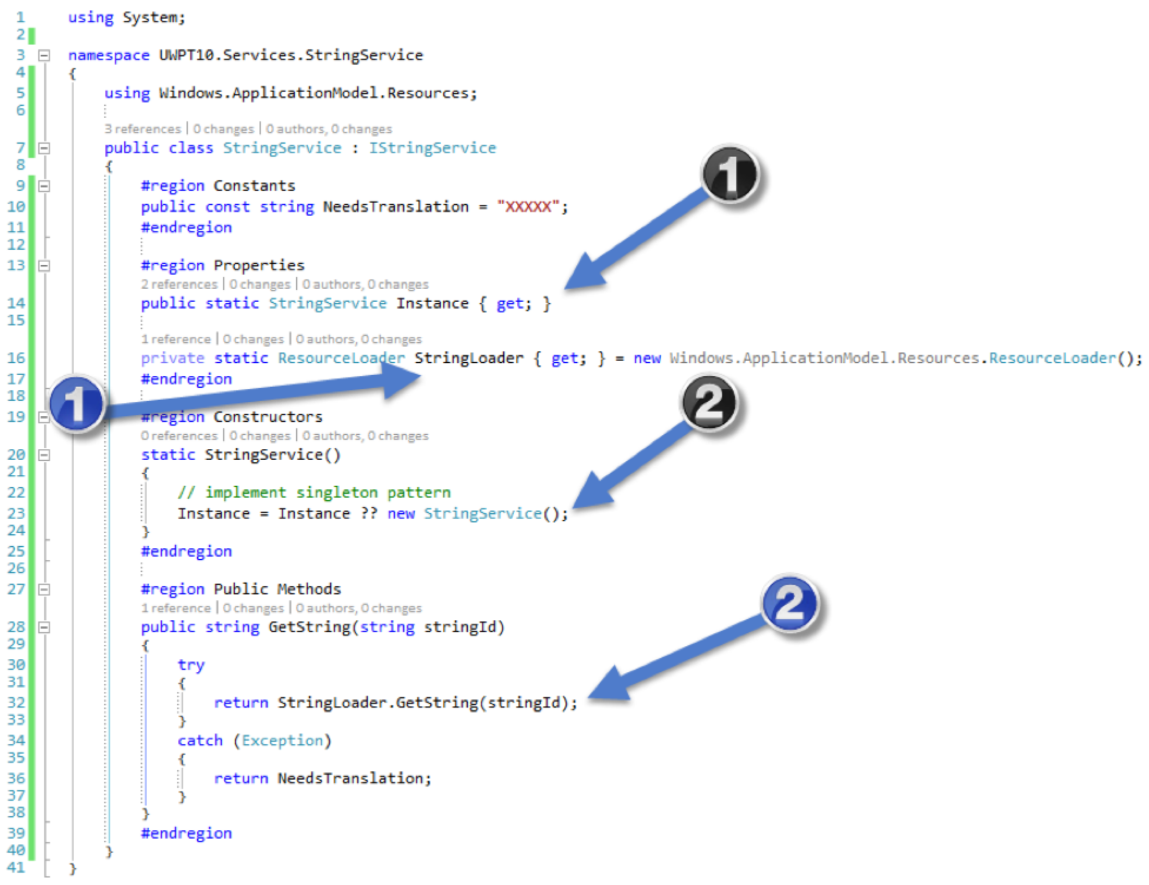


For *IStringService.cs*, we are going change the class to an interface and have one method signature for now, **GetString(string stringId)**;.



```
namespace UWPT10.Services.StringService
{
    1 reference | 0 changes | 0 authors, 0 changes
    public interface IStringService
    {
        1 reference | 0 changes | 0 authors, 0 changes
        string GetString(string stringId);
    }
}
```

For *StringService.cs*, we are going to change the class quite a bit. I will summarize and show you the code. First, we are going to change the class to use a singleton pattern. Then, we are going to implement the *IStringService* interface. See the code below:



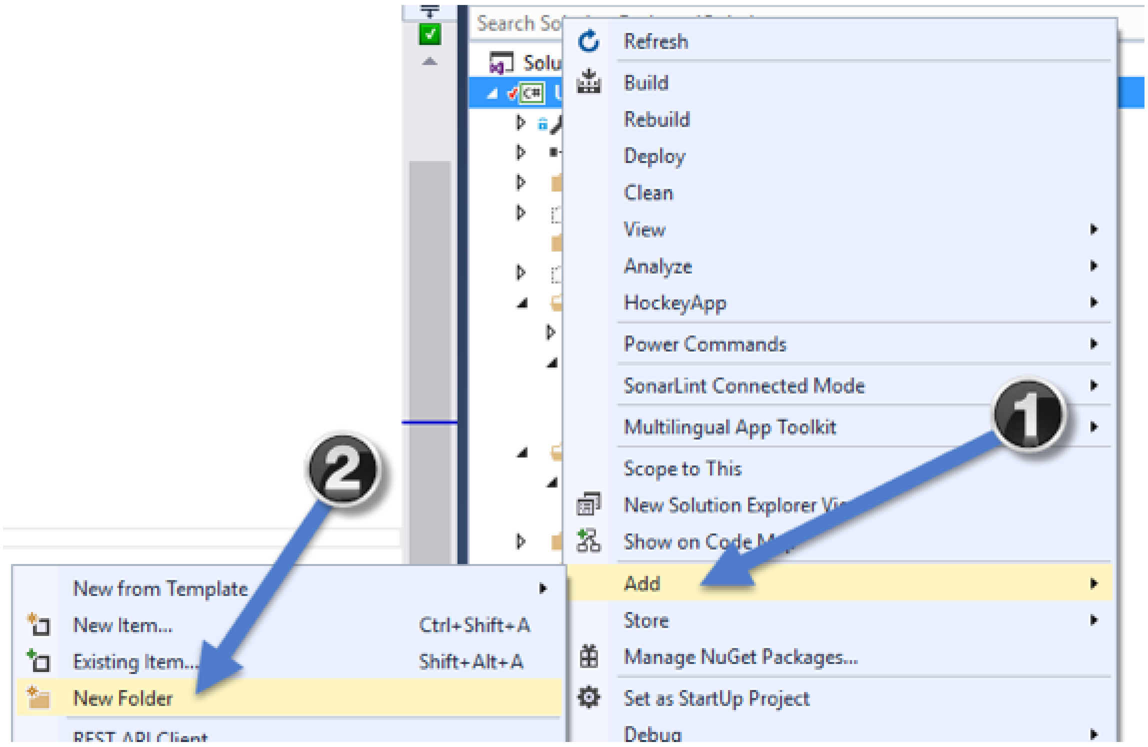
```
using System;

namespace UWPT10.Services.StringService
{
    using Windows.ApplicationModel.Resources;

    3 references | 0 changes | 0 authors, 0 changes
    public class StringService : IStringService
    {
        #region Constants
        public const string NeedsTranslation = "XXXXX";
        #endregion

        #region Properties
        2 references | 0 changes | 0 authors, 0 changes
        public static StringService Instance { get; }

        1 reference | 0 changes | 0 authors, 0 changes
        private static ResourceLoader StringLoader { get; } = new Windows.ApplicationModel.Resources.ResourceLoader();
        #endregion

        #region Constructors
        0 references | 0 changes | 0 authors, 0 changes
        static StringService()
        {
            // implement singleton pattern
            Instance = Instance ?? new StringService();
        }
        #endregion

        #region Public Methods
        1 reference | 0 changes | 0 authors, 0 changes
        public string GetString(string stringId)
        {
            try
            {
                return StringLoader.GetString(stringId);
            }
            catch (Exception)
            {
                return NeedsTranslation;
            }
        }
        #endregion
    }
}
```

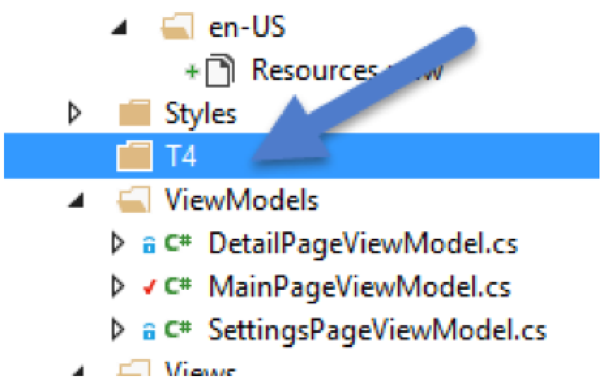
Auto Generate Code With T4 Text Template

Well, we could try to create a class or service that creates the strings, but then, that is a lot more code slinging and again, opens us up to messing up the resource name.

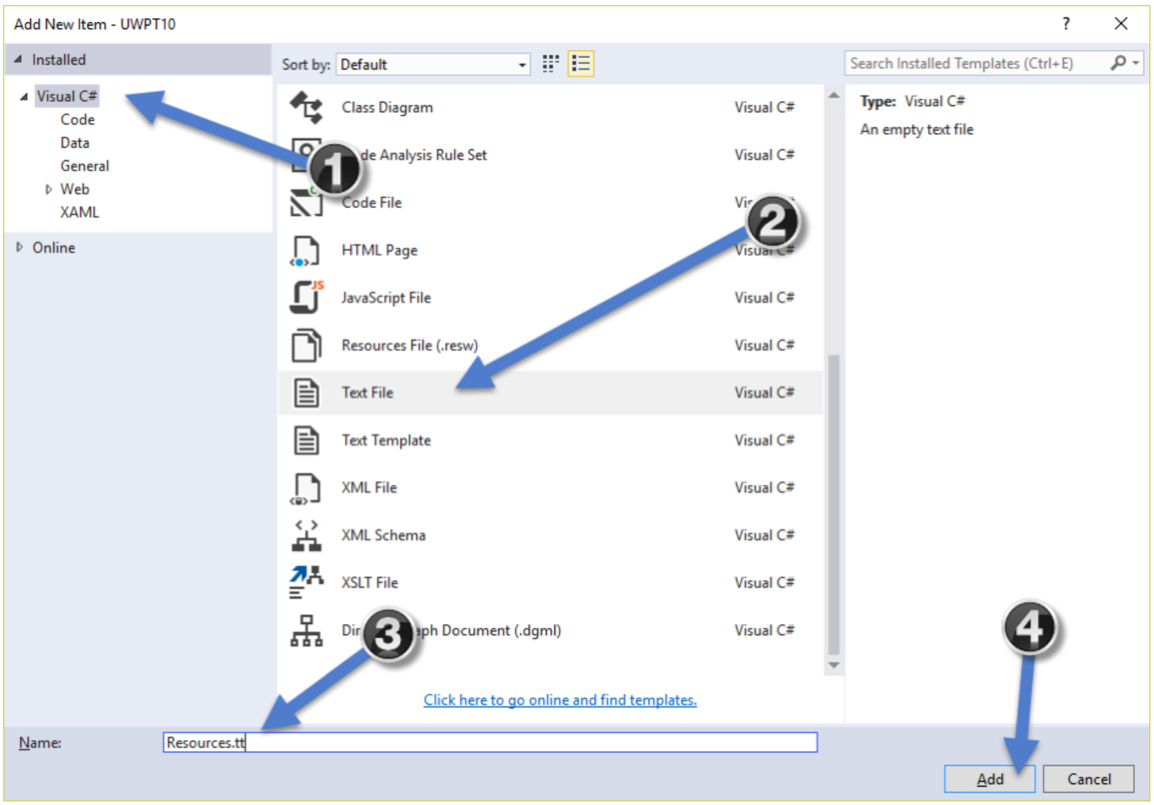
Enter, T4 Text Templates. If you haven't heard of it before, T4 Text Templates allow you to auto generate code, as well as other files in Visual Studio. I don't want to spend too much time on this topic as it could be a post all by itself. However, checkout the this link to read more. It is pretty cool stuff.

For now, I will show you what I did to generate the code necessary to access the strings. First, we need to **make** a new **directory** called, **T4**.

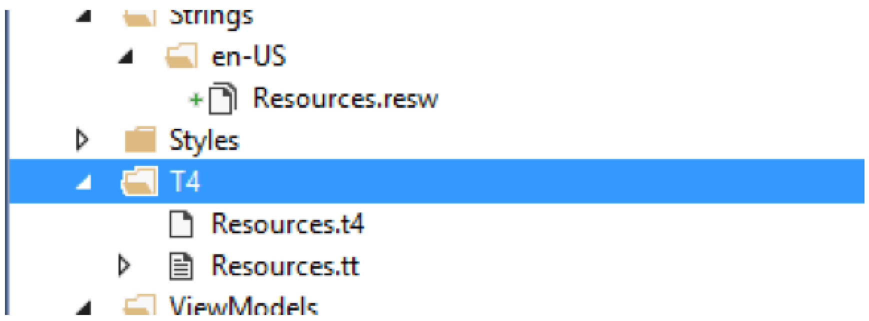




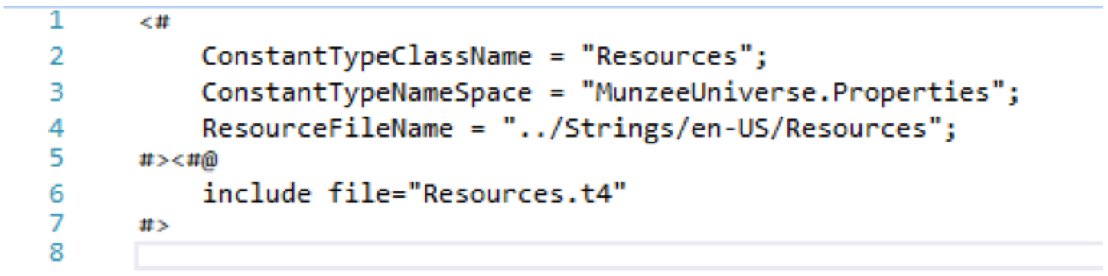
Now, let's **add** two **text files** to the T4 directory named: **Resources.t4** and **Resources.tt**. Do the following once for each file.



It should look like this when you are done.



Now **open Resources.tt** and let's **add** the following:

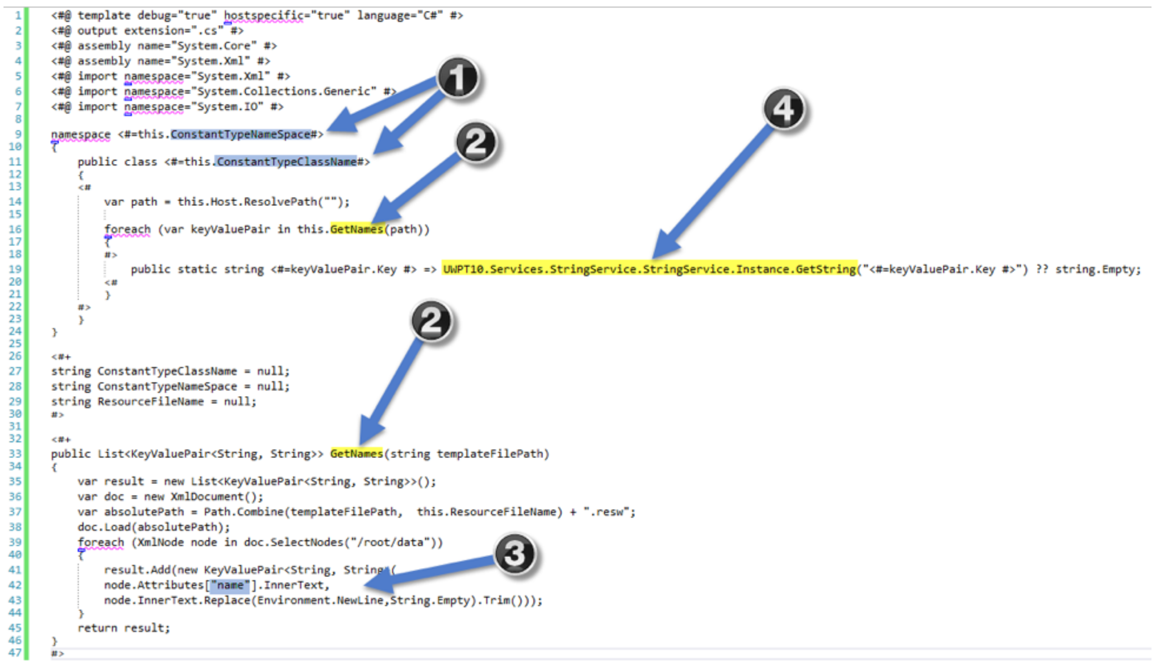


```
<#
    ConstantTypeClassName = "Resources";
    ConstantTypeNameSpace = "MunzeeUniverse.Properties";
    ResourceFileName = "../Strings/en-US/Resources";
#><#@
    include file="Resources.t4"
#>
```

Basically, this is the text template file that defines some constants and a code file, Resources.t4, to include.

1. *ConstantTypeClassName*. Name of the class.
2. *ConstantTypeNameSpace*. Namespace the class resides.
3. *ResourceFileName*. This is path to the resource file that has our resource strings.
4. *t4*. This file contains the code template that will be generated.

Now **open Resources.t4** and let's add the following:



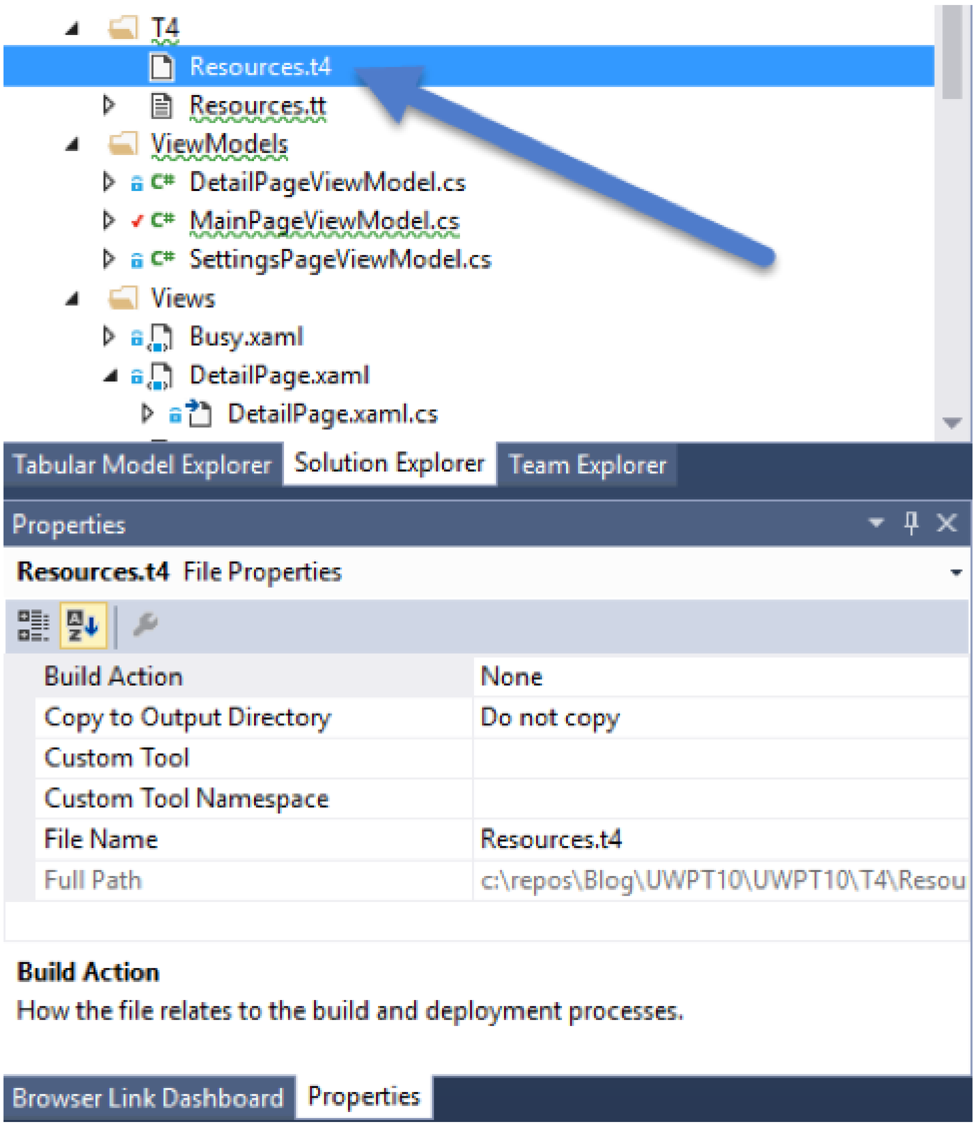
```
<#@ template debug="true" hostspecific="true" language="C#" #>
<#@ output extension=".cs" #>
<#@ assembly name="System.Core" #>
<#@ assembly name="System.Xml" #>
<#@ import namespace="System.Xml" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ import namespace="System.IO" #>

namespace <#=this.ConstantTypeNameSpace#>
{
    public class <#=this.ConstantTypeClassName#>
    {
    <#
        var path = this.Host.ResolvePath("");

        foreach (var keyValuePair in this.GetNames(path))
        {
        #>
            public static string <#=keyValuePair.Key #> => UWPT10.Services.StringService.StringService.Instance.GetString("<#=keyValuePair.Key #>") ?? string.Empty;
        <#
        }
    #>
    }
}

<#+
string ConstantTypeClassName = null;
string ConstantTypeNameSpace = null;
string ResourceFileName = null;
#>

<#+
public List<KeyValuePair<String, String>> GetNames(string templateFilePath)
{
    var result = new List<KeyValuePair<String, String>>();
    var doc = new XmlDocument();
    var absolutePath = Path.Combine(templateFilePath,  this.ResourceFileName) + ".resw";
    doc.Load(absolutePath);
    foreach (XmlNode node in doc.SelectNodes("/root/data"))
    {
        result.Add(new KeyValuePair<String, String>(
        node.Attributes["name"].InnerText,
        node.InnerText.Replace(Environment.NewLine,String.Empty).Trim()));
    }
    return result;
}
#>
```

The Resouces.tt file is the source code template that will get generated. Again, if you want to learn more about the syntax, hit up the link I provided earlier in this article. But for now, I will describe the important parts of this file.

1. Notice that we are using the constants defined in the Resources.t4 file to define the namespace and class name.
2. Notice that the code will attempt to look thru all of our resource string names by calling the method GetNames().
3. GetNames() returns a list of KeyValue(string,string) entries for each string resource key, "name" and it's value.
4. Now the cool part, notice that we create, for each resource KeyValue pair, a property with the name of the resource key that initializes the value with a call to our StringService with the KeyValue key as it's parameter.

This will make more sense in a bit. Now, before we apply the T4 text template and generate our code, let's make sure that the properties are set for each to do it's magic. The default values for each of these files should be the following:

Resources.t4

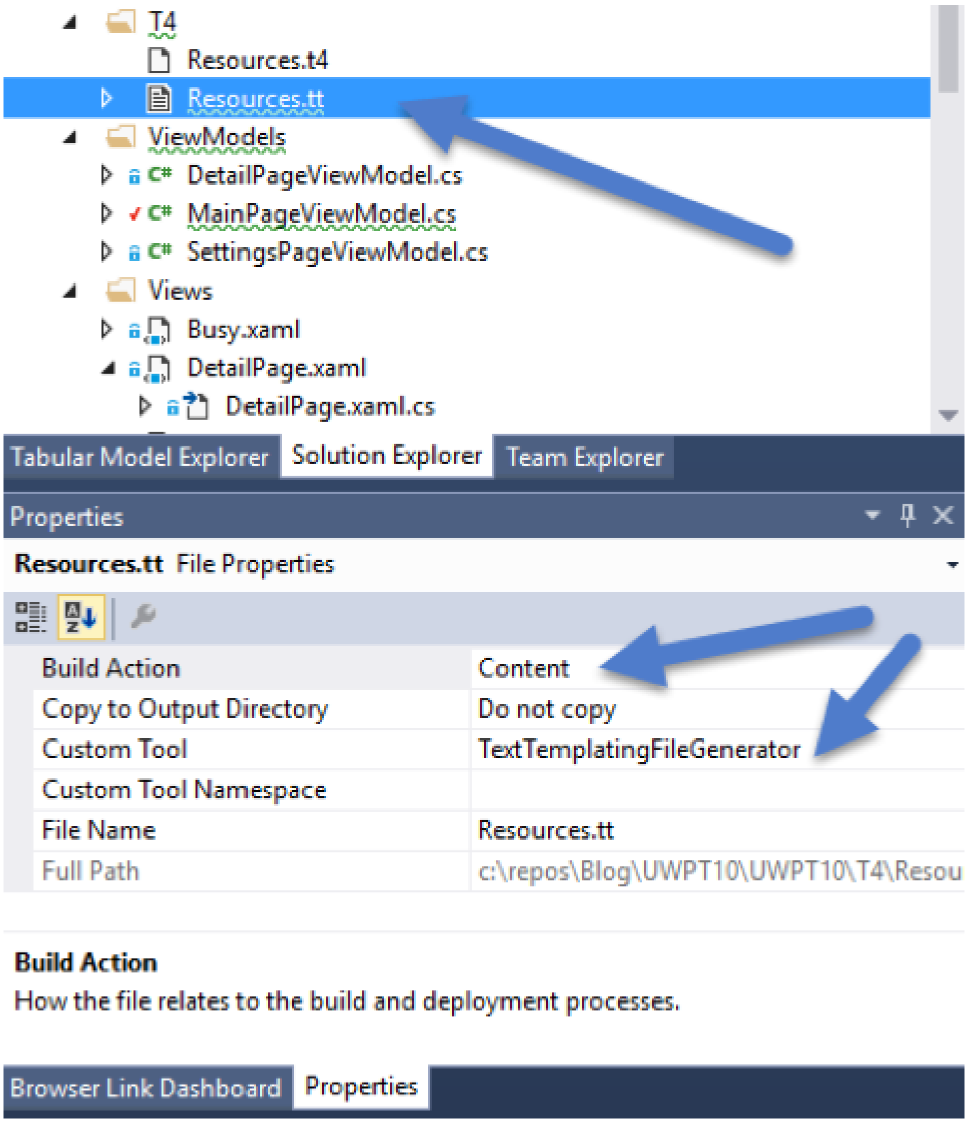


Resources.t4 File Properties

Build Action	None
Copy to Output Directory	Do not copy
Custom Tool	
Custom Tool Namespace	
File Name	Resources.t4
Full Path	c:\repos\Blog\UWPT10\UWPT10\T4\Resou

Build Action
How the file relates to the build and deployment processes.

Resources.tt



Resources.tt File Properties

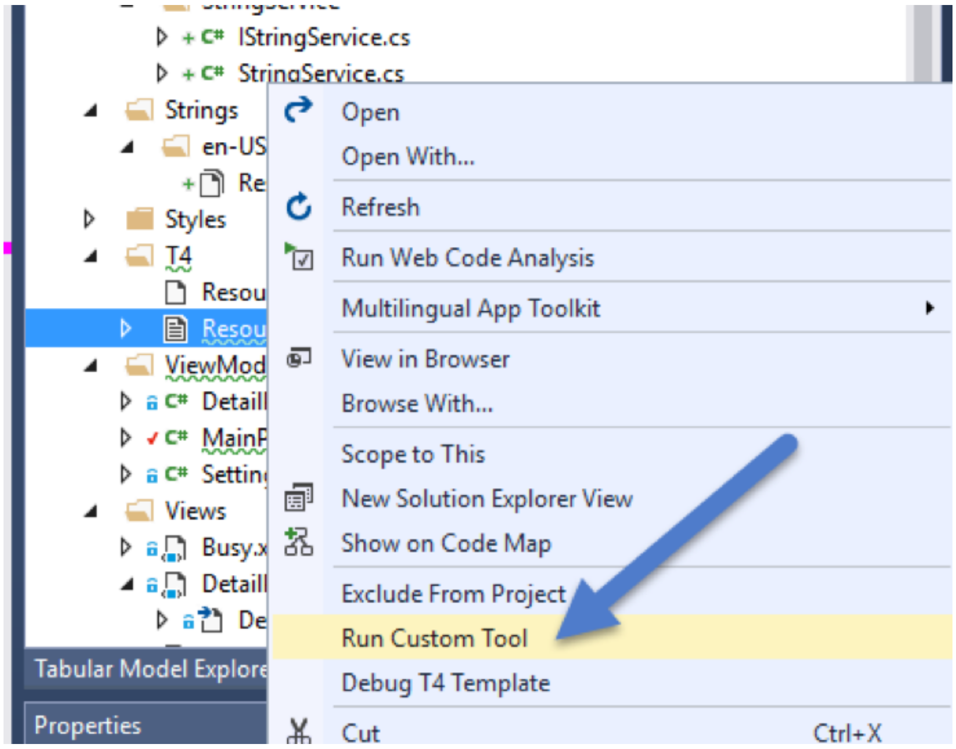
Build Action	Content
Copy to Output Directory	Do not copy
Custom Tool	TextTemplatingFileGenerator
Custom Tool Namespace	
File Name	Resources.tt
Full Path	c:\repos\Blog\UWPT10\UWPT10\T4\Resou

Build Action
How the file relates to the build and deployment processes.

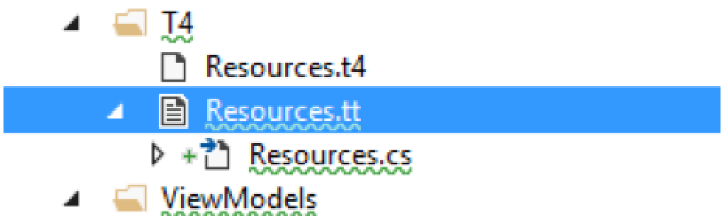
Notice the difference between Resouces.t4 and Resources.tt. Resources.tt, has two differences.

1. **Build Action** is set to **Content**
2. **Custom Tool** is set to **TextTemplatingFileGenerator**. This is the T4 text template file generator, as the name indicates. It creates the code from our template that we are going to use in our code.

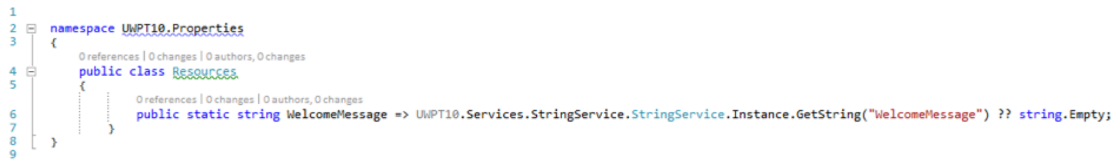
Now, let's watch the magic happen … **right click** on **Resources.tt** and **select Run Custom Tool**.



Unless there is some catastrophic error, you won't see much. However, if you look at your **Resources.tt** file in your tree structure, you will notice that there is a file under it if you expand the tree node called **Resources.cs**.



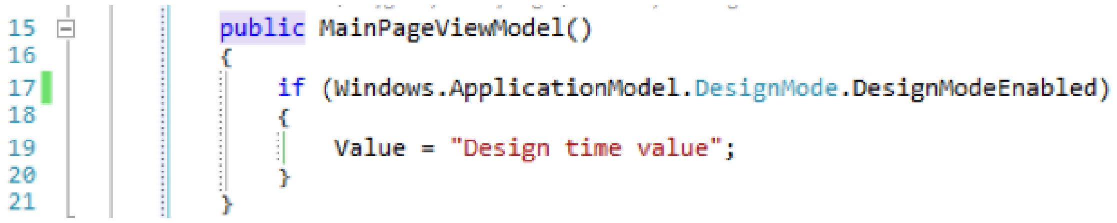
Open up **Resources.cs**.



```
namespace UWPT10.Properties
{
    0 references | 0 changes | 0 authors, 0 changes
    public class Resources
    {
        0 references | 0 changes | 0 authors, 0 changes
        public static string WelcomeMessage => UWPT10.Services.StringService.StringService.Instance.GetString("WelcomeMessage") ?? string.Empty;
    }
}
```

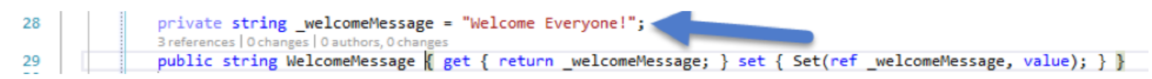
So, what you are looking at is the code that was generated by our T4 Text Template. The key thing to note here is that we now have a class that contains properties for each Resource string that is referenced by the resource key. What is the big deal? Well, two things. First, we didn't have to type in the resource key string avoiding us from making a typing error that won't get caught until runtime. Second, we now get full intellisense when we want to use the string. Let me show you how to do this...**open MainPageViewModel.cs**.

We can now **remove** our **ResouceLoader code** in constructor to make it look like this...



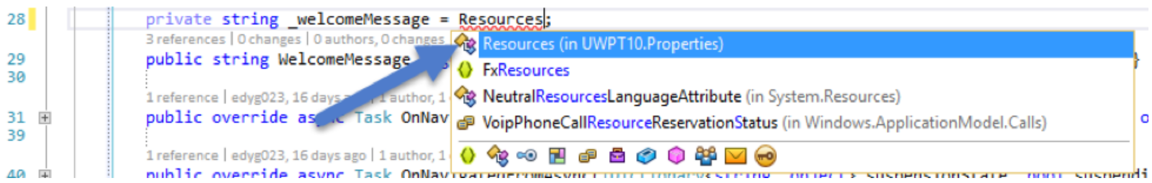
```
public MainPageViewModel()
{
    if (Windows.ApplicationModel.DesignMode.DesignModeEnabled)
    {
        Value = "Design time value";
    }
}
```

Replace our **"Welcome Everyone!"** text…



```
private string _welcomeMessage = "Welcome Everyone!";
3 references | 0 changes | 0 authors, 0 changes
public string WelcomeMessage { get { return _welcomeMessage; } set { Set(ref _welcomeMessage, value); } }
```

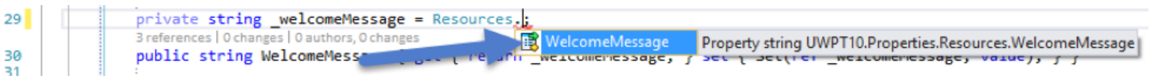
By **typing Resources** and **selecting Resources (in UWPT10.Properties)**



```
private string _welcomeMessage = Resources;
3 references | 0 changes | 0 authors, 0 changes
public string WelcomeMessage
1 reference | edyg023, 16 days ago | 1 author, 1
public override async Task OnNav
1 reference | edyg023, 16 days ago | 1 author, 1
```

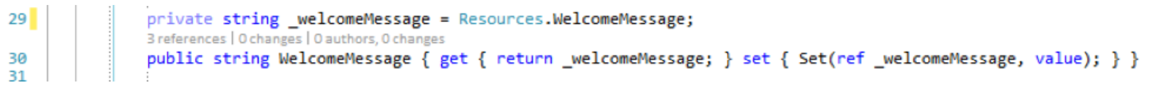
Resources (in UWPT10.Properties)
FxResources
NeutralResourcesLanguageAttribute (in System.Resources)
VoipPhoneCallResourceReservationStatus (in Windows.ApplicationModel.Calls)

Now, **type** the **period** after Resources and you should see our **WelcomeMessage** property. **Select it**.



```
private string _welcomeMessage = Resources.;
3 references | 0 changes | 0 authors, 0 changes
public string WelcomeMess
```

Your code should look like this …



```
private string _welcomeMessage = Resources.WelcomeMessage;
3 references | 0 changes | 0 authors, 0 changes
public string WelcomeMessage { get { return _welcomeMessage; } set { Set(ref _welcomeMessage, value); } }
```

Run it. Pretty cool huh? Now, we never have to worry about typing a string literal in our code for resources. Sweet! No more of those, "Oops, I spelled the resource wrong." errors.

BONUS: There is a nice Visual Studio Extension that will run the T4 Text Template custom tool every time your resource file changes. This is pretty handy as it can be a pain to run the custom tool every time you add a new string to your resource file.

What We Accomplished in Part 2 of Building a Universal Windows Platform Application

- Added a string resource for all of our strings.
- Created a string service for our application to use in the code.
- Used T4 text templating to create a wrapper class of properties that represent our strings so that we don't ever have to type any string literals in our code.

So, now we have a base for using strings in our code. Having this in place early in our project means we don't have to go back and refactor later. As far as strings go, there is still one more thing we will want to do. A lot of applications, in order to have more reach to larger audiences, will need to provide multiple language support. Having the ability to add new strings, get them translated and use them in code this early in the software development lifecycle will be a huge advantage.

That being said, in part 3 of this series, I thought I would add multilingual support using the Multilingual App Toolkit. So, come on back and join in on the fun.

I hope you enjoyed this post. If so, please share it with others.

1 Comment **Intertech Blog** 1 **Login**

Recommend Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Fourtit Aziz • a year ago

Hello,
I find your article very helpful thank you for the great work.
Could you please provide me with the source code as it is very hard for me to integrate the code from snaps.
Kindly

• Reply • Share ›

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Privacy**

Get Our Secret to Good Code Documentation Guide

Subscribe to our blog and gain access to our guide developed by our consulting teams.

Some ad blockers can block the form below.

Email

Subscribe

Recent Posts

Thousands of Passengers Stranded, Windows on Git, and Much More…

Building Full Stack Development Skills on Your Team

DDoS-ing GitHub, Google's ML Training, and More…

Top Trends in Javascript for 2018 and Beyond

SpaceX's Software, Email Overwhelm, and More…

Categories

Categories

Select Category ▼

Practical insights, tips, tutorials and examples from team of software development consultants and trainers.

Recent Posts

- Thousands of Passengers Stranded, Windows on Git, and Much More…
- Building Full Stack Development Skills on Your Team
- DDoS-ing GitHub, Google's ML Training, and More…
- Top Trends in Javascript for 2018 and Beyond
- SpaceX's Software, Email Overwhelm, and More…

Contact Details

1575 Thomas Center Drive
Eagan, MN 55122
General: 651.288.7000

Training: 651-288-7109
Consulting: 651-288-7001

Toll Free: 800-866-9884

About Intertech

Home
Consulting
IntertechU Course Selector