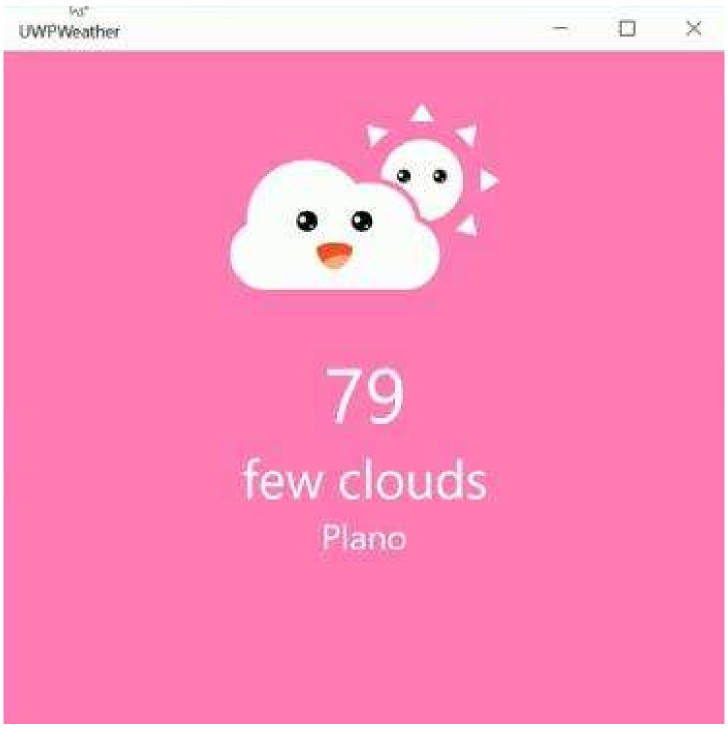
UWP-057 - UWP Weather - Introduction

The next app that we will build together is a very simple weather application that brings together a lot of cool technologies in order to produce a very interactive app. Now, simple weather apps are ubiquitous, however, building one ourselves will demonstrate some pretty cool features of Windows 10 and how to work with external services to get data to update and make your apps come alive. This app will be able to run on, both, mobile and desktop platforms and the way that it will work is - based on your current location - it's going to find the current weather conditions, your exact location, and display an image that relates to the weather condition.



One of the main things that you'll learn with this example is how to use location services for Windows 10 to determine the geo position of where your device, or your computer, is currently at. And once you have that information (latitude and longitude) we can then make calls out to an external web service - like openweathermap.org - which will return the weather for that location, and even future forecasts for the hours and days ahead.

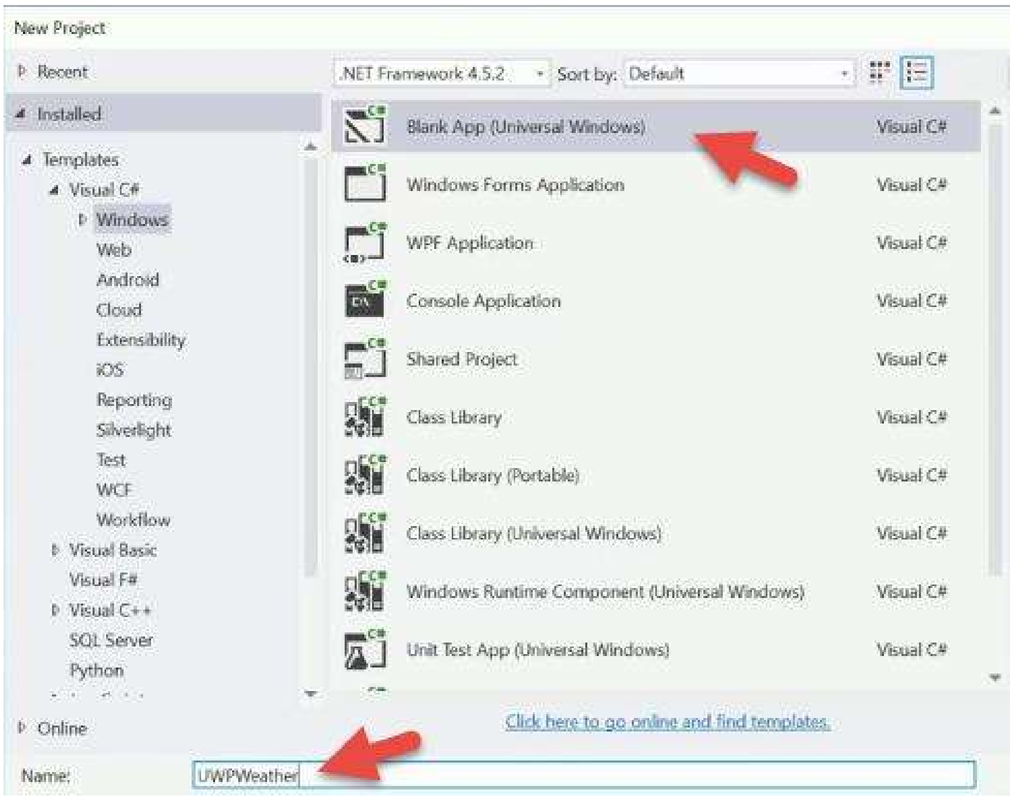
Part of this process will reveal how to retrieve data from a web service through a file format called JSON (JavaScript Object Notation), which in turn lets you deserialize that information into instances of classes that we can work with in C#. This will be helpful not only to the task at hand but also in learning a crucial development skill of serializing/deserializing data to and from formats like XML/JSON and objects in your code.

We will also look at how to work with the Phone Emulator to test our application. This will include testing not only the design and layout, but also how to change the position, and tell the location service, that we're actually coming from a given location.

And, finally, we will also take a look at how to build a live tile for our application that updates periodically with current information. Aside from building live tiles, this process will also teach you how to create a web service using ASP.NET MVC 5 and host it out of your website.

UWP-058 - UWP Weather - Setup and Working with the Weather API

Let's get started in creating our weather app by opening up a blank UWP project and naming it "UWPWeather."


New Project

Recent
Installed
Templates
Visual C#
Windows
Web
Android
Cloud
Extensibility
iOS
Reporting
Silverlight
Test
WCF
Workflow
Visual Basic
Visual F#
Visual C++
SQL Server
Python
Online
.NET Framework 4.5.2
Sort by: Default
Blank App (Universal Windows) Visual C#
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Shared Project Visual C#
Class Library Visual C#
Class Library (Portable) Visual C#
Class Library (Universal Windows) Visual C#
Windows Runtime Component (Universal Windows) Visual C#
Unit Test App (Universal Windows) Visual C#
Click here to go online and find templates.

Name: UWPWeather

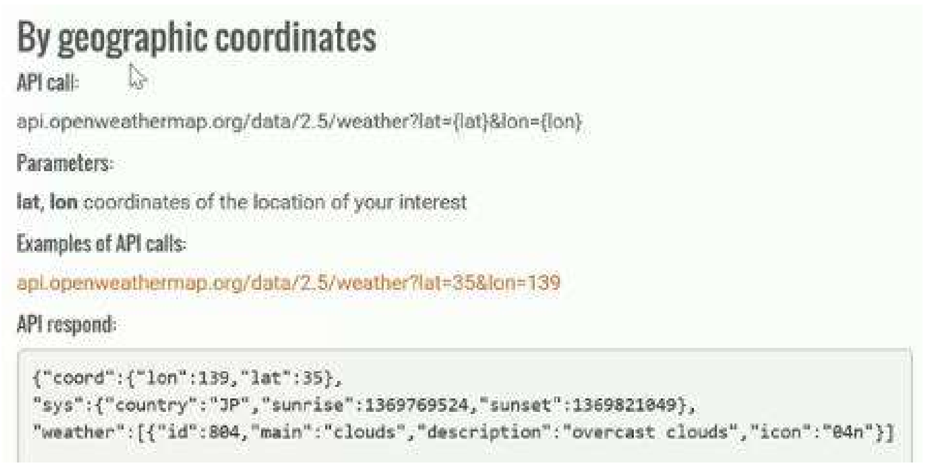
Before we get started with this project, note that we are going to be pulling weather data from

openweathermap.org

If you intend to publish an app using the API found on that site you will likely have to purchase a license depending on how many people intend to use the app. But for the simple purpose of demonstration used in this lesson you will not have to worry about that. With that in mind you should turn your attention next to the API information found at

openweathermap.org/current

Notice that there are a variety of ways to make API calls to this resource, depending on the information you provide via the URL. The one we will be using in our application will be the one that uses geographic coordinates



By geographic coordinates

API call:

api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}

Parameters:

lat, lon coordinates of the location of your interest

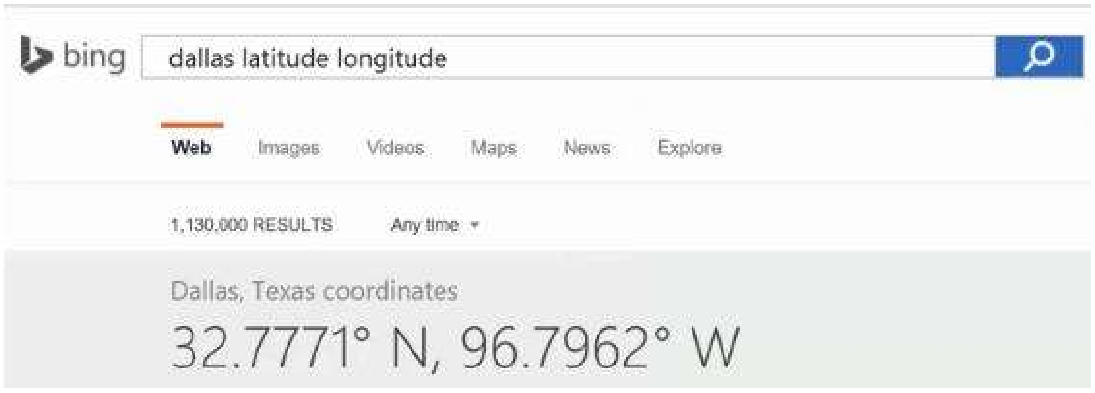
Examples of API calls:

api.openweathermap.org/data/2.5/weather?lat=35&lon=139

API respond:

```
{"coord":{"lon":139,"lat":35},
"sys":{"country":"JP","sunrise":1369769524,"sunset":1369821049},
"weather":[{"id":804,"main":"clouds","description":"overcast clouds","icon":"04n"}]
```

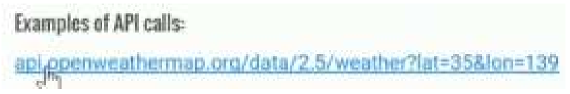
With that being the case let's begin by finding the latitude/longitude for the location we have in mind (in this case, we're interested in the coordinates for Dallas, Texas



Dallas, Texas coordinates

32.7771° N, 96.7962° W

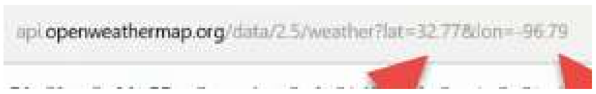
Copy that information somewhere and keep it handy while going back to the openweathermap.org API page – clicking on the example API call link



Examples of API calls:

api.openweathermap.org/data/2.5/weather?lat=35&lon=139

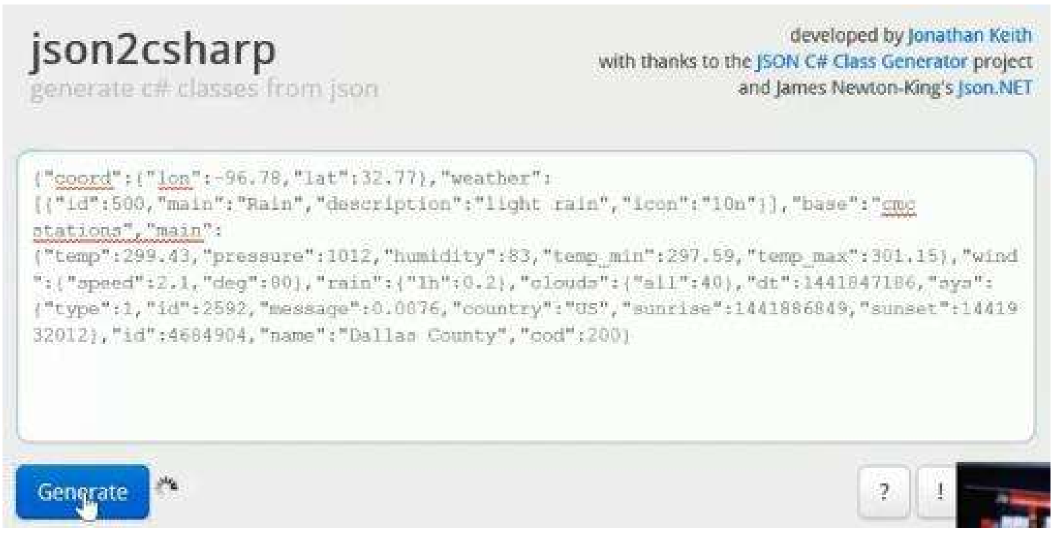
And replace the values for latitude and longitude with the values we obtained in the web search (latitude being roughly 32.77, while longitude is roughly -96.79)



api.openweathermap.org/data/2.5/weather?lat=32.77&lon=-96.79

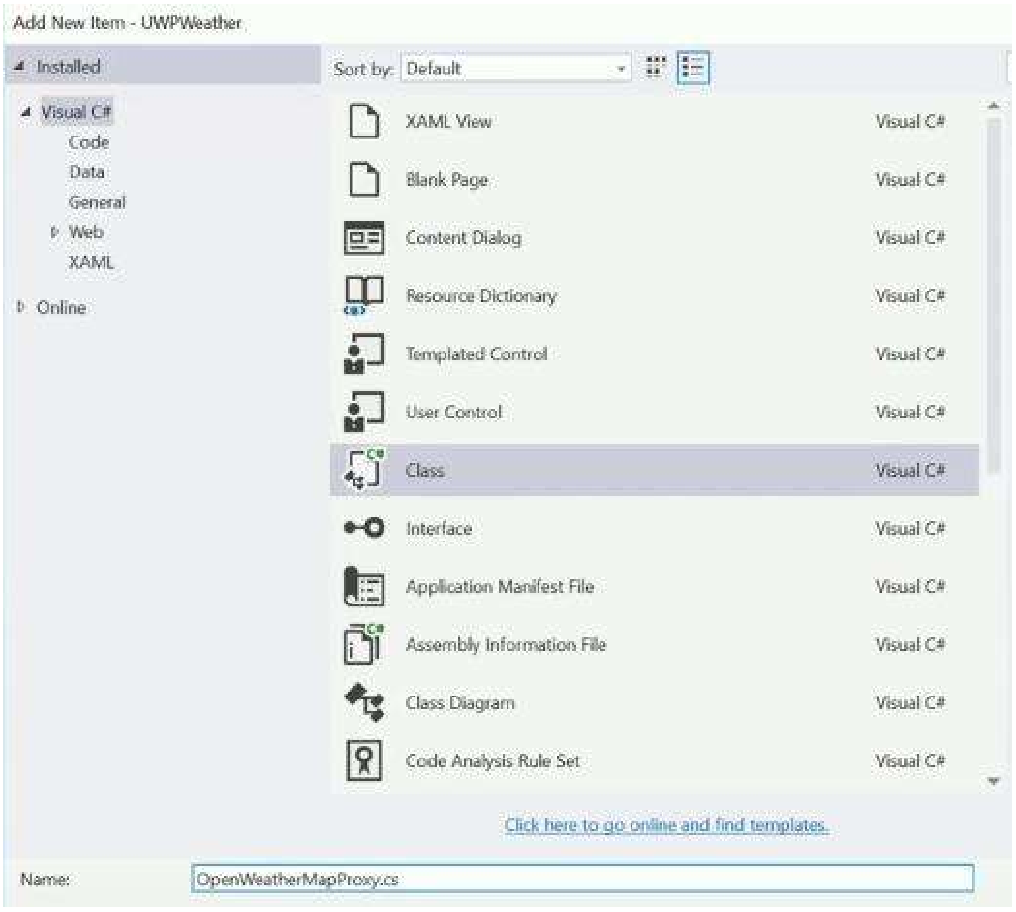
What you will get back is a bunch of information arranged in JSON format. Quickly convert that Object Notation into C# code by copying and pasting it into the converter found at

Json2csharp.com


json2csharp
generate c# classes from json
developed by Jonathan Keith
with thanks to the JSON C# Class Generator project
and James Newton-King's Json.NET
Generate
?
!

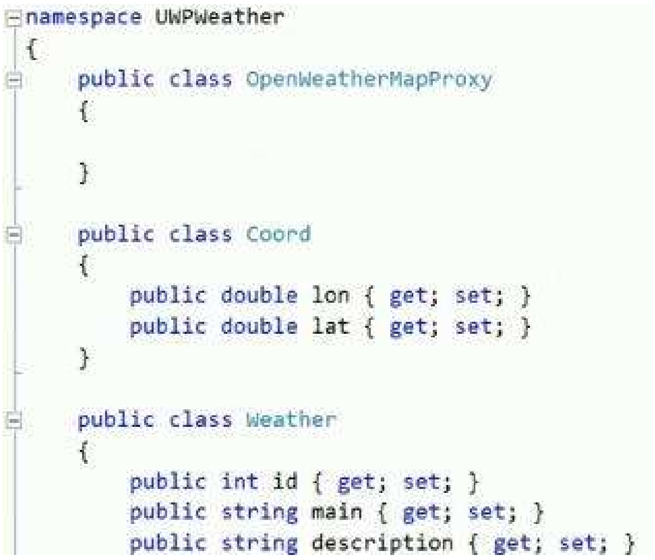
```
{"coord":{"lon":-96.78,"lat":32.77},"weather":
[{"id":500,"main":"Rain","description":"light rain","icon":"10n"}],"base":"cmc
stations","main":
{"temp":299.43,"pressure":1012,"humidity":83,"temp_min":297.59,"temp_max":301.15},"wind
":{"speed":2.1,"deg":80},"rain":{"1h":0.2},"clouds":{"all":40},"dt":1441847186,"sys":
{"type":1,"id":2592,"message":0.0876,"country":"US","sunrise":1441886849,"sunset":14419
32012},"id":4684904,"name":"Dallas County","cod":200}
```

The resulting conversion is a deserialized version of this information, represented in C# Classes and Properties. Copy and paste all of that into a new class within your Visual Studio project. Call that class "OpenWeatherMapProxy"


Add New Item - UWPWeather
Installed
Visual C#
Code
Data
General
Web
XAML
Online
Sort by: Default
XAML View Visual C#
Blank Page Visual C#
Content Dialog Visual C#
Resource Dictionary Visual C#
Templated Control Visual C#
User Control Visual C#
Class Visual C#
Interface Visual C#
Application Manifest File Visual C#
Assembly Information File Visual C#
Class Diagram Visual C#
Code Analysis Rule Set Visual C#
Click here to go online and find templates.

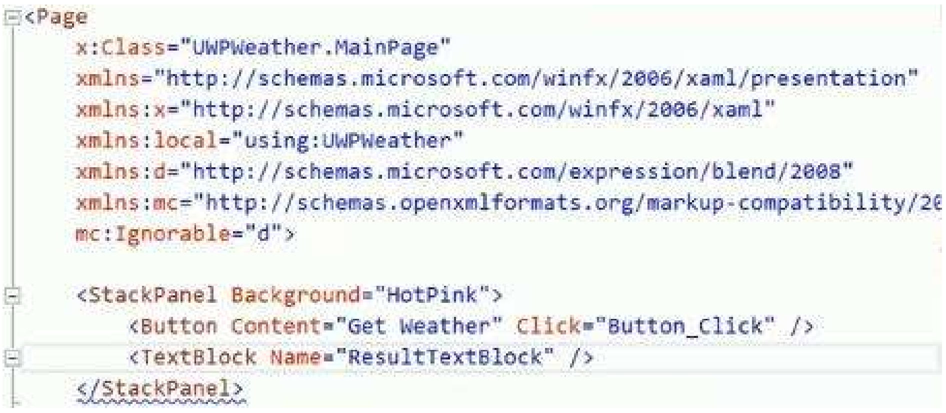
Name: OpenWeatherMapProxy.cs

Step 1: alongside that class just paste in everything you copied from the JSON-to-C# conversion (partially represented in the image below)



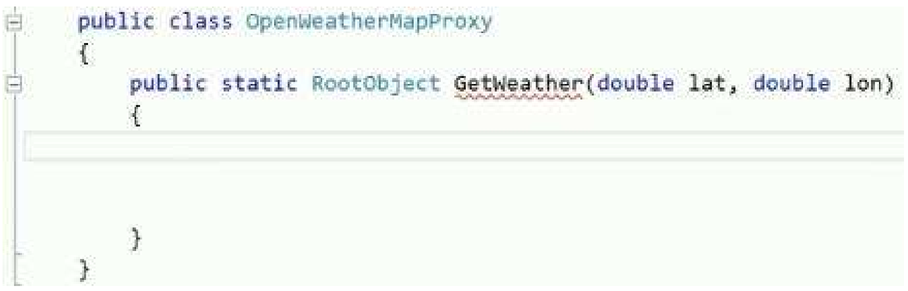
```
namespace UWPWeather
{
    public class OpenWeatherMapProxy
    {

    }

    public class Coord
    {
        public double lon { get; set; }
        public double lat { get; set; }
    }

    public class Weather
    {
        public int id { get; set; }
        public string main { get; set; }
        public string description { get; set; }
```

Step 2: Now in the MainPage.xaml start with a simple StackPanel so we can simply test grabbing the data and displaying it



```
<Page
    x:Class="UWPWeather.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:UWPWeather"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/20
    mc:Ignorable="d">

    <StackPanel Background="HotPink">
        <Button Content="Get Weather" Click="Button_Click" />
        <TextBlock Name="ResultTextBlock" />
    </StackPanel>
```

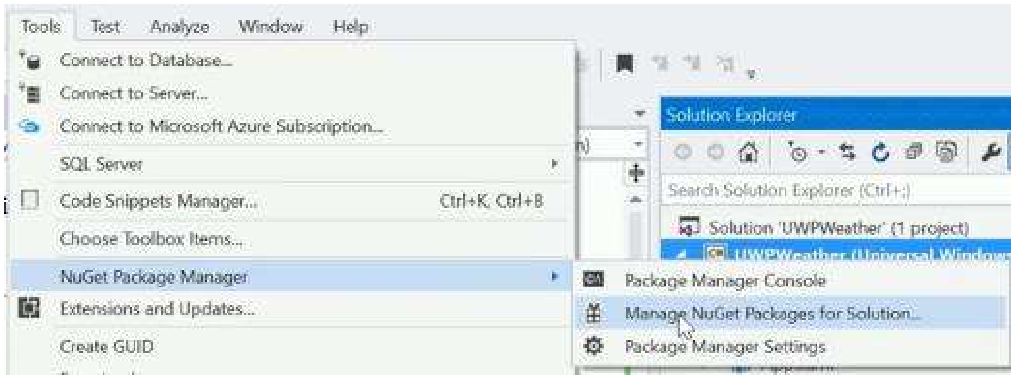
Step 3: create the static method called GetWeather() with the aim of making a call to the outside web service via an HttpClient



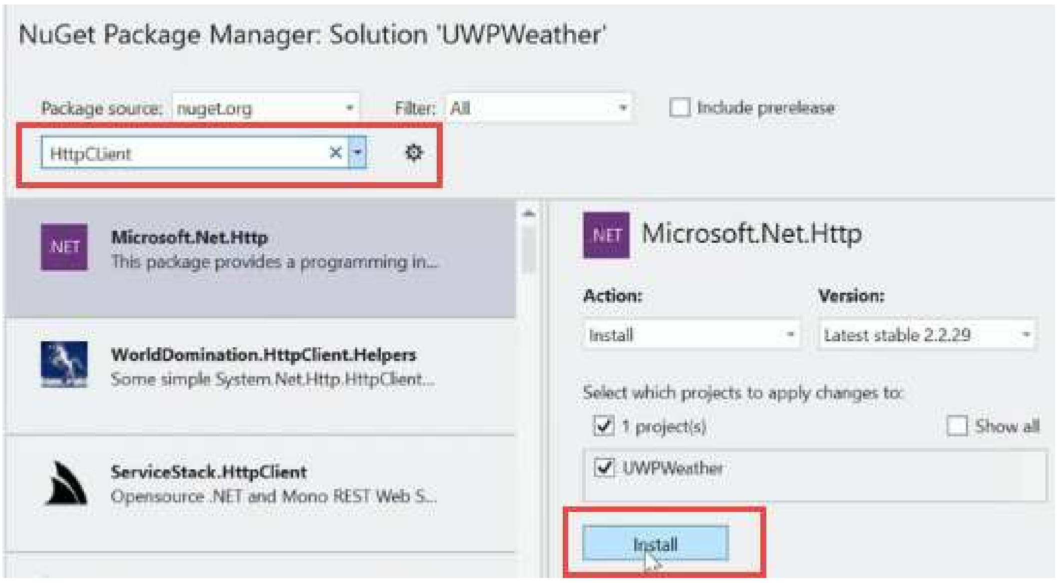
```
public class OpenWeatherMapProxy
{
    public static RootObject GetWeather(double lat, double lon)
    {


    }
}
```

Step 4: add the HttpClient to this code block by going to

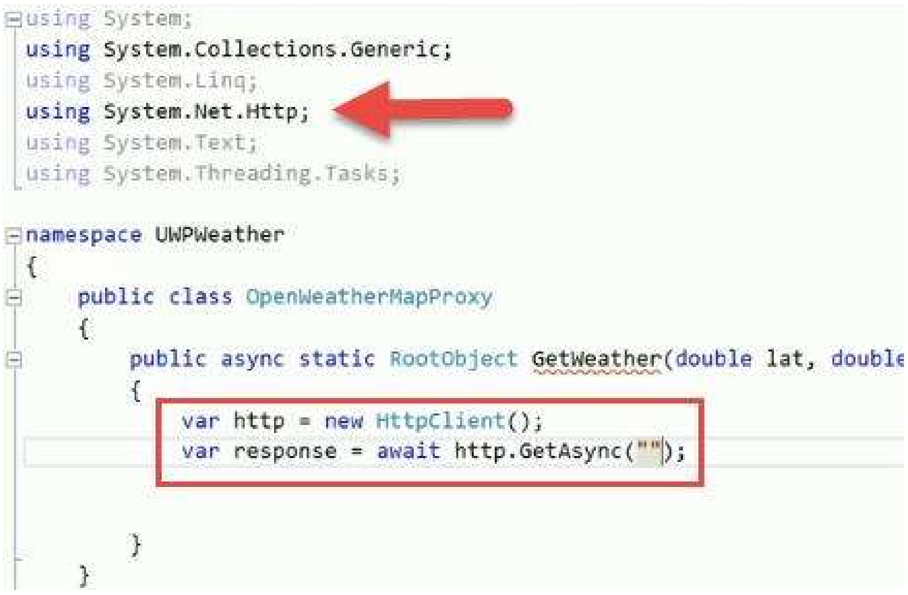
Tools > NuGet Package Manager > Manage NuGet Packages for Solution


Tools
Test
Analyze
Window
Help
Connect to Database...
Connect to Server...
Connect to Microsoft Azure Subscription...
SQL Server
Code Snippets Manager... Ctrl+K, Ctrl+B
Choose Toolbox Items...
NuGet Package Manager
Extensions and Updates...
Create GUID
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'UWPWeather' (1 project)
UWPWeather (Universal Window
Package Manager Console
Manage NuGet Packages for Solution...
Package Manager Settings

Step 5: search for the HttpClient and install it (click and accept the resulting nag screens)


NuGet Package Manager: Solution 'UWPWeather'
Package source: nuget.org
Filter: All
Include prerelease
HttpClient
Microsoft.Net.Http
This package provides a programming in...
WorldDomination.HttpClient.Helpers
Some simple System.Net.Http.HttpClient...
ServiceStack.HttpClient
Opensource .NET and Mono REST Web S...
Microsoft.Net.Http
Action:
Install
Version:
Latest stable 2.2.29
Select which projects to apply changes to:
1 project(s)
Show all
UWPWeather
Install

And back in your code add the using statement and code as follows

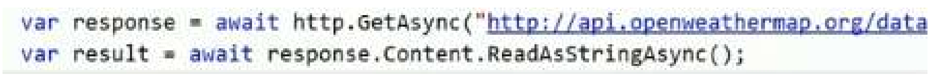


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace UWPWeather
{
    public class OpenWeatherMapProxy
    {
        public async static RootObject GetWeather(double lat, double
        {
            var http = new HttpClient();
            var response = await http.GetAsync("");


        }
    }
```

Step 6: Now, for testing purposes, simply copy and paste that previous URL we had before in between the double quotes of the http.GetAsync() method. That URL again is

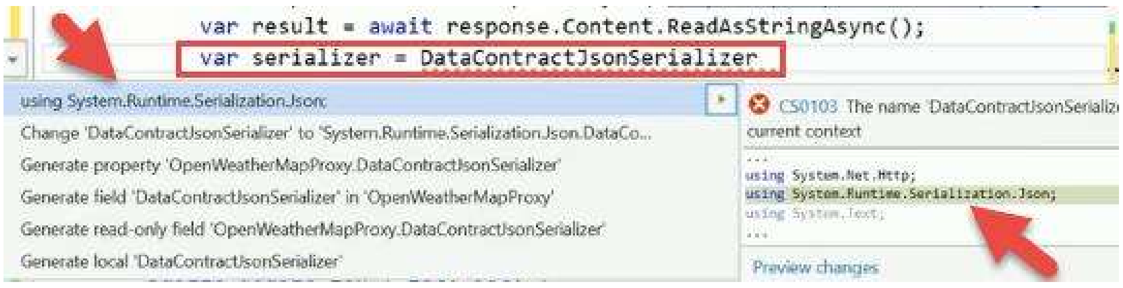
http://api.openweathermap.org/data/2.5/weather?lat=32.77&lon=-96.79

Now, take all of that data pulled from the web server and stored locally in a variable called "response," and store the content as a string in another variable called "result"



```
var response = await http.GetAsync("http://api.openweathermap.org/data
var result = await response.Content.ReadAsStringAsync();
```

Step 6: Now let's use a serializer to handle the serialization/deserialization of JSON/Object data by typing in this reference (hit the Ctrl key + the period key while DataContractJsonSerializer is selected to import the namespace, else add the reference to "using System.Runtime.Serialization.Json;" manually at the top of your script)



```
var result = await response.Content.ReadAsStringAsync();
var serializer = DataContractJsonSerializer
```

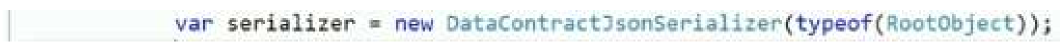
using System.Runtime.Serialization.Json;
Change 'DataContractJsonSerializer' to 'System.Runtime.Serialization.Json.DataCo...
Generate property 'OpenWeatherMapProxy.DataContractJsonSerializer'
Generate field 'DataContractJsonSerializer' in 'OpenWeatherMapProxy'
Generate read-only field 'OpenWeatherMapProxy.DataContractJsonSerializer'
Generate local 'DataContractJsonSerializer'

CS0103 The name 'DataContractJsonSerializ
current context

```
...
using System.Net.Http;
using System.Runtime.Serialization.Json;
using System.Text;
...
```

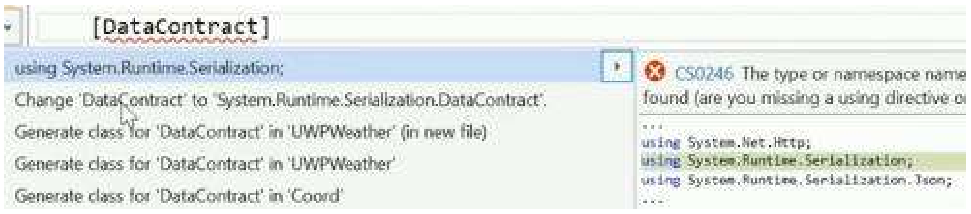
Preview changes

And complete this line of code as follows



```
var serializer = new DataContractJsonSerializer(typeof(RootObject));
```

Step 7: Add an attribute to each class in order to allow serialization. Type in "[DataContract]" above the class and import the requisite namespace (using System.Runtime.Serialization) as follows

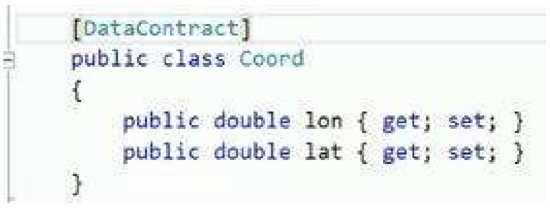


```
[DataContract]
```

using System.Runtime.Serialization;
Change 'DataContract' to 'System.Runtime.Serialization.DataContract'.
Generate class for 'DataContract' in 'UWPWeather' (in new file)
Generate class for 'DataContract' in 'UWPWeather'
Generate class for 'DataContract' in 'Coord'

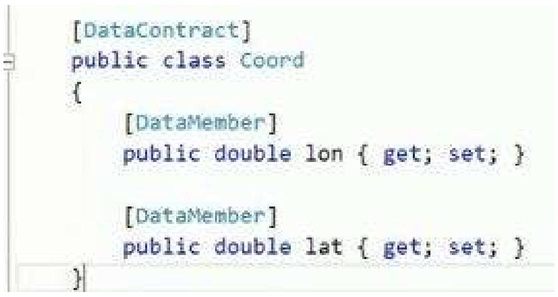
CS0246 The type or namespace name
found (are you missing a using directive o

```
...
using System.Net.Http;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Json;
...
```



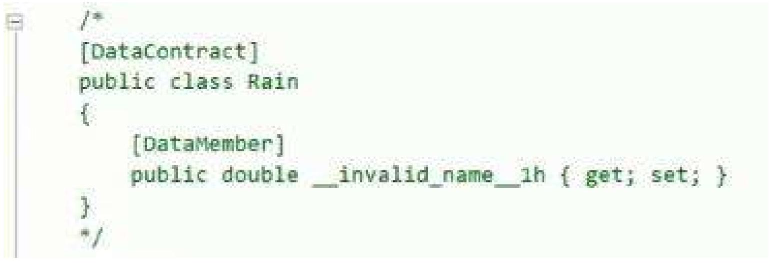
```
[DataContract]
public class Coord
{
    public double lon { get; set; }
    public double lat { get; set; }
}
```

That's for serializing the class, now to serialize the properties within the class add an attribute to the top of each property (do this for every class and property that we copied and pasted from the JSON-to-C# conversion)

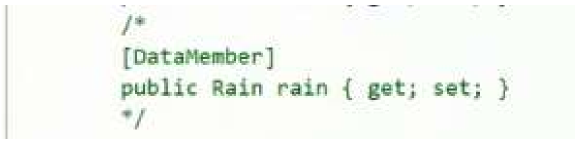


```
[DataContract]
public class Coord
{
    [DataMember]
    public double lon { get; set; }

    [DataMember]
    public double lat { get; set; }
}
```

Step 8: Go ahead now and comment out some code that might be invalid or not important to our app



```
/*
[DataContract]
public class Rain
{
    [DataMember]
    public double __invalid_name__1h { get; set; }
}
*/
```



```
/*
[DataMember]
public Rain rain { get; set; }
*/
```

Step 9: Add “using System.IO;” to your using declarations at the top of the document and add a MemoryStream in your code and properly encode JSON using UTF8

```
            var serializer = new DataContractJsonSerializer(typeof(RootObject));

            var ms = new MemoryStream(Encoding.UTF8.GetBytes(result));
```

A MemoryStream allows us to just have data come in and go out - whenever you have something sending data at a different rate of speed than something accepting data, you use a Stream. By using a MemoryStream here we are just keeping everything in memory until it all gets sorted out. Data will go in one end, and then it'll be taken out as it's needed on the other end.

Step 10: Now, let’s get data out of the serializer and return it at the end of our method

```
            var ms = new MemoryStream(Encoding.UTF8.GetBytes(result));
            var data = (RootObject)serializer.ReadObject(ms);

            return data;
```

And to make this work properly as an async method, we need to modify the return type in the method signature as follows

```
        public async static Task<RootObject> GetWeather(double lat, double lon)
        {
```

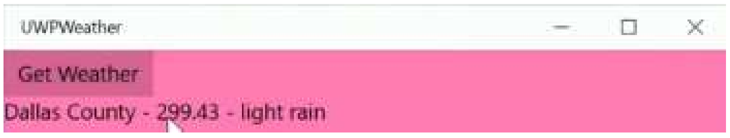
Step 11: Let’s make use of this code by referencing it in MainPage.xaml.cs

```
        private async void Button_Click(object sender, RoutedEventArgs e)
        {
            RootObject myWeather = await OpenWeatherMapProxy.GetWeather(20.0, 30.0);

            ResultTextBlock.Text = myWeather.name + " - " + myWeather.
```

And to get the temperature and weather description complete this line of code with

```
myWeather.main.temp + " - " + myWeather.weather[0].description;
```

The result when you run the program should look something like this

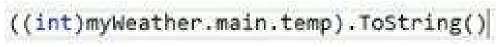


```
Dallas County - 299.43 - light rain
```

Step 12: Get the temperature reading to look right by adding the following modifications to your code

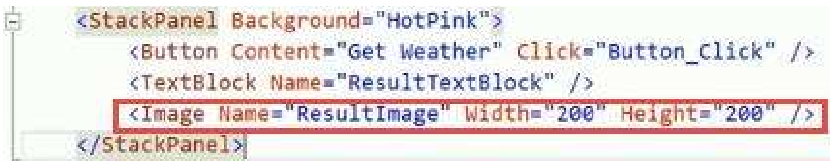
```
tp://api.openweathermap.org/data/2.5/weather?lat=32.77&lon=-96.7[illegible]&units=imperial
```

Now cast the temperature to int, and after that convert it to a string to get rid of the decimal with this modification



```
((int)myWeather.main.temp).ToString()
```

Step 13: You may have noticed that the OpenWeatherMap API gives us access to icons, with specific names for each (you can find them at openweathermap.org/weather-conditions). Let's implement an icon in our app by adding the following

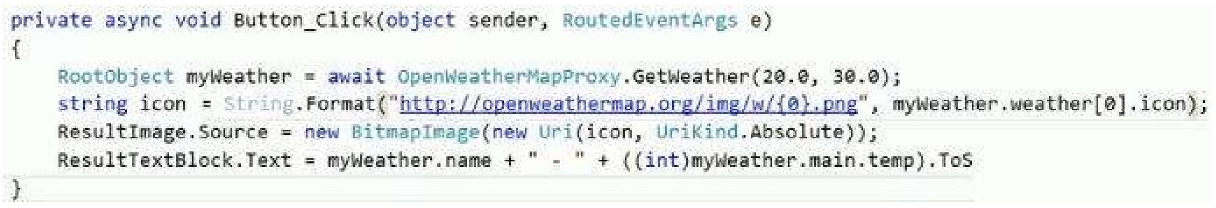


```
    <StackPanel Background="HotPink">
        <Button Content="Get Weather" Click="Button_Click" />
        <TextBlock Name="ResultTextBlock" />
        <Image Name="ResultImage" Width="200" Height="200" />
    </StackPanel>
```

And in MainPage.xaml.cs start by adding the namespace

Using Windows.UI.Xaml.Media.Imaging;

And, finally, add the code to the Button_Click method



```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    RootObject myWeather = await OpenWeatherMapProxy.GetWeather(20.0, 30.0);
    string icon = String.Format("http://openweathermap.org/img/w/{0}.png", myWeather.weather[0].icon);
    ResultImage.Source = new BitmapImage(new Uri(icon, UriKind.Absolute));
    ResultTextBlock.Text = myWeather.name + " - " + ((int)myWeather.main.temp).ToS
}
```

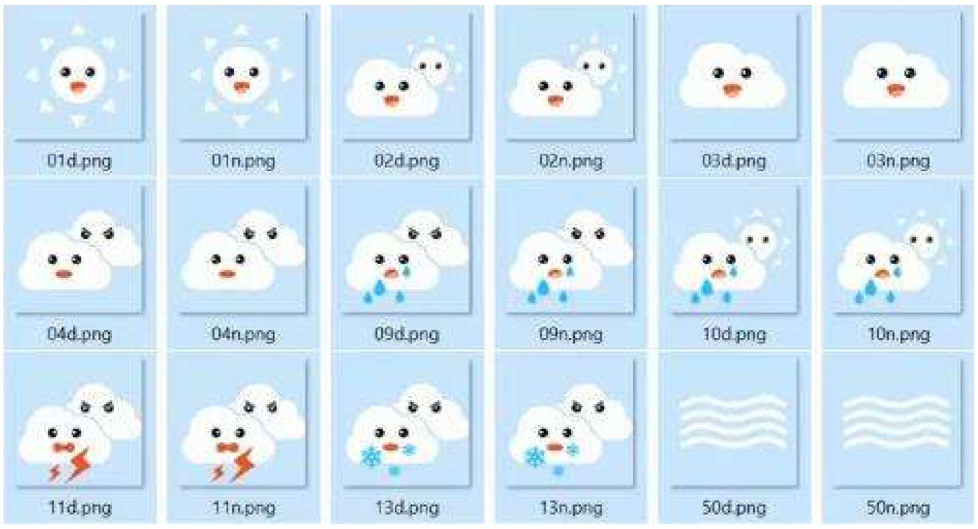
When you run the program you should now see an image in the application, however, it's a bit generic so in the next lesson we will look at replacing it with our own local asset.

UWP-059 - UWP Weather - Accessing the GPS Location

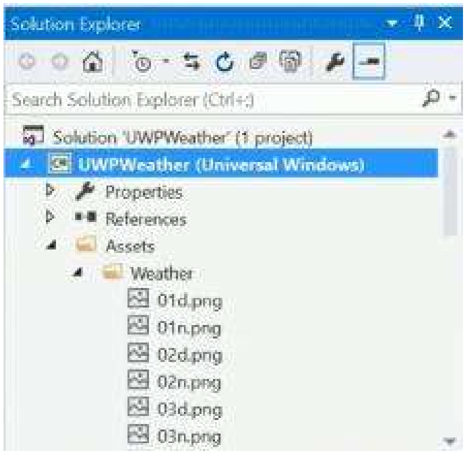
For this lesson you can download the icons from

http:// openweathermap.org/weather-conditions

Or, you can create your own icons with the same dimensions and names used in this lesson. The icons used in this lesson look like this

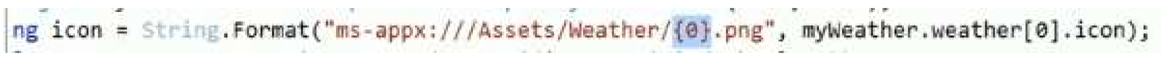


Step 1: Create a folder called "Weather" and drag/drop your icons into that folder



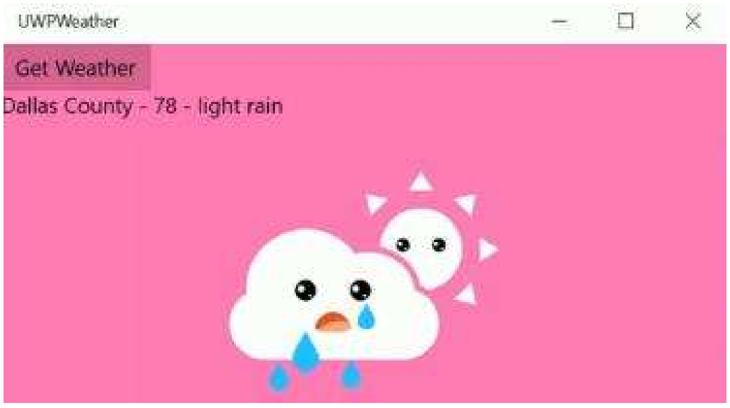
Solution 'UWPWeather' (1 project)
UWPWeather (Universal Windows)
Properties
References
Assets
Weather
01d.png
01n.png
02d.png
02n.png
03d.png
03n.png

Step 2: Modify the code from the previous lesson to refer to your local directory of icons, rather than the web server



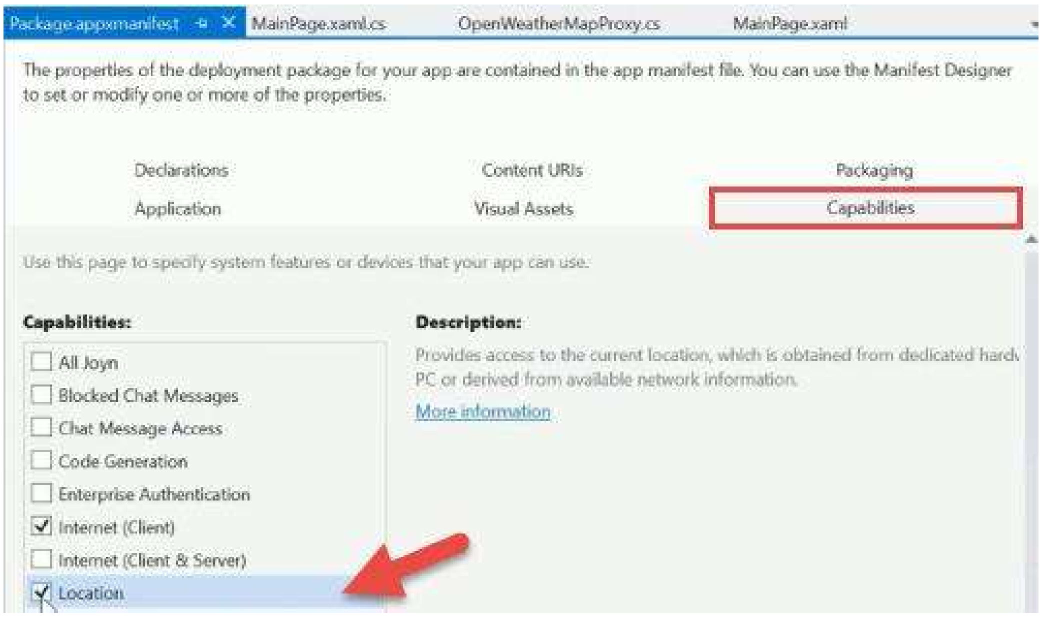
```
ng icon = String.Format("ms-appx:///Assets/Weather/{0}.png", myWeather.weather[0].icon);
```

Your application should now show the icon when you run it



Dallas County - 78 - light rain

Step 3: In order to utilize a sensor on the platform you're running on, you have to declare a capability in the Package.appxmanifest



The properties of the deployment package for your app are contained in the app manifest file. You can use the Manifest Designer to set or modify one or more of the properties.

Use this page to specify system features or devices that your app can use.

Capabilities:

- [ ] All Joyn
- [ ] Blocked Chat Messages
- [ ] Chat Message Access
- [ ] Code Generation
- [ ] Enterprise Authentication
- [✓] Internet (Client)
- [ ] Internet (Client & Server)
- [✓] Location

Description:

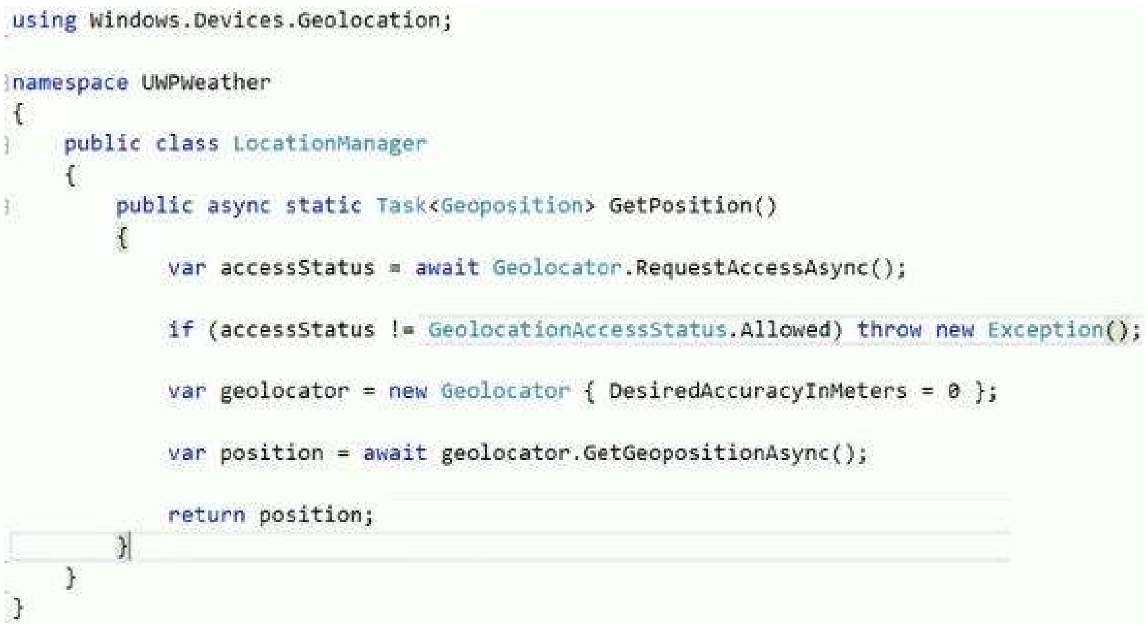
Provides access to the current location, which is obtained from dedicated hard PC or derived from available network information.

For more information, and code, that allows for more sophisticated location services in your app visit the following URL

Msdn.microsoft.com/en-us/library/windows/xaml/mt219698.aspx

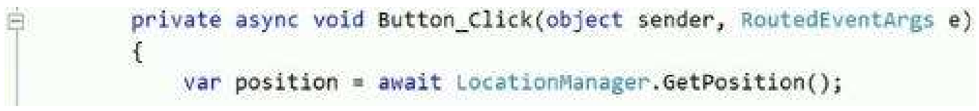
Or, use a search engine and look for “Windows Dev Center Get current location.” However, for this demonstration we will just keep things simple.

Step 4: Add a new class to the project and call it “LocationManager.” Add the using declaration and method shown here



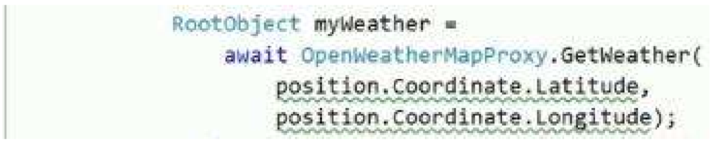
```
using Windows.Devices.Geolocation;

namespace UWPWeather
{
    public class LocationManager
    {
        public async static Task<Geoposition> GetPosition()
        {
            var accessStatus = await Geolocator.RequestAccessAsync();

            if (accessStatus != GeolocationAccessStatus.Allowed) throw new Exception();

            var geolocator = new Geolocator { DesiredAccuracyInMeters = 0 };

            var position = await geolocator.GetGeopositionAsync();

            return position;
        }
    }
}
```

And make a reference to GetPosition() within the Button_Click() method



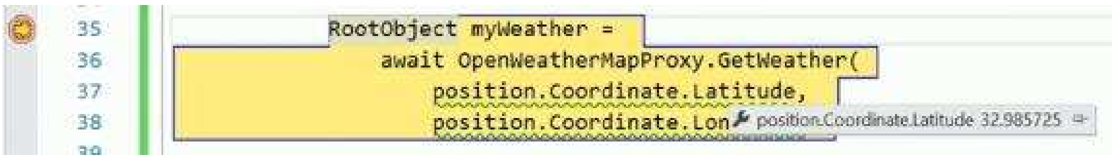
```
        private async void Button_Click(object sender, RoutedEventArgs e)
        {
            var position = await LocationManager.GetPosition();
```

And in that same method modify the GetWeather() call with the following code



```
            RootObject myWeather =
                await OpenWeatherMapProxy.GetWeather(
                    position.Coordinate.Latitude,
                    position.Coordinate.Longitude);
```

Note the green squiggly lines means this code is considered deprecated, so there may come a time in the future when it is no longer supported. You can now test this line of code by setting a breakpoint and checking the value for these variables (with the green squiggly lines)

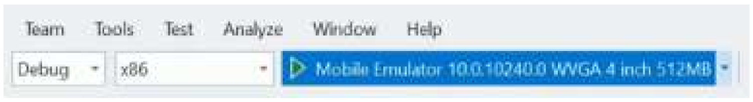


```
            RootObject myWeather =
                await OpenWeatherMapProxy.GetWeather(
                    position.Coordinate.Latitude,
                    position.Coordinate.Lon
```

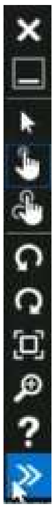
In the next lesson we will use the phone emulator, simulating changing the geo position within the country, and see what it would look like if the app was running from Seattle, or from New York, or from Chicago, etc.

UWP-060 - UWP Weather - Testing Location in the Phone Emulator

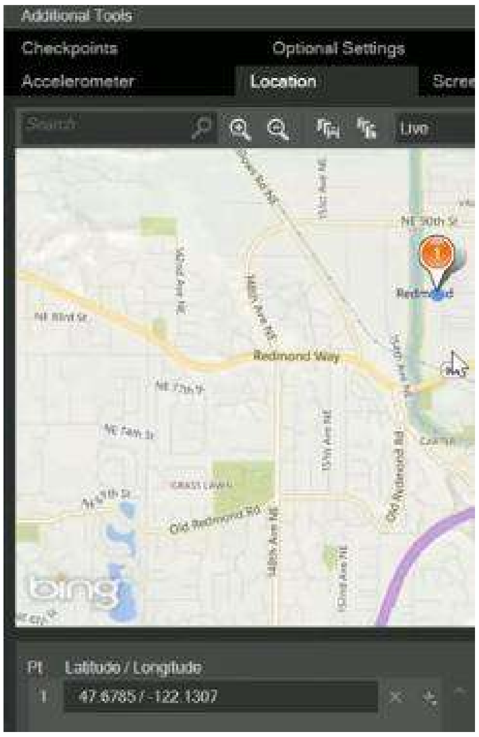
In this lesson, we will learn a little bit more about how to use some of the features of the Phone Emulator. The first thing you should do is change the environment in Visual Studio to a Mobile Emulator and the run the application



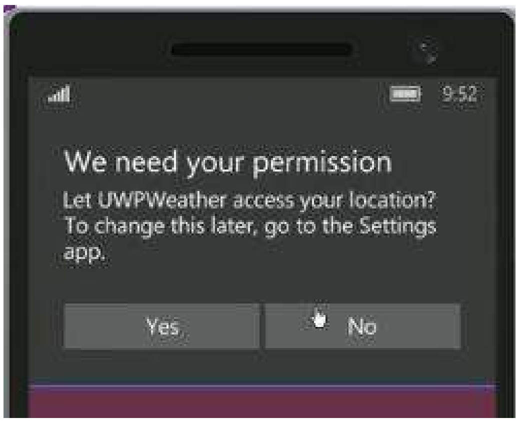
What we will want to do is test our app to see if it's grabbing coordinates correctly - getting geo coordinates from different places in the world, and so forth. So, once the emulator is running, double click the chevron icon to open up the Tools pane for the emulator and go to the location tab.



And on the map that pops up, zoom in and click on a location to tell the emulator that that is your location (in this case we are choosing Redmond, Washington)



Step 1: In your running app click on the “Get Weather” button and it will ask you to allow location access for the UWPWeather app

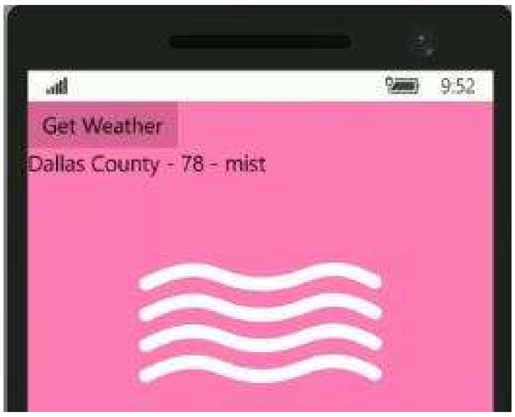


9:52

We need your permission

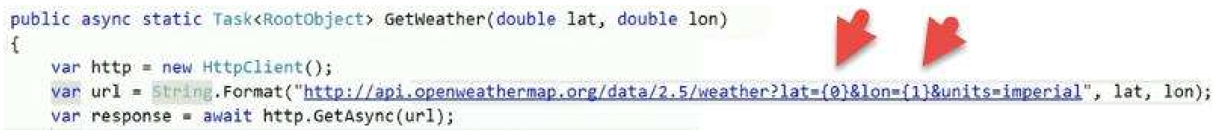
Let UWPWeather access your location? To change this later, go to the Settings app.

You may notice that even though the location is different from before, it still reports the previous location’s weather that was hard-coded in the app previously



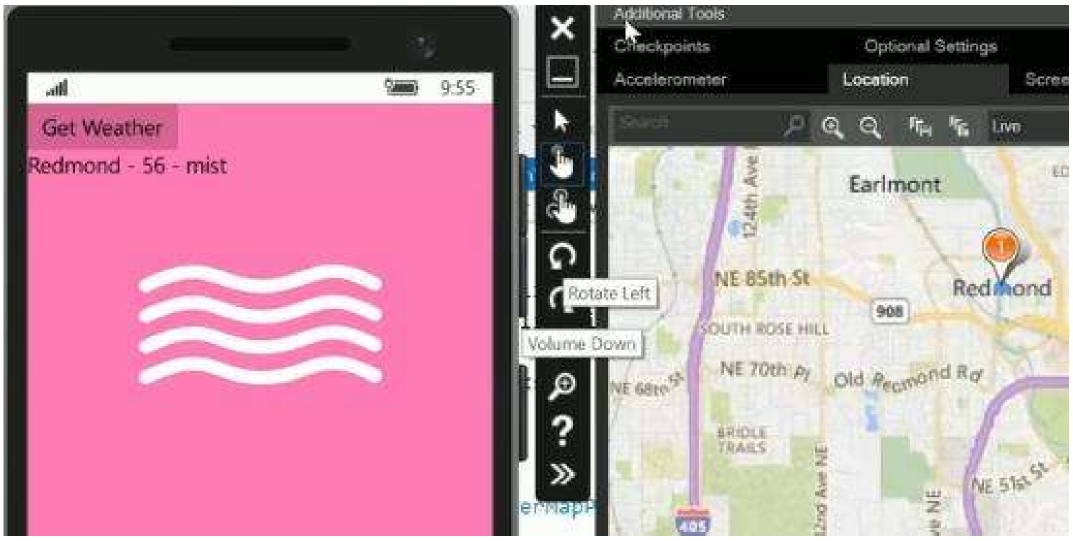
Dallas County - 78 - mist

Step 2: So, in the code we need to modify the references for the latitude and longitude we supplied previously. To do so, make all of the changes to the GetWeather() method shown here, paying particular attention to the formatting used in the URL (using {0} and {1} as placeholders for the lat, and lon arguments)



```
public async static Task<RootObject> GetWeather(double lat, double lon)
{
    var http = new HttpClient();
    var url = String.Format("http://api.openweathermap.org/data/2.5/weather?lat={0}&lon={1}&units=imperial", lat, lon);
    var response = await http.GetAsync(url);
```

Now run the application, again choosing a different location on the map and you should see it reflected in the app itself.



Redmond - 56 - mist

As you can see the emulator can be used to troubleshoot and test out functionality in your code that is otherwise difficult to pin down.

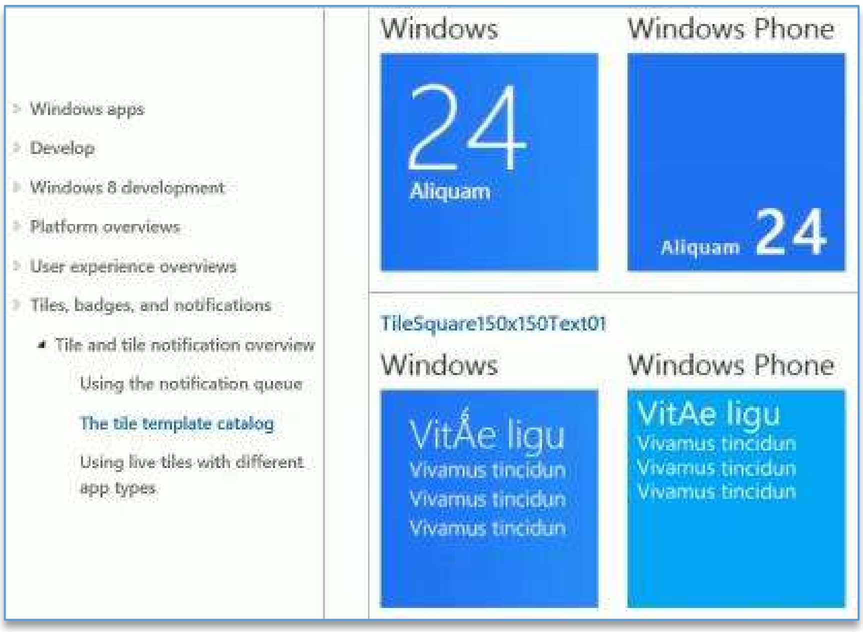
UWP-061 - UWP Weather - Updating the Tile with Periodic Notifications

Our application is now feature complete, for the most part. However, we still need to address some layout issues and make it look a little nicer but before we get to that point, let's add one more cool feature. And that feature is adding Live Tiles to your application, and have those tiles update based on content from your application.

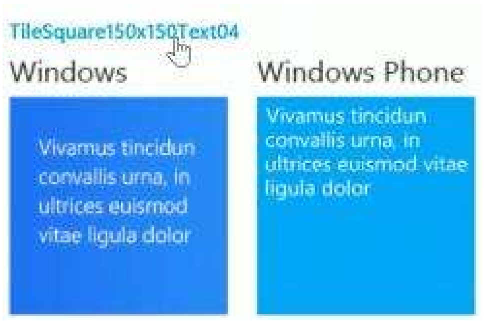
First, we will change the content of the tile on demand, which means the program is going to actively push data into that tile. And then second, we will schedule periodic notification - and give it some source online to pull data from - to present data to the user.

For more information on tiles, refer to the tile template catalog

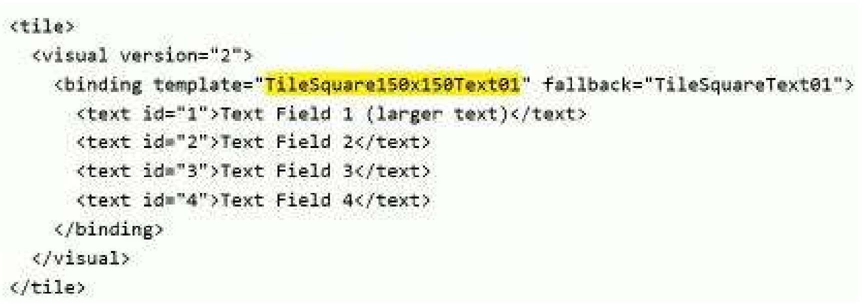
http://bit.do/template-catalog



Take note of the slightly different ways a tile may appear depending on the platform it will be displayed on. When you find a tile that fits the scenario you have in mind for your app, click on the link to that tile in order to get more descriptive information on it, including its template.

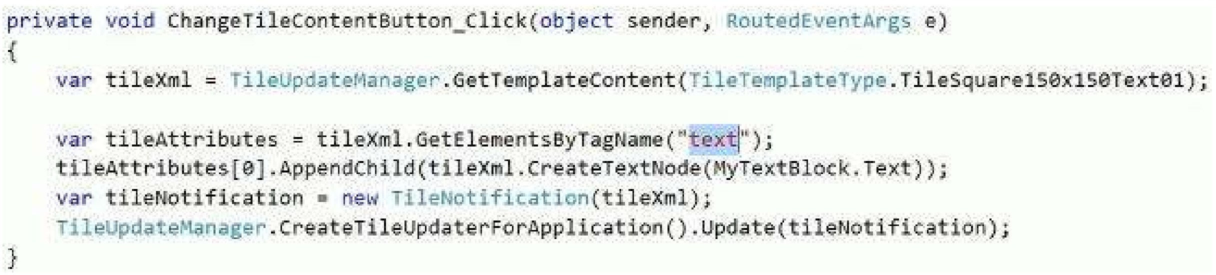

TileSquare150x150Text04
Windows
Windows Phone
Vivamus tincidun convallis urna, in ultrices euismod vitae ligula dolor
Vivamus tincidun convallis urna, in ultrices euismod vitae ligula dolor

And you can pull, from that resource, the XML template information for use within your code



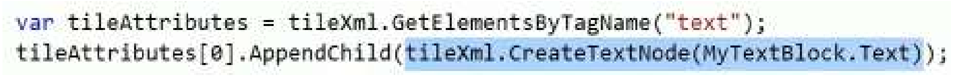
```
<tile>
  <visual version="2">
    <binding template="TileSquare150x150Text01" fallback="TileSquareText01">
      <text id="1">Text Field 1 (larger text)</text>
      <text id="2">Text Field 2</text>
      <text id="3">Text Field 3</text>
      <text id="4">Text Field 4</text>
    </binding>
  </visual>
</tile>
```

Below you will find some example code which shows this relationship between the XML template and your code



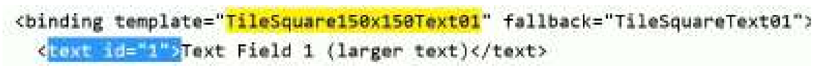
```
private void ChangeTileContentButton_Click(object sender, RoutedEventArgs e)
{
    var tileXml = TileUpdateManager.GetTemplateContent(TileTemplateType.TileSquare150x150Text01);

    var tileAttributes = tileXml.GetElementsByTagName("text");
    tileAttributes[0].AppendChild(tileXml.CreateTextNode(MyTextBlock.Text));
    var tileNotification = new TileNotification(tileXml);
    TileUpdateManager.CreateTileUpdaterForApplication().Update(tileNotification);
}
```

Let’s break this code block down a bit, and describe what it’s doing. Here, we will find an element with the name “text” and we will add a child to that first element with whatever we typed into that TextBlock.



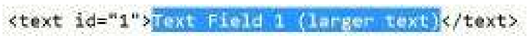
```
var tileAttributes = tileXml.GetElementsByTagName("text");
tileAttributes[0].AppendChild(tileXml.CreateTextNode(MyTextBlock.Text));
```

So, in the XML, it will find this text element, and then add a child to the element (highlighted in blue)



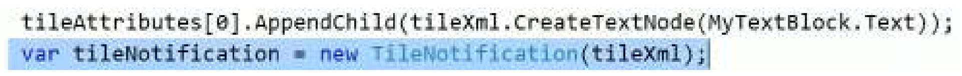
```
<binding template="TileSquare150x150Text01" fallback="TileSquareText01">
  <text id="1">Text Field 1 (larger text)</text>
```

And it will then fill in the actual text between that element's text tags, with whatever we typed in.



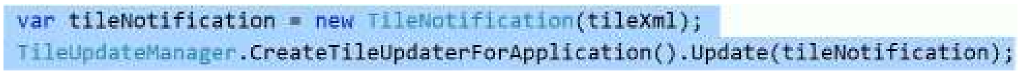
```
<text id="1">Text Field 1 (larger text)</text>
```

And then after it's done that, it will add that new XML to a new tile notification object.



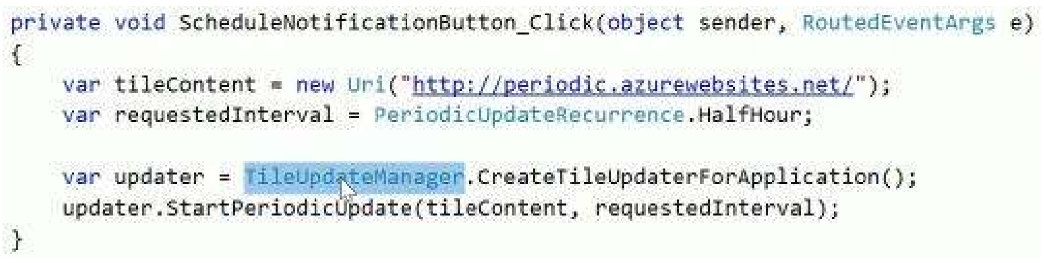
```
tileAttributes[0].AppendChild(tileXml.CreateTextNode(MyTextBlock.Text));
var tileNotification = new TileNotification(tileXml);
```

And then tell the TileUpdateManager to go update our application's tile with this other tile template that we've implemented.



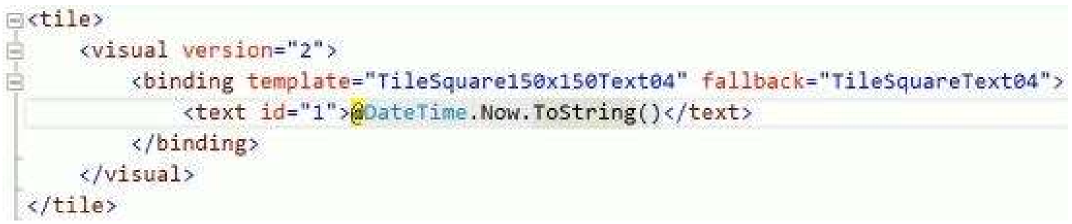
```
var tileNotification = new TileNotification(tileXml);
TileUpdateManager.CreateTileUpdaterForApplication().Update(tileNotification);
```

Okay, so that's the basic course of events here in five lines of code. Next, we have a ScheduleNotificationButton_Click() event asks the TileUpdateManager to just update every half hour with this URL that's given to it.



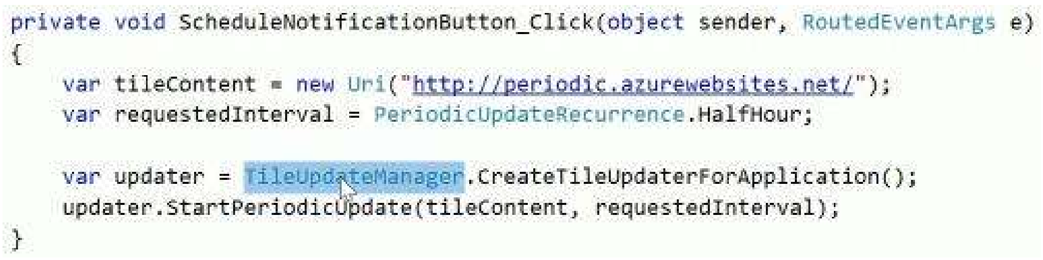
```
private void ScheduleNotificationButton_Click(object sender, RoutedEventArgs e)
{
    var tileContent = new Uri("http://periodic.azurewebsites.net/");
    var requestedInterval = PeriodicUpdateRecurrence.HalfHour;

    var updater = TileUpdateManager.CreateTileUpdaterForApplication();
    updater.StartPeriodicUpdate(tileContent, requestedInterval);
}
```

So, now we would need to just create some sort of web service, at that URL, that will return XML sufficient for updating our tile. Below is part of an example web service, running on Azure, that simply returns the string formatted current DateTime (and, as you can see, it's using a particular tile template to display that information).



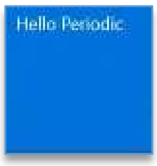
```
<tile>
    <visual version="2">
        <binding template="TileSquare150x150Text04" fallback="TileSquareText04">
            <text id="1">@DateTime.Now.ToString()</text>
        </binding>
    </visual>
</tile>
```

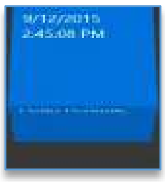
So, again, this bit of code reaches out to that web service (roughly every half hour) and pushes that remote information to update a local tile

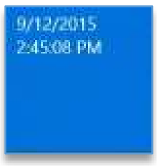


```
private void ScheduleNotificationButton_Click(object sender, RoutedEventArgs e)
{
    var tileContent = new Uri("http://periodic.azurewebsites.net/");
    var requestedInterval = PeriodicUpdateRecurrence.HalfHour;

    var updater = TileUpdateManager.CreateTileUpdaterForApplication();
    updater.StartPeriodicUpdate(tileContent, requestedInterval);
}
```

And, within your device you will see the tile automatically animate and display updated information in real time (similar to this)



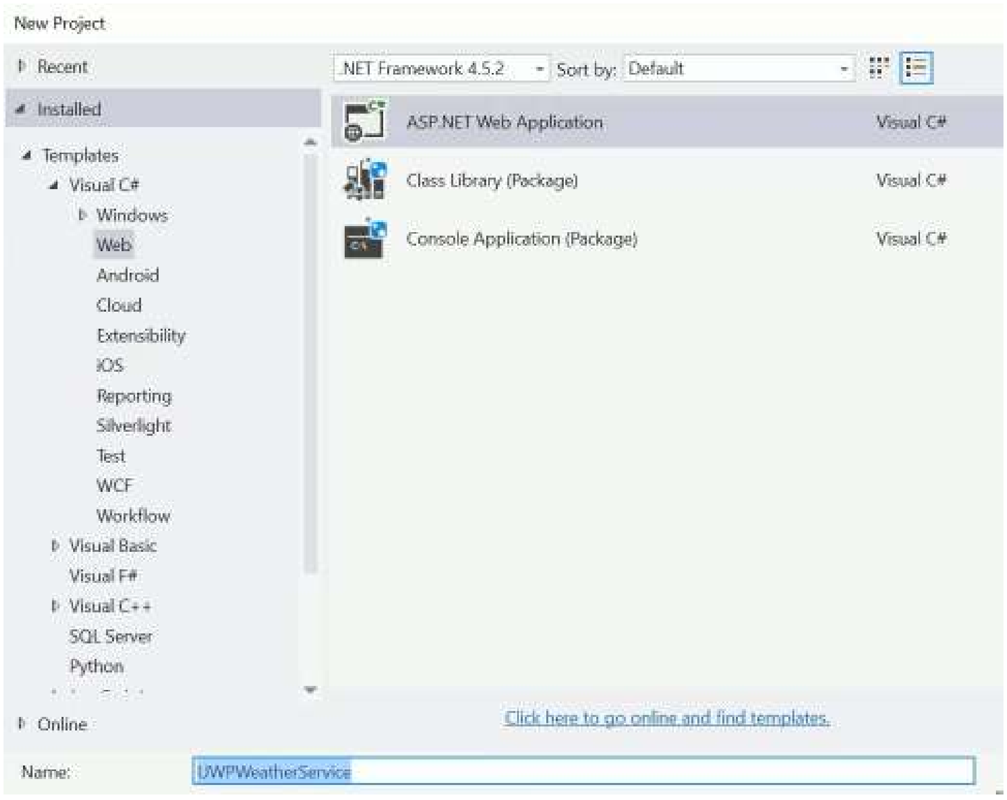




So, that's the basics for how this process is supposed to work. For more info on periodic updating (polling sequence, etc), take a look at this URL

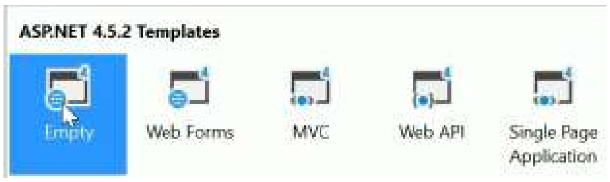
http://bit.do/periodic

Now, let's implement these ideas into our current UWPWeather project. To do that, let's first create a web service through an ASP.NET Web Application project called UWPWeatherService (Note that this is an example only and will be a unique web service at a unique URL, you will want to create your own unique web service to implement this on your end. Also note that this web service will not be available when you are reading this)

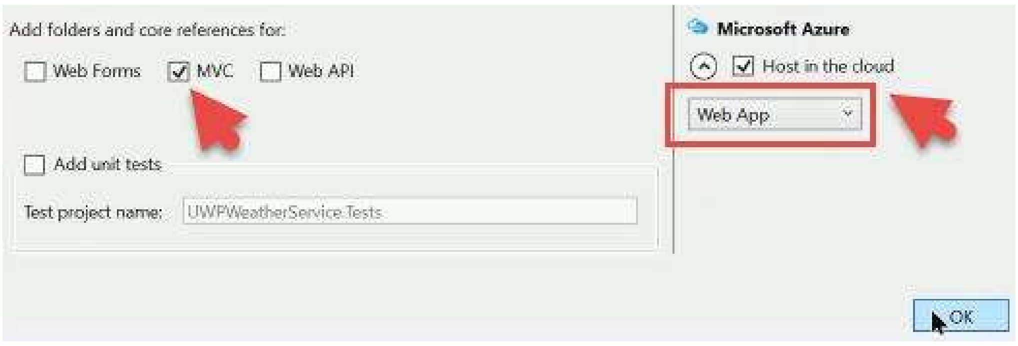


Name: UWPWeatherService

And select an Empty template for this project



Also, it will be an MVC Web App, hosted in the cloud


Add folders and core references for:

☐ Web Forms ☑ MVC ☐ Web API

☐ Add unit tests

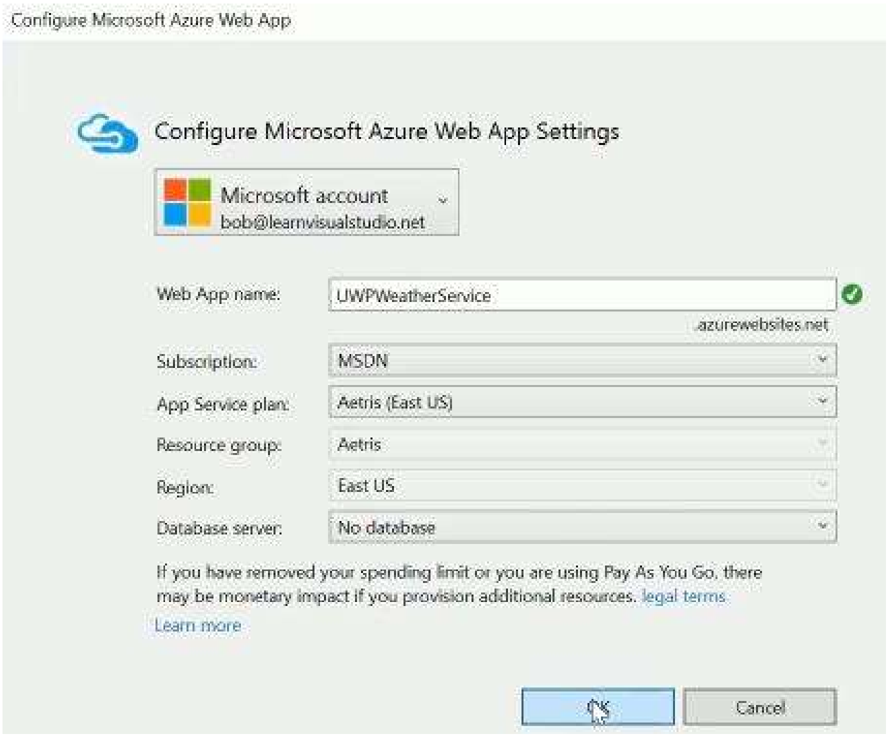
Test project name: UWPWeatherService.Tests

Microsoft Azure

☑ Host in the cloud

Web App

After that login with your Microsoft Account, and set up the web service using the same name as the project (this will be hosted, in this case, at http://UWPWeatherService.azurewebsites.net)


Configure Microsoft Azure Web App

Configure Microsoft Azure Web App Settings

Microsoft account
bob@learnvisualstudio.net

Web App name: UWPWeatherService .azurewebsites.net

Subscription: MSDN

App Service plan: Aetris (East US)

Resource group: Aetris

Region: East US

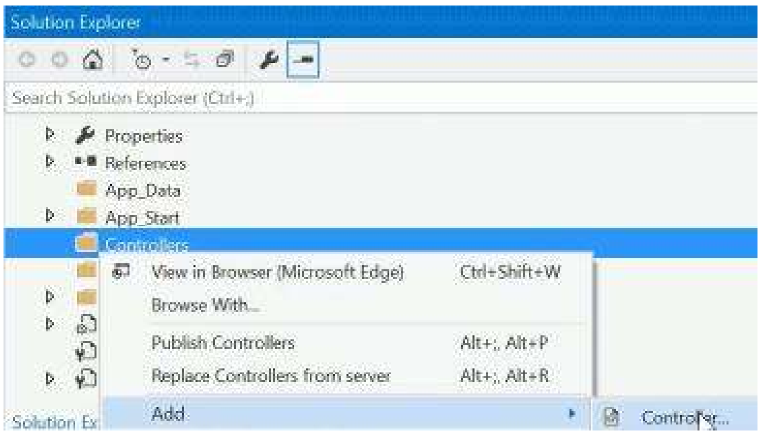
Database server: No database

If you have removed your spending limit or you are using Pay As You Go, there may be monetary impact if you provision additional resources. legal terms

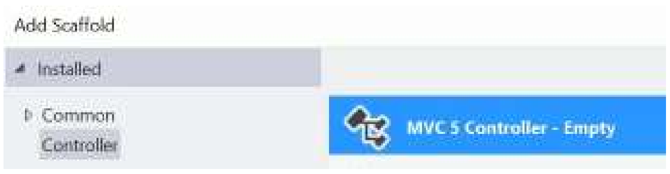
Learn more

This should setup the publishing credentials, as well as the project itself for the MVC application. The first thing to do is right click on “Controllers” in the Solution Explorer and select

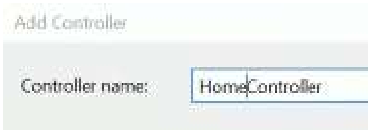
Add > Controller…


Solution Explorer
Search Solution Explorer (Ctrl+;)
Properties
References
App_Data
App_Start
Controllers
View in Browser (Microsoft Edge) Ctrl+Shift+W
Browse With...
Publish Controllers Alt+;, Alt+P
Replace Controllers from server Alt+;, Alt+R
Add
Controller...
Solution Ex

And select “MVC 5 Controller – Empty”

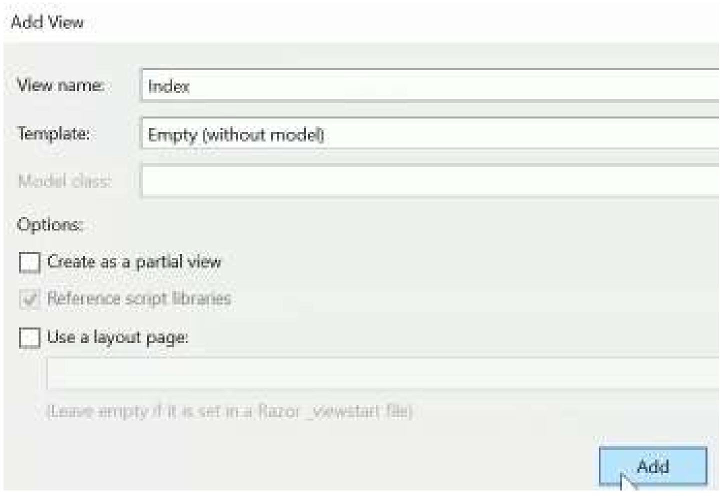


And then in the next dialog add a Controller called “HomeController”



Controller name: HomeController

Once the project is open, add a View by right clicking empty space in HomeController.cs and selecting from the menu “Add View,” and then in that dialog add it as follows



View name: Index

Template: Empty (without model)

Model class:

Options:

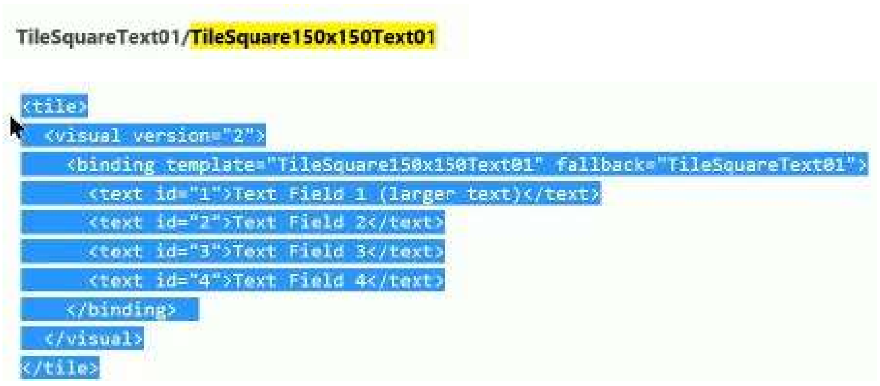
☐ Create as a partial view

✓ Reference script libraries

☐ Use a layout page:

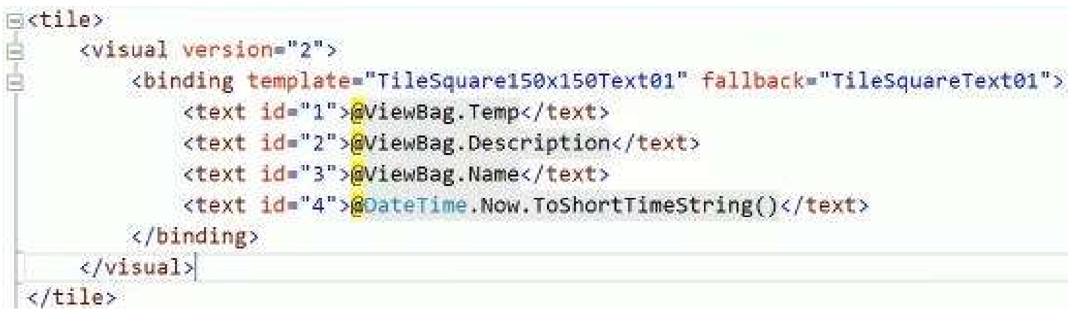
(Leave empty if it is set in a Razor _viewstart file)

Now, remove everything from Index.cshtml and from the online template catalog we referred to earlier copy and paste, into Index.cshtml, the following XML from TileSquareText01/TileSquare150x150Text01

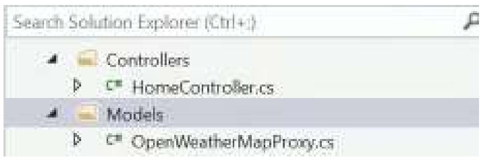

TileSquareText01/TileSquare150x150Text01

```
<tile>
  <visual version="2">
    <binding template="TileSquare150x150Text01" fallback="TileSquareText01">
      <text id="1">Text Field 1 (larger text)</text>
      <text id="2">Text Field 2</text>
      <text id="3">Text Field 3</text>
      <text id="4">Text Field 4</text>
    </binding>
  </visual>
</tile>
```

And in Index.cshtml edit the XML as follows


```
<tile>
    <visual version="2">
        <binding template="TileSquare150x150Text01" fallback="TileSquareText01">
            <text id="1">@ViewBag.Temp</text>
            <text id="2">@ViewBag.Description</text>
            <text id="3">@ViewBag.Name</text>
            <text id="4">@DateTime.Now.ToShortTimeString()</text>
        </binding>
    </visual>
</tile>
```

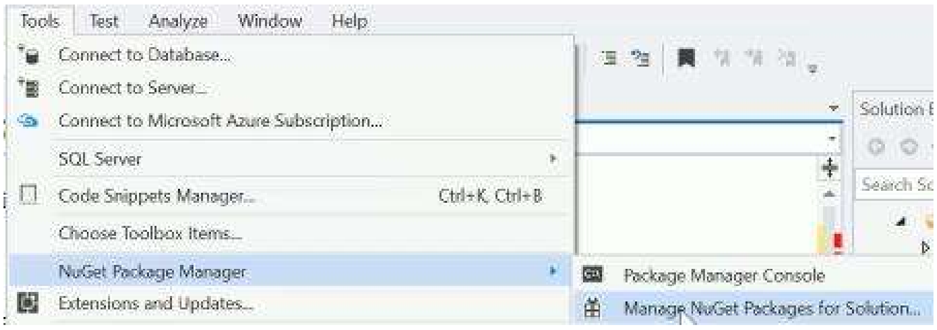
And then in the Solution Explorer, add a class to the Models folder called “OpenWeatherMapProxy”



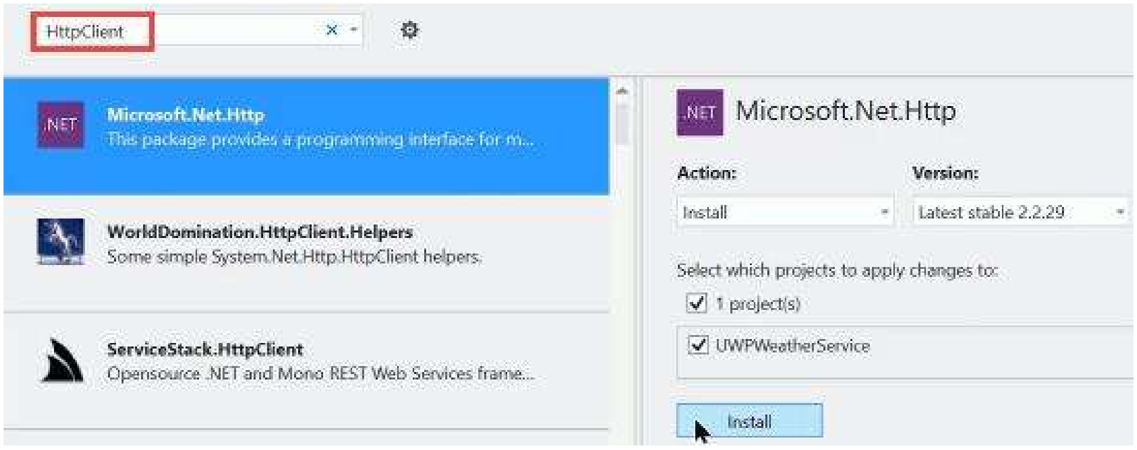
And this will be identical to the file of the same name in the UWPWeather application, so simply copy and paste the contents from that file into this new one within UWPWeatherService.

Now, add an HttpClient by going to

Tools > NuGet Package Manager > Manage NuGet Packages for Solution…



And add an HttpClient (accept and click OK to the following prompts)


HttpClient

Microsoft.Net.Http
This package provides a programming interface for m...

WorldDomination.HttpClient.Helpers
Some simple System.Net.Http.HttpClient helpers.

ServiceStack.HttpClient
Opensource .NET and Mono REST Web Services frame...

Microsoft.Net.Http

Action: Install

Version: Latest stable 2.2.29

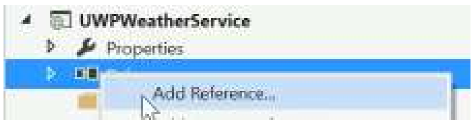
Select which projects to apply changes to:

☑ 1 project(s)

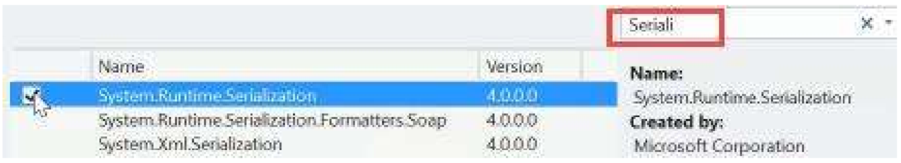
☑ UWPWeatherService

Install

In the Solution Explorer, right-click on “References” and select “Add Reference…”



And add a reference to System.Runtime.Serialization


Seriali

	Name	Version
☑	System.Runtime.Serialization	4.0.0.0
	System.Runtime.Serialization.Formatters.Soap	4.0.0.0
	System.Xml.Serialization	4.0.0.0

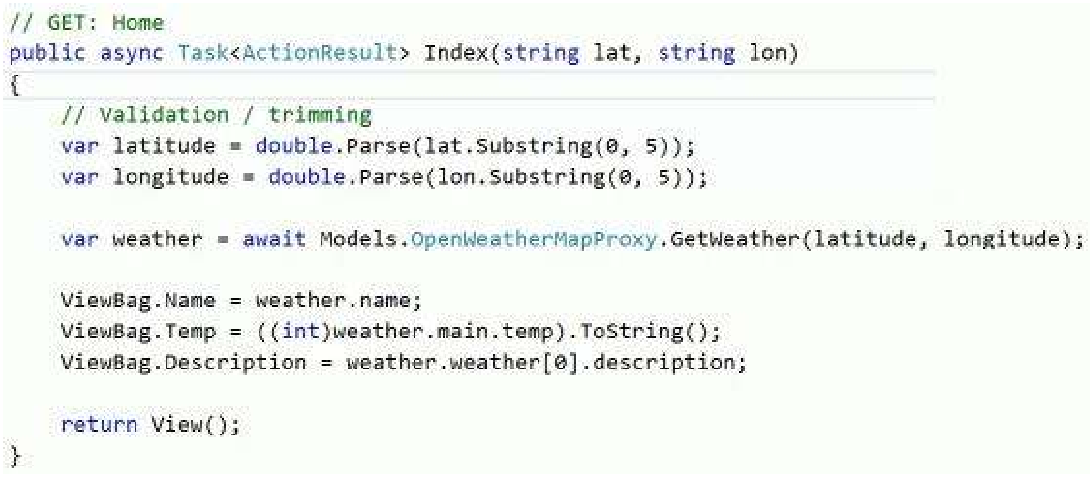
Name: System.Runtime.Serialization

Created by: Microsoft Corporation

Also add these namespaces to the top of OpenWeatherMapProxy.cs

```
using System.Threading.Tasks;
using System.Net.Http;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Json;
using System.IO;
using System.Text;
```

Now, inside of HomeController.cs, write the following method that populates those values from Index.cshtml and also takes in a given location



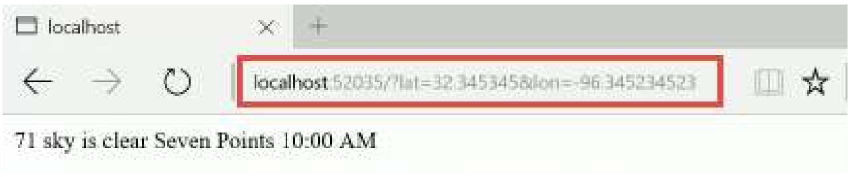
```
// GET: Home
public async Task<ActionResult> Index(string lat, string lon)
{
    // Validation / trimming
    var latitude = double.Parse(lat.Substring(0, 5));
    var longitude = double.Parse(lon.Substring(0, 5));

    var weather = await Models.OpenWeatherMapProxy.GetWeather(latitude, longitude);

    ViewBag.Name = weather.name;
    ViewBag.Temp = ((int)weather.main.temp).ToString();
    ViewBag.Description = weather.weather[0].description;

    return View();
}
```

And add to the top of HomeController.cs the using statement

```
using System.Threading.Tasks;
```

If you now run this locally

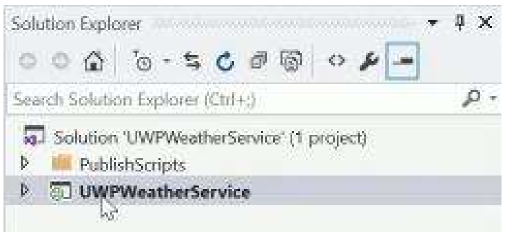
And supply values for latitude/longitude, you should see something like this (the values will be unique for you, of course)



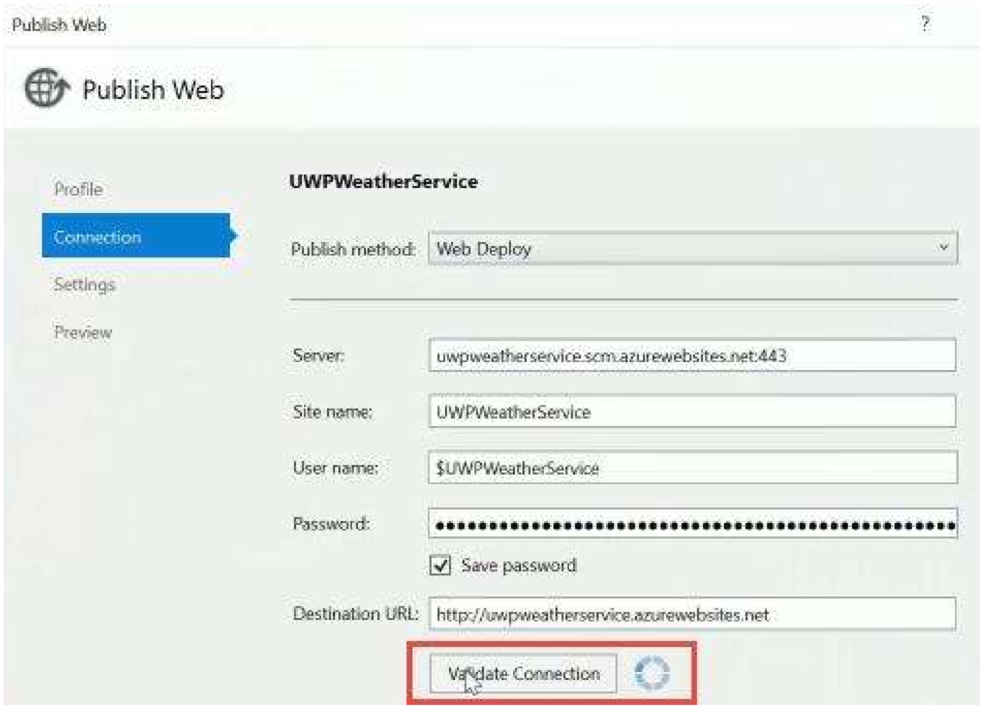
```
localhost:52035/?lat=32.345345&lon=-96.345234523
```

```
71 sky is clear Seven Points 10:00 AM
```

Now, you can deploy it to your remote service by right-clicking on the project name in the Solution Explorer, and selecting "Publish"


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'UWPWeatherService' (1 project)
PublishScripts
UWPWeatherService

Click on "Validate Connection"



UWPWeatherService

Publish method: Web Deploy

Server: uwpweatherservice.scm.azurewebsites.net:443

Site name: UWPWeatherService

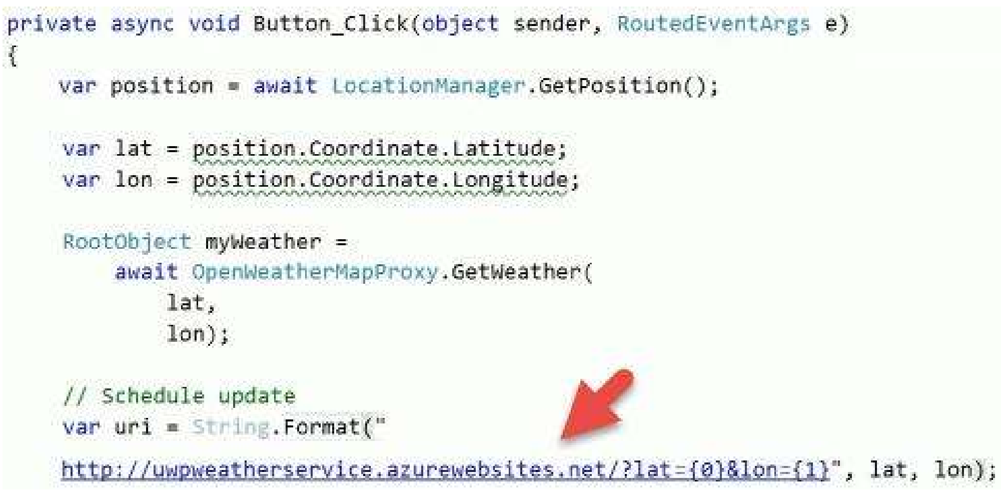
User name: $UWPWeatherService

Password: ●●

[✓] Save password

Destination URL: http://uwpweatherservice.azurewebsites.net

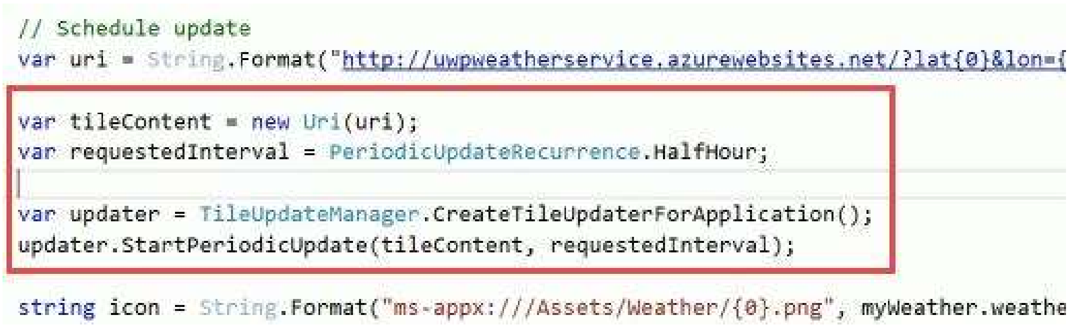
If that succeeds, click on “Publish,” and once publishing is complete, go back to the MainPage.xaml.cs for the original UWPWeather app and modify the Button_Click() method



```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    var position = await LocationManager.GetPosition();

    var lat = position.Coordinate.Latitude;
    var lon = position.Coordinate.Longitude;

    RootObject myWeather =
        await OpenWeatherMapProxy.GetWeather(
            lat,
            lon);

    // Schedule update
    var uri = String.Format("
    http://uwpweatherservice.azurewebsites.net/?lat={0}&lon={1}", lat, lon);
```

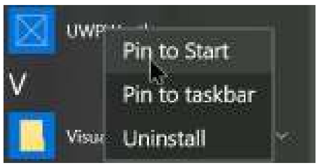
Note that the URL will be unique to the one you happen to have deployed to. Leave the code as-is below this point.

And then reference the code we mentioned at the outset into this method as follows

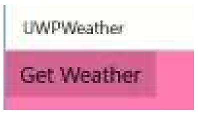


```
// Schedule update
var uri = String.Format("http://uwpweatherservice.azurewebsites.net/?lat{0}&lon={

var tileContent = new Uri(uri);
var requestedInterval = PeriodicUpdateRecurrence.HalfHour;

var updater = TileUpdateManager.CreateTileUpdaterForApplication();
updater.StartPeriodicUpdate(tileContent, requestedInterval);

string icon = String.Format("ms-appx:///Assets/Weather/{0}.png", myWeather.weathe
```

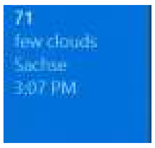
Now, to see the efforts pay off, find and pin the UWPWeatherApp



And run the program locally, clicking on “Get Weather”



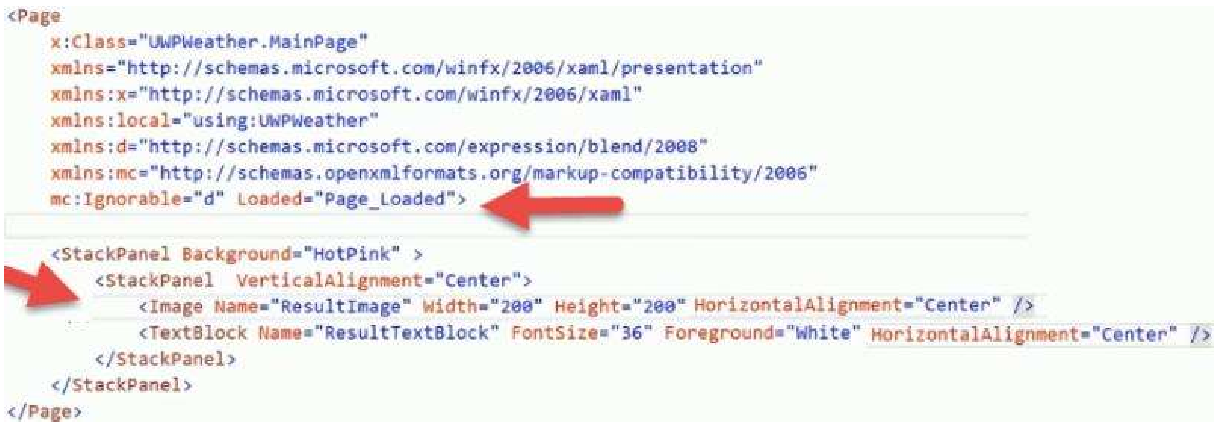
And after a short period of time the remote web service should update your local tile with the weather information you input



UWP-062 - UWP Weather - Finishing Touches

Let’s finalize this app by putting on some finishing touches such as removing the “Get weather” button, centering elements and adding some exception handling.

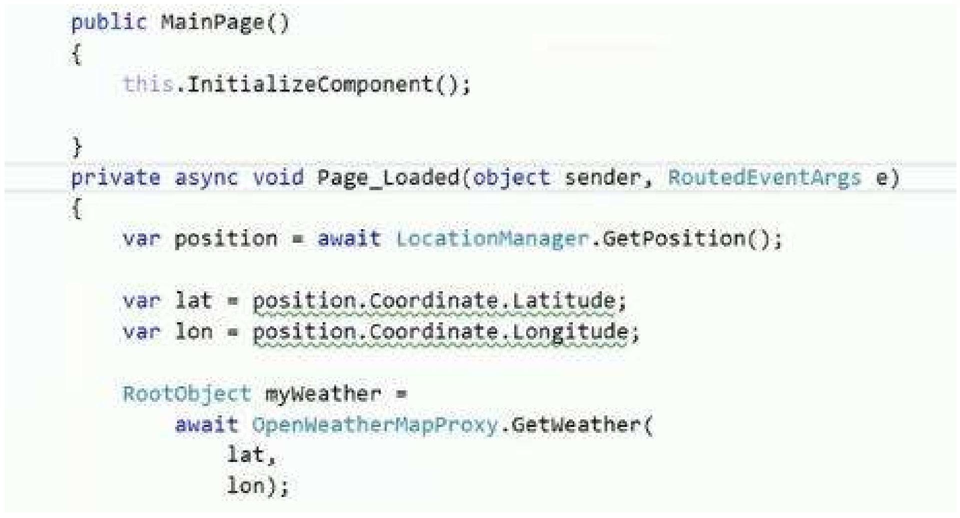
Step 1: Modify MainPage.xaml as follows



```
<Page
    x:Class="UWPWeather.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:UWPWeather"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" Loaded="Page_Loaded">

    <StackPanel Background="HotPink" >
        <StackPanel  VerticalAlignment="Center">
            <Image Name="ResultImage" Width="200" Height="200" HorizontalAlignment="Center" />
            <TextBlock Name="ResultTextBlock" FontSize="36" Foreground="White" HorizontalAlignment="Center" />
        </StackPanel>
    </StackPanel>
</Page>
```

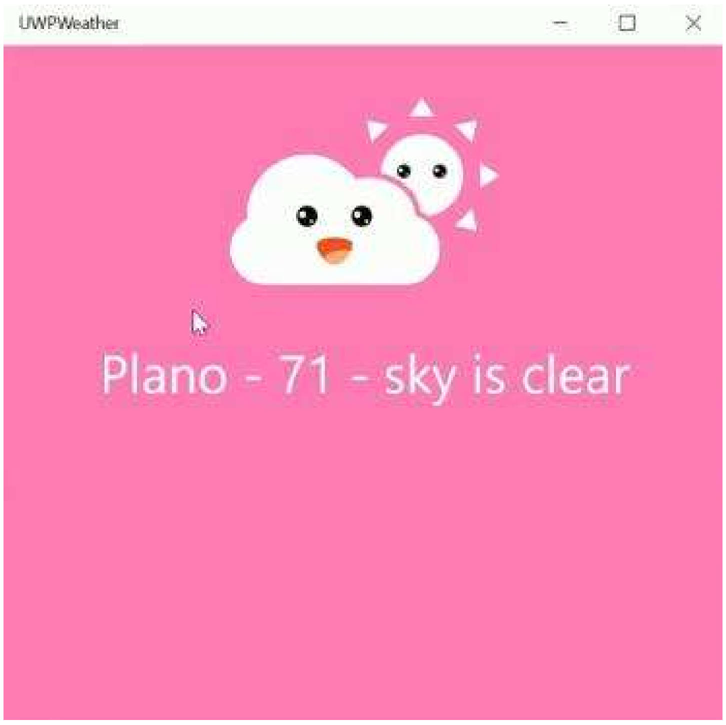
Step 2: In MainPage.xaml.cs take the entire code block within the Button_Click() method and move it to

the Page_Loaded() method (partially displayed below)

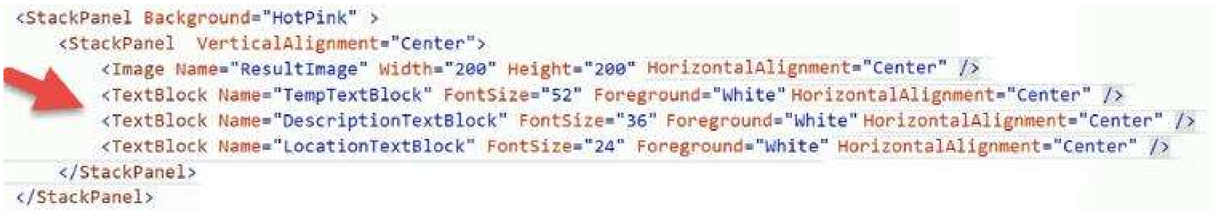


```
public MainPage()
{
    this.InitializeComponent();

}
private async void Page_Loaded(object sender, RoutedEventArgs e)
{
    var position = await LocationManager.GetPosition();

    var lat = position.Coordinate.Latitude;
    var lon = position.Coordinate.Longitude;

    RootObject myWeather =
        await OpenWeatherMapProxy.GetWeather(
            lat,
            lon);
```

And when you run the program it should look like this

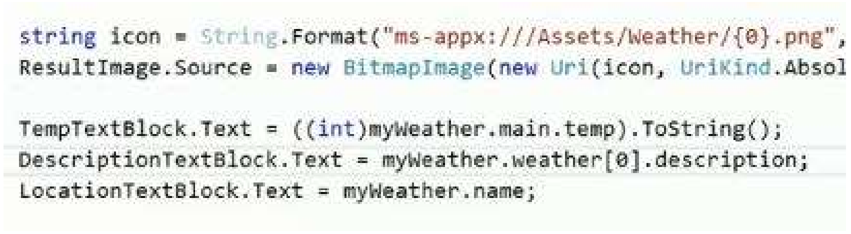


Step 3: Ideally, the information for each item displayed should be on its own line, so make the following changes to the StackPanel



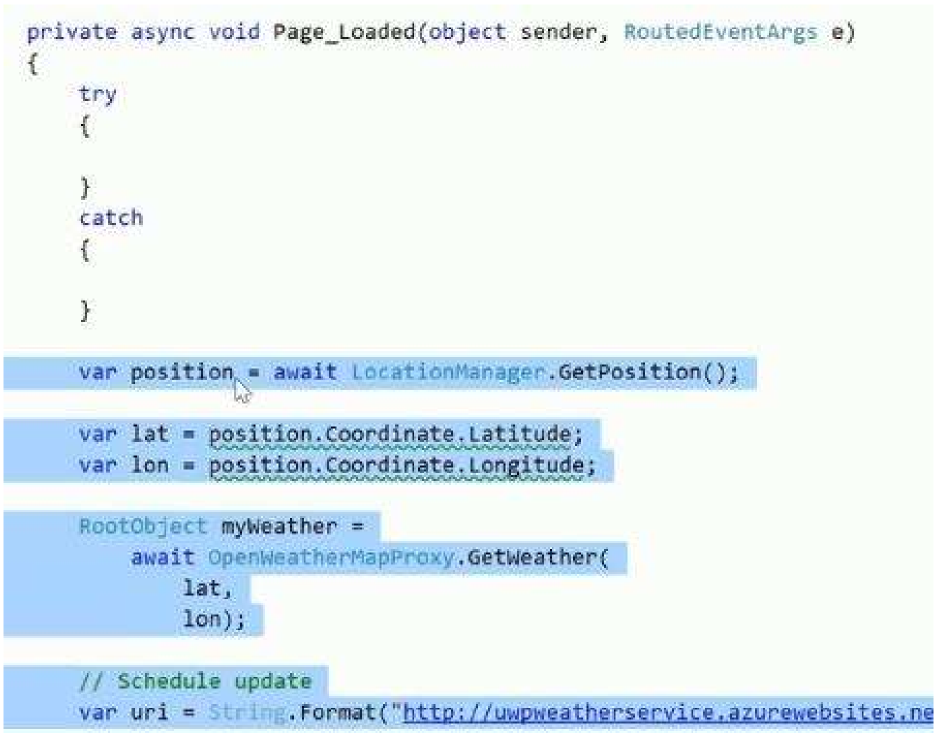
```
<StackPanel Background="HotPink" >
    <StackPanel  VerticalAlignment="Center">
        <Image Name="ResultImage" Width="200" Height="200" HorizontalAlignment="Center" />
        <TextBlock Name="TempTextBlock" FontSize="52" Foreground="White" HorizontalAlignment="Center" />
        <TextBlock Name="DescriptionTextBlock" FontSize="36" Foreground="White" HorizontalAlignment="Center" />
        <TextBlock Name="LocationTextBlock" FontSize="24" Foreground="White" HorizontalAlignment="Center" />
    </StackPanel>
</StackPanel>
```

And in MainPage.xaml.cs reference those changes by modifying the original text being displayed



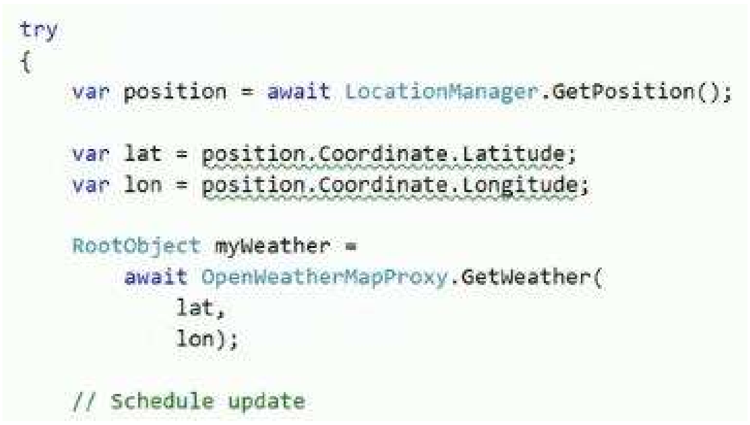
```
string icon = String.Format("ms-appx:///Assets/Weather/{0}.png",
ResultImage.Source = new BitmapImage(new Uri(icon, UriKind.Absol

TempTextBlock.Text = ((int)myWeather.main.temp).ToString();
DescriptionTextBlock.Text = myWeather.weather[0].description;
LocationTextBlock.Text = myWeather.name;
```

Step 4: Now, to include structured exception handling let's handle a try/catch block of code. This is important whenever you have a state in your code that you can't control (such as, an unresponsive server).



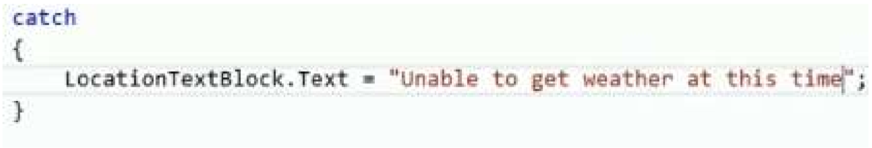
```
private async void Page_Loaded(object sender, RoutedEventArgs e)
{
    try
    {

    }
    catch
    {

    }

    var position = await LocationManager.GetPosition();

    var lat = position.Coordinate.Latitude;
    var lon = position.Coordinate.Longitude;

    RootObject myWeather =
        await OpenWeatherMapProxy.GetWeather(
            lat,
            lon);

    // Schedule update
    var uri = String.Format("http://uwpweatherservice.azurewebsites.ne
```

Normally you would only wrap try/catch around the block of code that might fail, but in this case if any of it fails it will result in a crash of the application, so let’s simply put the entire code block in Page_Loaded() within a try statement (partially represented below)



```
try
{
    var position = await LocationManager.GetPosition();

    var lat = position.Coordinate.Latitude;
    var lon = position.Coordinate.Longitude;

    RootObject myWeather =
        await OpenWeatherMapProxy.GetWeather(
            lat,
            lon);

    // Schedule update
```

And if an exception occurs, simply out put the error that was caught like so



```
catch
{
    LocationTextBlock.Text = "Unable to get weather at this time";
}
```

That pretty much completes the entirety of this weather app. You can definitely make a few tweaks to improve it if you wish. Here’s an optional challenge you can take on if I you so choose: have it so that along with the current weather conditions, you were able to see the forecast for the next seven days (hint: refer to the OpenWeatherMap API for a way to accomplish this).