Name: Olexandr Goloshchapov Date: 8/3/2020

Lab 6 – Model View View Model (MVVM) Pattern

Objectives

Part 1 – Setting up the Sunrise Sunset Project

Part 2 – Getting the Data

Part 3 – Implementing the MVVM Pattern

Reflection

Background/Scenario

This lab, you will be implementing the Model View View Model (or MVVM) Pattern with a graphical interface programed in Windows Presentation Foundation (or WPF) to query a web endpoint and display the results.

The application presents the user with the option to enter a Longitude and a Latitude GPS position and it will query the Sunrise Sunset API and will return the various times of sunrise and sunset information after formatting.

The MVVM pattern is a very useful pattern due to the fact it separates the core business logic completely from the user interface code allowing for separation of concerns and increased testability. MVVM leverages WPF's data binding features allowing user interface developers and backend developers to focus on their own parts of the project.

Required Resources

- Visual Studio 2017
- Internet Connection

Part 1: Setting up the Sunrise Sunset Project

The starter code for this lab is just the user interface. Because this is not a GUI focused class, the user interface is completely implemented. This doesn't mean you have to use it as is. If you want to, feel free to redesign and style the user interface to your own liking.

Step 1: Download the lab starter code.

Download the MVVMStarterCode.zip from Canvas. Ensure it builds and run it. The user interface should hopefully load correctly.

Step 2: Identify both Views.

Two Views have been implemented. The MainWindow.xaml and the SmallWindow.xaml Views. The MainWindow View automatically launches when the project starts. The SmallWindow View is only launched when the "small" command line parameter is passed when launching the application.

Where is the code that determines which View to load located?

In App.xaml.cs OnStartup method. The Uri constructor, by default,

creates a new instance MainWindow.xaml file based on MainWindow.xaml file.

Step 3: Launch the application in "small" mode.

Open the project's settings page and click on the Debug options. Under the Command Line Arguments section, type in "small" without the quotations. Save and run the program. You should see that the program has now launched the SmallWindow View instead of the MainWindow View.

What does typing things in the Command Line Arguments box do?

Typing things is passing arguments into the OnStartup method.

Typing thing is also an alternative to typing arguments manually in the command line.

Remove small from the arguments box and leave it blank for now so we can use the MainWindow view.

Part 2: Getting the Data

This lab, you will be working with real live data accessed from a web endpoint.

Step 1: Explore the API

The data that we will be using in the lab comes from:

https://sunrise-sunset.org/api

Visit the page for the API's documentation. The documentation lists the API base address and parameters.

The base address is:

https://api.sunrise-sunset.org/json

and there are several parameters to pass in. For example, lat, lng, etc…

When working with REST APIs, you pass parameters by concatenating the base address with the parameters separated by a "?" and separating each additional parameter with a "&". Parameters and their data are separated by "=". For example:

https://api.sunrise-sunset.org/json?lat=36.7201600&lng=-4.4203400&date=2019-07-29&formatted=0

What are the parameters and their values passed in for the example above?

Parameter	Value
lat	36.7201600
lng	4.4203400
date	2019-07-29
formatted	0

Step 2: Find better GPS coordinates.

Open Google Maps and find OIT's campus on the map. If you click on the building on the map, you should see a popup that shows the address of what you clicked on with the corresponding GPS

coordinate. Take note of OIT's GPS coordinates and replace the sample's GPS coordinates with OIT's. Replace the date in the sample's to today's date.

What is the new address with the updated data?

https://api.sunrise-sunset.org/json?lat=45.3217219&lng=-122.7686344&date=2020-07-31&formatted=0

Step 3: View the data.

Paste the new address from the prior step into a new web browser window. If your parameters were valid, you should be presented with data in your browser window.

What format is the data in?

JSON

Keep the window open for now, we'll use it later.

Step 4: Create a new Library project.

Create a new Library project and name it SunriseSunsetLib. Rename the auto generated Class1 file and class to SunriseSunsetApi and make it public.

Create a public function, CallApi that takes in a double longitude, latitude, and a DateTime date. For now, have the return type of the function void. We'll update that later.

Create a private const string member variable for the class and paste your API address you created from Step 2.

Replace the hard coded parameter data from your API address with the string format markers: {0} {1} and {2}.

Step 5: Create a new Test Project.

Add a new Test Project to your solution. We'll use this to test your CallApi function. You can use whichever testing framework you wish.

Create a new test method and instantiate an instance of your SunriseSunSetApi class and call the CallApi function. Use the parameters from Step 2 as the test parameters to pass into the CallApi function.

Run the test, it should pass since we haven't Asserted anything yet. We'll come back to this.

Step 6: Call the API

In the CallApi function insantiate a new instance of the WebClient class. (You will have to add a new using statement at the top of your file.) Because WebClient implements IDisposable, wrap it inside a using block.

Using string.Format() pass in your const string with your API address and pass in the longitude, latitude, and date from the function parameters.

Call the DownloadString() function on WebClient and pass in your formatted string. Set the return from the function to a local string variable outside the using block.

Step 7: Parse the API data.

After calling the API, we got string data back. But to use the data, we will need to deserialize the data first. Open the NuGet package manager and install the Json.NET package for your library.

Outside of the WebClient using block, call the JsonConvert.DeserializeObject() function and pass in your string data you got back. Notice the DeserializeObject returns an object type. Working with an object type isn't ideal, but luckily, the DeserializeObject<T>() has a generic overload.

If we create a new class with properties matching all of our expected API data, Json.Net can automatically populate our class for us.

Step 8: Creating the SunriseSunsetData class.

We could create the class manually, but why do that web Visual Studio can do it for us?

Going back to your web browser window you had open with the results of your API call, copy all the text.

Back in Visual Studio, create a new .cs file and name it SunriseSunsetData.cs.

Delete the auto generated class. Then go to the Edit menu, click on Paste Special, and select Paste JSON as Classes.

Visual Studio will automatically parse the sample JSON data and automatically create classes for you!

Rename the default named RootObject class to SunriseSunsetResult and rename the Results class to SunriseSunsetData.

Step 9: Finishing the CallApi function

Back to the DeserializeObject function, put the SunriseSunsetResult class type into the <> brackets so the function should now return SunriseSunsetResult instead of object.

Check if the status property is "OK"

If it is, return result, if not, return null. Change the return type of the function to SunriseSunsetData.

Step 10: Test the CallApi function.

Update the test to now store the result of the CallApi function in a local SunriseSunsetData variable.

Assert the sunrise and sunset times match the expected you got from the text you got in the web browser window. Do not move on until your test is green.

Part 3: Implementing the MVVM Pattern

Now that we are able to get our data, we can now start on the main application.

Step 1: Creating the Model.

Create a new class MainModel.cs in the WPF project. Instantiate a private member variable of SunriseSunsetApi. Create a function GetData that takes in a double longitude, latitude, and date and return the CallApi function.

Step 2: Creating the View Model.

Create a new class MainVM.cs in the WPF project.

Create a public string property Sunrise.

Create two public double properties and name them Longitude and Latitude.

Create a public DateTime property and name it Date.

Note: Create properties and not fields. Properties are the ones with the get and set methods.

Create a default constructor and hardcode the Longitude, Latitude, and Date properties to the same values as we have been using.

> Tip: To help you create properties, you can type "propfull" and immediately press tab twice. Visual Studio will automatically type out the full property syntax for you.

Step 3: Implement INotifyPropertyChanged.

Implement the INotifyPropertyChanged interface on your VM class. When you implement the interface, it adds an Event PropertyChanged to your class. The event is what lets WPF know any data has been modified and needs to be updated on the UI (and vice versa). We need to call the event in code.

Create a new function NotifyPropertyChanged that takes in a string and name it propertyName. The propertyName parameter will be the name of the Property that has been modified.

Inside the function, call the PropertyChanged event while checking for null and pass in this as the sender and the propertyName as parameter for the new PropertyChangedEventArgs.

You can leave the function as is, since it works now, but to make things easier, we can decorate the propertyName string with an attribute CallerMemberName which will auto fill the propertyName string with the name of who called the function.

It should all look like this:

```
public event PropertyChangedEventHandler PropertyChanged;
private void NotifyPropertyChanged([CallerMemberName]string
propertyname = "")
{
    PropertyChanged?.Invoke(this,
        new PropertyChangedEventArgs(propertyname));
}
```

Now, call the NotifyPropertyChanged() function in the set method of your Sunrise property.

Step 4: Create the WPF Bindings

Open MainWindow.xaml to open the XAML Designer.

Click the Latitude TextBox and it should highlight the corresponding line of XAML.

Add the Text property and set it to "{Binding Latitude}"

It should look like this:

```
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Latitude}" />
```

Do the same for Longitude, Sunset, and Date.

The only difference is that Date will be SelectedDate instead of Text because it is DatePicker and not a TextBox.

Step 5: Add the DataContext.

At the very top of the XAML file right before the first <Grid> tag, set the window's DataContext to a new instance of the MainVM class.

```
<Window.DataContext>
    <local:MainVM />
</Window.DataContext>
```

Save and build the project. If everything was set up correctly, you should now see the default Longitude, Latitude, and Date prefilled out in the textboxes.

Step 6: Implement the Calculate button. Part 1

Before we can connect the button to anything, we need to create a new ICommand class that we can bind the button to. Normally you would create a new class for every command, but we'll create a generic class that we can reuse for all of our commands to make things easier.

Create a new class called Relay command and implement the ICommand interface.

Create a private member of type Action.

Create a constructor that takes in an Action as a parameter and set the private Action as the value of the parameter.

Have CanExecute return true.

Inside the Execute function, call the private action function.

Step 7: Implement the Calculate button. Part 2

In your VM, create a public property of type ICommand and name it CalculateCommand.

In the constructor, instantiate the CalculateCommand property to a new instance of RelayCommand.

Create a new function (or lambda expression) that calls the GetData function. Pass in the Latitude, Longitude, and Date properties to the GetData function. Set the Sunrise property to the sunrise value (call ToLongTimeString()) of the result of the GetData function.

Pass the function name into the constructor of RelayCommand.

Step 8: Bind the Calculate button to the CalculateCommand.

Back in the XAML, add the Command property to the button and set it to "{Binding CalculateCommand}" value.

```
<Button Content="Calculate" Command="{Binding CalculateCommand}" .../>
```

Step 9: Run the application.

If everything was set up correctly, the Calculate button should set the Sunrise textbox to the time of the sunrise.

Step 10: Bind everything else.

Set up the rest of the bindings for all the other text boxes just like you did with Sunrise.

Make sure all result text boxes are in the HH:MM:SS AMPM format. (That includes Day Length!)

Hint: Day Length is in seconds.

Implement the Save button to save to a file on disk the output of the sunrise sunset api.

Part 4: Implementing the MVVM Pattern Continued

Now that we got the MainWindow to display all of the data, do the same for SmallWindow. Set up all the bindings for SmallWindow just like you did for MainWindow. You do not need to create a new VM or modify any of the code at all. The only thing you will need to change is SmallWindow's XAML.

Run the program in "small" mode and verify everything works.

Reflection

1. Did you have any trouble with WPF or any other parts of the lab?

I had trouble with correctly implementing the Calculate command within the Main VM constructor, but had a good grasp on it after the live lab on Saturday. Some of the date formatting can be really finicky to consistently set across the entire application.

2. The CallApi() function breaks all kinds of SOLID principles. What kinds of things would you refactor to better follow SOLID design?

I would store the api calls within the SunriseSunsetResult class and use the result within SunriseSunsetApi constructor. This way, only one instance of SunriseSunsetApi is needed within Main VM, and the data can be called as needed from within the new object.