Lab 2: Advanced UWP concepts

Throughout the labs, we’ll create the MovieWatchr Universal Windows App: an app to track the movies you want to watch and log what you thought of the ones you have watched. In this lab exercise, we’ll connect the app from lab 1 to the TMDb.org API to retrieve relevant movie information in an app service and we’ll add a background task to update the app’s Live Tile.

At any time during the lab, you can take a look at the solution in the **H02\End** folder to see a sample implementation of everything we’re going to be doing in this lab.

# Preparation

Make sure you have the following installed on your machine:

* Windows 10 Technical Preview (build 10074)
  + Note that installing Windows in a virtual machine will prohibit you from using the Emulator and will limit you to testing on actual phone/ARM hardware
* Visual Studio 2015 RC with the Windows 10 Tools

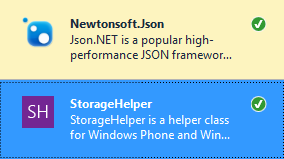
## Starting with the base solution

As we need a few minor tweaks to our **H01\End** solution, you can use the solution in the **H02\Start** folder as a starting point for this lab. The following paragraph will describe how you can get to that exact same state starting from the **H01\End** or your own end solution from lab 1.

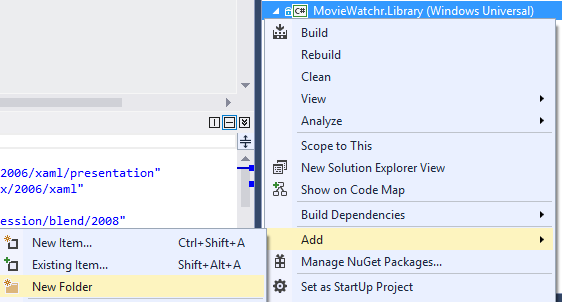
## Updating the lab 1 end solution

First, let’s move the Models directory under the **MovieWatchr** project to the **MovieWatchr.Library** project by clicking and dragging the folder while holding the Shift key on the keyboard. This is necessary, so we can use the Movie model in the projects that we will add to the solution later on.

Also add the **Newtonsoft.Json** and **StorageHelper** NuGet packages to the **MovieWatchr.Library** project, as we’ll be using them later on in the implementation.

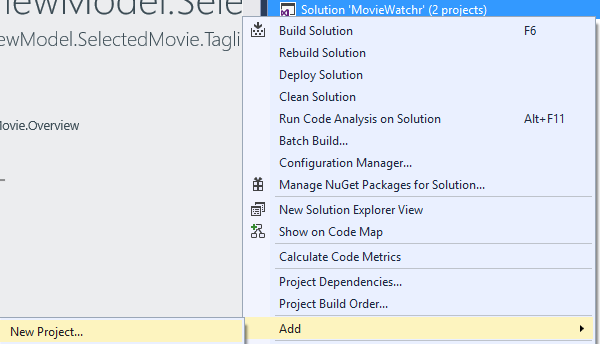


Now, let’s add a folder called **Services** to the **MovieWatchr.Library** project by right-clicking the project and selecting **Add 🡪 New Folder**



In this folder, create two new classes, called **MovieWatchrAppService.cs** and **TileService.cs**. The actual implementation will follow later.

Next, we’ll add two projects: one to hold our app service and one to hold our background tasks. Right-click on the solution and select **Add 🡪 New Project…**



In the **Add New Project** dialog, select **Visual C# 🡪 Windows Universal 🡪 Windows Runtime Component (Windows Universal) Visual C#** and add two projects, named **MovieWatchr.Service** and **MovieWatchr.Tasks**.

Under the **MovieWatchr.Service** project, add a new class called **TMDbLookupTask.cs**. Under the **MovieWatchr.Tasks** project, add two classes called **MetadataCompletionTask.cs** and **TileUpdaterTask.cs**.

Lastly, add a reference to the **MovieWatchr.Library** project to both the **MovieWatchr.Service** and **MovieWatchr.Tasks** projects and also add the **Newtonsoft.Json** NuGet package to just the **MovieWatchr.Service** project.

# Getting an API key from themoviedb.org

We’ll be using themoviedb.org to fetch actual movie data in our app, so start by signing up for an account at <https://www.themoviedb.org/account/signup>. After you’ve done so, go to your account page and request an API key from [https://www.themoviedb.org/account/<username>/api](https://www.themoviedb.org/account/%3cusername%3e/api).

# Creating the App Service

We’re going to deploy an app service as part of our app, so multiple apps can use our functionality to look up movies on themoviedb.org. To do so, we need to specify the app service in our main app’s manifest under **MovieWatchr 🡪 Package.appxmanifest**. Find the node that says </uap:VisualElements> and immediately below, add the following:

<Extensions>

<uap:Extension Category="windows.appService"

EntryPoint="MovieWatchr.Service.TMDbLookupTask">

<uap:AppService Name="moviewatchr-tmdblookup" />

</uap:Extension>

</Extensions>

This will register our app service with the app and will listen to the **moviewatchr-tmdblookup** name. Next, we need to implement the actual app service in the **TMDbLookupTask.cs** class we added to the **MovieWatchr.Service** project.

An app service is a special kind of background task on the Universal Windows Platform, so start by implementing the **IBackgroundTask** interface and its methods. This will add a Run method to the class. We need to define a BackgroundTaskDeferral in our class, so we can keep a chatty connection open to the app service:

private static BackgroundTaskDeferral \_serviceDeferral;

Inside the Run method of our background task, we need to get the actual deferral from the taskInstance we receive as a parameter:

\_serviceDeferral = taskInstance.GetDeferral();

To start listening to requests in the app service, hook up the AppServiceConnection.RequestReceived event:

var appService = taskInstance.TriggerDetails as AppServiceTriggerDetails;

if (appService != null && appService.Name == "moviewatchr-tmdblookup")

appService.AppServiceConnection.RequestReceived += AppServiceConnection\_RequestReceived;

private async void AppServiceConnection\_RequestReceived(AppServiceConnection sender, AppServiceRequestReceivedEventArgs args)

{ }

For the actual implementation when we receive a request, it’s up to us to determine a communication protocol. The app service request receives a request object in the arguments, which holds a **ValueSet**. This is a simple key/value dictionary of serializable objects.

In our app, we’ll use the **Title** key in our ValueSet to send over the movie title we want to look up and will send back a different ValueSet, containing an **Error** key to indicate whether or not an error occurred and a **MovieJson** key that will hold the actual JSON returned by themoviedb.org. An implementation could look like:

private static HttpClient \_httpClient;

private static string \_tmdbApiKey = "TMDB\_API\_KEY"; // Swap out with your TMDb API key

private static string \_tmdbSearchUri = "http://api.themoviedb.org/3/search/movie?api\_key={0}&query={1}";

private static string \_tmdbMovieUri = "http://api.themoviedb.org/3/movie/{0}?api\_key={1}";

private async void AppServiceConnection\_RequestReceived(AppServiceConnection sender, AppServiceRequestReceivedEventArgs args)

{

var message = args.Request.Message;

var title = message["Title"] as string;

if (!string.IsNullOrWhiteSpace(title))

{

var messageDeferral = args.GetDeferral();

var returnMessage = new ValueSet();

try

{

var searchResult = await \_httpClient.GetStringAsync(new Uri(string.Format(\_tmdbSearchUri, \_tmdbApiKey, title)));

var movieId = JObject.Parse(searchResult)["results"][0]["id"];

var movieResult = await \_httpClient.GetStringAsync(new Uri(string.Format(\_tmdbMovieUri, movieId, \_tmdbApiKey)));

// Build the return message with the movie JSON

returnMessage.Add("Error", false);

returnMessage.Add("MovieJson", movieResult);

}

catch (Exception ex)

{

returnMessage.Add("Error", true);

}

finally

{

await args.Request.SendResponseAsync(returnMessage);

messageDeferral.Complete();

}

}

}

# Using the app service from our app

To use the app service, we’ll create a helper service class to simplify calling the app service. We’ll do this in the **MovieWatchr.Library 🡪 Services 🡪 MovieWatchrAppService.cs** class. We’ll be adding two methods and the variable that will hold our connection instance:

public static AppServiceConnection AppServiceConnection = null;

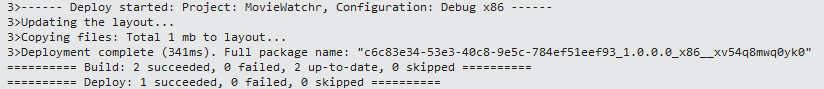
public static async Task EnsureConnectionToAppServiceAsync() { }

public static async Task<AppServiceResponse> SendMessageAsync(string message) { }

Inside the **EnsureConnectionToAppServiceAsync** method, we’ll be creating the connection to the app service (as we are not fully in control of its lifecycle, we need to ensure the connection before sending messages), using its name (moviewatchr-tmdblookup) and package family name. The package family name can be retrieved from the Windows Store, once the app name has been reserved, to ensure a unique name across the Store.

During the development, you can get the package family name by deploying the app and looking at the Visual Studio output or calling the Windows.ApplicationModel.Package.Current.Id.FamilyName API.

NOTE: By the PFN that is displayed in the Build Output window (below) holds some extra information. In the case below, the actual package family name is c6c83e34-53e3-40c8-9e5c-784ef51eef93\_xv54q8mwq0yk0.



After creating the **AppServiceConnection** instance, call its **OpenAsync()** method to open the connection.

To send the actual message, we need to create a **ValueSet** that will hold our message and use the **SendMessageAsync** method of the **AppServiceConnection** instance to send the actual message.

## Calling the app service from the UI

Not that everything is in place, the final piece is connecting the app service to the add button of our UI. Also, we want to store the movie collection into persistent storage, so movies we add are saved between app launches. To make this easy, we’ll be using the **StorageHelper** NuGet package. This is a simple wrapper around the Windows storage APIs and its source can be viewed on GitHub (<https://github.com/rajenki/storagehelper>). To use it, simply call the following in the **LoadMovies** method of our **MainViewModel.cs**:

Movies = await Storage.LoadAsync<ObservableCollection<Movie>>("Movies");

if (Movies == null)

Movies = new ObservableCollection<Movie>();

This will load the stored movies collection if it exists or otherwise initialize a new, empty collection in our ViewModel.

Finally, in the **AddMovie** method of our **MainViewModel.cs** class, we can call the app service through our **MovieWatchrAppService** wrapper:

await MovieWatchrAppService.EnsureConnectionToAppServiceAsync();

var response = await MovieWatchrAppService.SendMessageAsync(title);

if (response.Status == AppServiceResponseStatus.Success)

{

if ((bool)response.Message["Error"])

Movies.Add(new Movie { Title = title });

else

{

var movieJson = (string)response.Message["MovieJson"];

var movie = JsonConvert.DeserializeObject<Movie>(movieJson);

Movies.Add(movie);

}

}

Storage.SaveAsync("Movies", Movies);

All that remains now is to build our solution and test out adding new movies, which will get their metadata from themoviedb.org

# Adding background tasks to update the metadata and tiles

To support an offline scenario, we’ll be adding a background task that will periodically check if there have been movies added without themoviedb.org metadata. This background task will then call our app service to add the missing metadata in the background, so the user will see the data the next time they open the app.

We’ll also add a background task to periodically update the app’s Live Tile with the latest movie that was added, so the live tile displays fresh information. This functionality can of course be expanded to provide even more meaningful and useful information to the user.

The first step is defining the two background tasks in our **Package.appxmanifest**:

<Extension Category="windows.backgroundTasks"

EntryPoint="MovieWatchr.Tasks.TileUpdaterTask">

<BackgroundTasks>

<Task Type="timer" />

</BackgroundTasks>

</Extension>

<Extension Category="windows.backgroundTasks"

EntryPoint="MovieWatchr.Tasks.MetadataCompletionTask">

<BackgroundTasks>

<Task Type="timer" />

</BackgroundTasks>

</Extension>

## Implementing the metadata completion background task

Open the **MovieWatchr.Tasks** 🡪 **MetadataCompletionTask.cs** class and implement the **IBackgroundTask** interface, as this will be our background tasks’ entry point. In the Run method, just request the task deferral (as we’ll be calling an asynchronous method) and call the following method (which we’ll implement next):

await MovieWatchrAppService.UpdateMoviesMetadataAsync();

We’ll be needing a few extra methods to register the task with the Windows Operating System, as the task will run independently from our main app:

public static IAsyncOperation<bool> Register()

{

return RegisterInternal().AsAsyncOperation();

}

private async static Task<bool> RegisterInternal()

{

if (!IsTaskRegistered())

{

await BackgroundExecutionManager.RequestAccessAsync();

Unregister();

var condition = new SystemCondition(SystemConditionType.InternetAvailable);

var builder = new BackgroundTaskBuilder { Name = TaskName, TaskEntryPoint = typeof(MetadataCompletionTask).FullName };

builder.SetTrigger(new TimeTrigger(30, false));

builder.AddCondition(condition);

builder.Register();

return true;

}

return false;

}

public static void Unregister()

{

var entry = BackgroundTaskRegistration.AllTasks.FirstOrDefault(t => t.Value.Name == TaskName);

if (entry.Value != null)

entry.Value.Unregister(true);

}

public static bool IsTaskRegistered()

{

return BackgroundTaskRegistration.AllTasks.Any(t => t.Value.Name == TaskName);

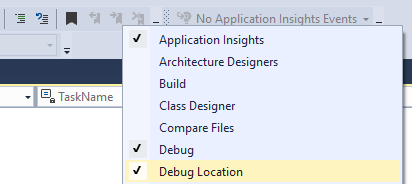
}

The most important part is inside the **RegisterInternal** method, with the creation of the **SystemCondition**, that specifies our task should only run when an active internet connection is available and the actual creation and registration of the background task with the **BackgroundTaskBuilder**. The other methods just make its usage a bit easier and robust. To register the background task, simply call its **Register** method when the app launches (for instance in the **MainPage\_Loaded** method:

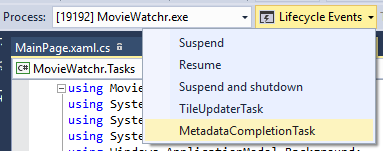
if (!MetadataCompletionTask.IsTaskRegistered()) MetadataCompletionTask.Register();

## Testing our background task

To test our newly created background task, we either deploy the app, run it and wait 30 minutes (as previously defined in the **SetTrigger** method), but that’s quite inefficient during development. The easier method is to run the app using Visual Studio’s **F5/Start Debugging** and using Visual Studio’s capabilities to manually trigger and registered background task of the app. Once the app is running with the debugger attached, right-click on Visual Studio’s toolbar and select the **Debug Location** toolbar.



On that toolbar, you get a drop-down menu with the app’s **Lifecycle Events**, but also the registered background tasks, which you can run by clicking on:



## Implementing the tile updater background task

Similar to the metadata completion background task, we’ll add a background task to refresh the app’s tile. As we’ve already registered the task in our app’s manifest, open up the **MovieWatchr.Tasks** **🡪 TileUpdaterTask.cs** file and implement the **IBackgroundTask** interface.

We need the **Register**, **RegisterInternal**, **Unregister** and **IsTaskRegistered** methods that we also used in our **MetadataCompletionTask**, with a different **RegisterInternal** implementation, as we don’t need the internet availability condition:

private async static Task<bool> RegisterInternal()

{

if (!IsTaskRegistered())

{

await BackgroundExecutionManager.RequestAccessAsync();

var builder = new BackgroundTaskBuilder { Name = TaskName, TaskEntryPoint = typeof(TileUpdaterTask).FullName };

builder.SetTrigger(new TimeTrigger(30, false));

builder.Register();

return true;

}

return false;

}

To update the app’s tile, we need to send an XML snippet to it, containing the new layout. The actual APIs to send the tile update are as follows:

XmlDocument tileDOM = new XmlDocument();

tileDOM.LoadXml(tileXmlString);

TileNotification tile = new TileNotification(tileDOM);

TileUpdateManager.CreateTileUpdaterForApplication().Update(tile);

The way you want to create the actual XML is up to you, but take a look at the **H02/End** solution for an example implementation. You can create a method called **GetLastMovieWithMetadataAsync** in the **MovieWatchr.Library 🡪 Services 🡪 TileService.cs** class to be able to fetch the last movie added that contains metadata:

public async static Task<Movie> GetLastMovieWithMetadataAsync()

{

var movies = await Storage.LoadAsync<ObservableCollection<Movie>>("Movies");

if (movies == null) return null;

return movies.Reverse().FirstOrDefault<Movie>(m => m.Id != 0);

}

After that implementation, your tile should now update every 30 minutes with the latest movie added.