Windows 10

Hands-on lab

App Services

September 2015

Microsoft

CONTENTS

OVERVIEW **3**

EXERCISE 1: CREATE AND REGISTER THE APP SERVICE **4**

Task 1 – Create the EmployeeLookupService project 4

Task 2 – Create the Windows Runtime Component 5

Task 3 – Implement the IBackgroundTask interface 6

Task 4 – Handle the app service connection 7

Task 5 – Register the app service 10

EXERCISE 2: CALL THE APP SERVICE FROM ANOTHER APP **11**

Task 1 – Create a blank Universal Windows app 12

Task 2 – Create the UI and a simple model 13

Task 3 – Open the app service connection 15

Task 4 – Display the results and debug the background task 18

SUMMARY **21**

Overview

App services are headless services that can encapsulate business logic, data, and transactions for other apps without the performance expense of launching another app. They provide functionality similar to Web Services, but in the context of Store Apps. Some of the scenarios where app services shine include pulling data and cached data.

In the case of pulling data, you may wish your app to be able to access or process data from another app without launching the app itself. Headless app services using background tasks are an efficient way to get access to that data.

Location services are a great example of pulling cached data to save resources. You may need your app to access local map data with a minimum of data usage. If another app, such as Bing Maps, has already downloaded and cached the information your app needs, you can use an app service to pull it over without using additional bandwidth.

In this lab, you will create an employee lookup app service that takes one or more employee IDs as input and returns the corresponding employee names. You will then create an app to query the service and display the results.

Objectives

This lab will show you how to:

- Create and register an app service
- Call the app service in the background from another app
- Pass data back to the calling app
- Debug the app service

System requirements

You must have the following to complete this lab:

- Microsoft Windows 10

- Microsoft Visual Studio 2015

Setup

You must perform the following steps to prepare your computer for this lab:

1. Install Microsoft Windows 10.
2. Install Microsoft Visual Studio 2015.

Exercises

This Hands-on lab includes the following exercises:

1. Create and Register the App Service
2. Call the App Service from Another App

Estimated time to complete this lab: **45 to 60 minutes**.

Exercise 1: Create and Register the App Service

In this exercise, you will create a solution for the employee lookup app service. The solution will contain a Windows Runtime Component with the app service and an app derived from the Blank App template in which you will register the app service. We will not build a UI for the app itself, but we will use its package family name later on to allow other apps to target the app service.

Task 1 – Create the EmployeeLookupService project

Create a new solution for the app service project.

1. In a new instance of Visual Studio 2015, choose **File > New> Project** to open the New Project dialog. Navigate to **Installed > Templates > Visual C# > Windows > Universal** and select the **Blank App (Universal Windows)** template.

2. Name the project **EmployeeLookupService**. Save the project to the folder where you store your Hands-on Labs.

3. Set your Solution Configuration to **Debug** and your Solution Platform to **x86**. Select **Local Machine** from the Debug Target dropdown menu.

4. Build and run your app. You will see a blank app window with the frame rate counter enabled by default for debugging.

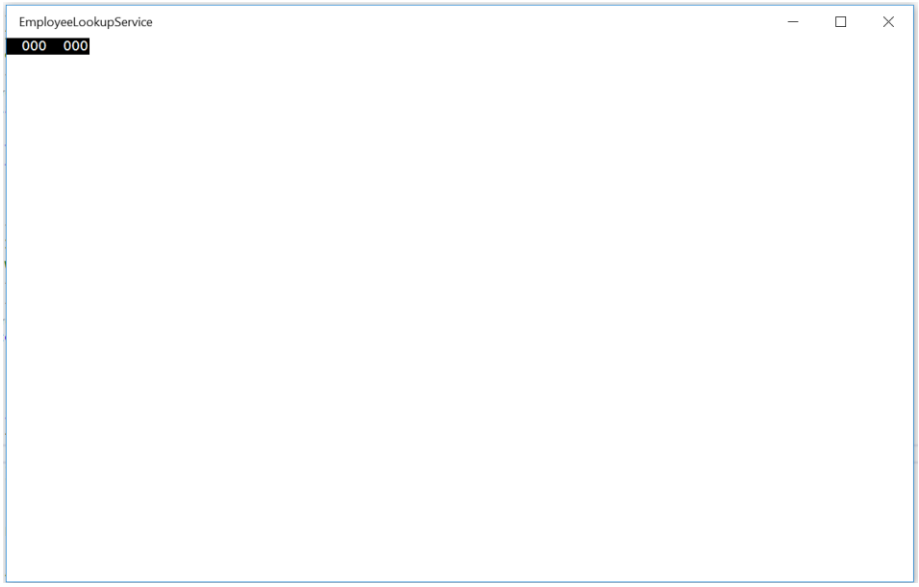

Figure 1
The blank universal app running in Desktop mode.

Note: In this lab, we will use the EmployeeLookupService app as a container for the app service, which will run as a background task. The MainPage view will remain blank.

5. Return to Visual Studio and stop debugging.

Task 2 – Create the Windows Runtime Component

The app service will run as a background task, which needs to be in a project of type Windows Runtime Component. We will get started by creating the Windows Runtime Component.

1. Right-click on the EmployeeLookupService solution and choose **Add > New Project**. Add a project of type **Visual C# > Windows > Universal > Windows Runtime Component (Universal Windows)**. Give it the name **EmployeeLookupService.Background**.

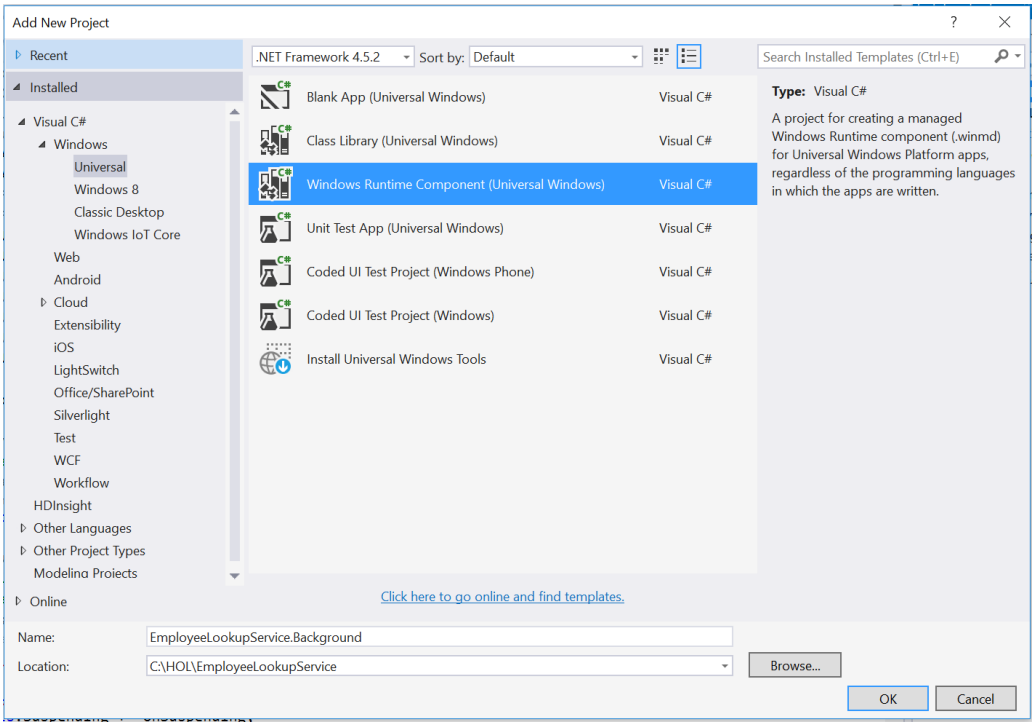

Figure 2

Add the WinRT component for the app service.

2. Right-click on Class1.cs in the EmployeeLookupService.Background project and choose Rename. Give it the name **EmployeeLookup.cs**. If prompted to perform a rename of all references to Class1, choose **Yes**.

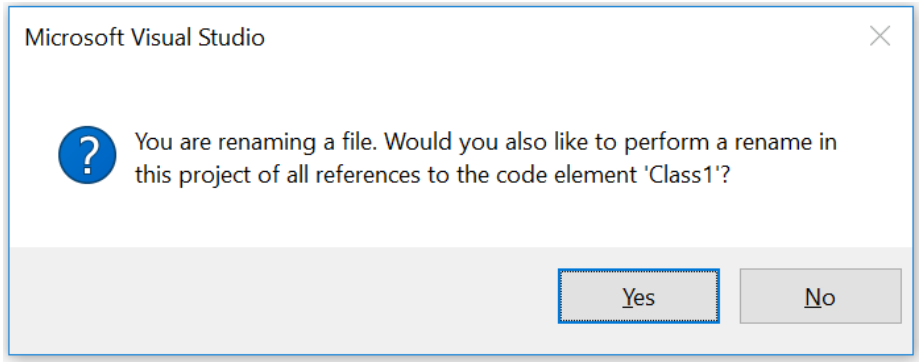

Figure 3

Rename Class1.cs to EmployeeLookup.cs.

Task 3 – Implement the IBackgroundTask interface

Now that your WinRT component is in place, you can implement the basic structure of a background task.

Note: Check out the **Background Tasks** lab for a more in-depth look at background tasks and their structure.

1. Open **EmployeeLookup.cs** and add the **Windows.ApplicationModel.Background** namespace.

```
C#
using Windows.ApplicationModel.Background;
```

2. Implement the IBackgroundTask interface.

```
C#
public sealed class EmployeeLookup : IBackgroundTask
{
    public void Run(IBackgroundTaskInstance taskInstance)
    {
        throw new NotImplementedException();
    }
}
```

3. Create a deferral at the class level and open the deferral in the **Run** method. Subscribe to the **taskInstance.Canceled** event to close the deferral in case the task is cancelled.

```
C#
private BackgroundTaskDeferral deferral;

public void Run(IBackgroundTaskInstance taskInstance)
{
    this.deferral = taskInstance.GetDeferral();

    taskInstance.Canceled += TaskInstance_Canceled;

}

private void TaskInstance_Canceled(IBackgroundTaskInstance sender,
BackgroundTaskCancellationReason reason)
{
    if (this.deferral != null)
    {
        this.deferral.Complete();
    }
}
```

Task 4 – Handle the app service connection

The app service we are creating will take an employee ID as input and return the corresponding employee name. In this task, you will create a simple dictionary with employee lookup information, detect the incoming app service connection, and return a message with the requested information.

> **Note:** The AppServiceConnection is an asynchronous connection to the app service entrypoint opened by the calling app. The calling app can send a message to the app service, which can handle the message and return a response. You will learn how to open the connection in the next exercise. For more on AppServiceConnections, visit https://msdn.microsoft.com/en-us/library/windows/apps/windows.applicationmodel.appservice.appserviceconnection.aspx

1. Create a private dictionary at the class level to hold employee lookup data. We will provide entries for four employees.

C#

```
public sealed class EmployeeLookup : IBackgroundTask
{
    private BackgroundTaskDeferral deferral;

    private Dictionary<string, string> employees =
        new Dictionary<string, string>
                { { "0", "John Smith" }, { "1", "Mary Echo" },
                  { "2", "Jane Doe" }, { "3", "Bob Harvey" } };
```

2. Add the **Windows.ApplicationModel.AppService** namespace to **EmployeeLookup.cs**.

C#

```
using Windows.ApplicationModel.AppService;
```

3. Add an **appServiceConnection** variable at the class level. In the **Run** method, set **appServiceConnection** to the instance of the incoming app service connection found on the task instance trigger details object.

C#

```
public sealed class EmployeeLookup : IBackgroundTask
{
    private BackgroundTaskDeferral deferral;

    private AppServiceConnection appServiceConnection;

    private Dictionary<string, string> employees = new Dictionary<string,
        string> { { "0", "John Smith" }, { "1", "Mary Echo" }, { "2", "Jane
        Doe" }, { "3", "Bob Harvey" } };

    public void Run(IBackgroundTaskInstance taskInstance)
    {
        this.deferral = taskInstance.GetDeferral();
```

```
        taskInstance.Canceled += TaskInstance_Canceled;

        var trigger = taskInstance.TriggerDetails as AppServiceTriggerDetails;
        this.appServiceConnection = trigger.AppServiceConnection;
    }
```

4. Subscribe to the app service connection **RequestReceived** event at the end of the **Run** method. When the request has been handled, we will mark the deferral as complete.

C#

```
    this.appServiceConnection = trigger.AppServiceConnection;
    this.appServiceConnection.RequestReceived +=
        AppServiceConnection_RequestReceived;
}

private void AppServiceConnection_RequestReceived(AppServiceConnection sender,
AppServiceRequestReceivedEventArgs args)
{
    var deferral = args.GetDeferral();

    deferral.Complete();
}
```

5. Add the **Windows.Foundation.Collections** namespace to the **EmployeeLookup** class.

C#

```
using Windows.Foundation.Collections;
```

6. Any app that calls the employee lookup service will need to send a message with the employee ID for lookup. A simple object can be passed to a background task using a ValueSet. In the AppServiceConnection_RequestReceived event handler, receive the incoming message and return a ValueSet in response that contains the employee lookup results. The employee id will be the key for the response, and the employee name will be the value.

C#

```
private async void AppServiceConnection_RequestReceived(AppServiceConnection
sender, AppServiceRequestReceivedEventArgs args)
{
    var deferral = args.GetDeferral();
    var requestMessage = args.Request.Message;
    var responseMessage = new ValueSet();

    foreach (var item in requestMessage)
    {
        if (employees.ContainsKey(item.Value.ToString()))
        {
```

```
            responseMessage.Add(item.Value.ToString(),
                employees[item.Value.ToString()]);
        }
    }

    await args.Request.SendResponseAsync(responseMessage);

    deferral.Complete();
}
```

Note: A ValueSet is a map with a key of type string and a value of type object. Only serializable types can be put into the map. For more on ValueSets, visit https://msdn.microsoft.com/en-us/library/windows/apps/windows.foundation.collections.valueset.aspx

Task 5 – Register the app service

An app service must be registered to be accessible. We will register the EmployeeLookup app service in the associated app and define its entrypoint in the manifest.

1. Open the EmployeeLookupService **package.appxmanifest** in the manifest editor.
2. On the **Declarations** tab, add an **App Service** declaration. Give it the name **EmployeeLookupService** and set the entry point to **EmployeeLookupService.Background.EmployeeLookup**. Save the manifest.

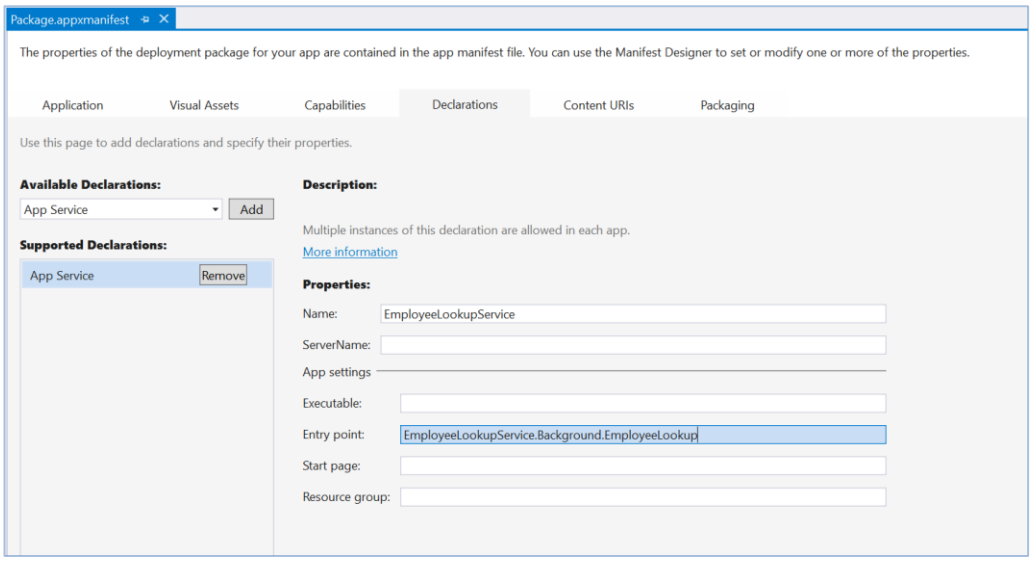

Figure 4

Register the app service in the EmployeeLookupService manifest.

3. Right-click on the **References** folder in the EmployeeLookupService app and choose **Add > Reference**. Add a reference to the **EmployeeLookupService.Background** project.

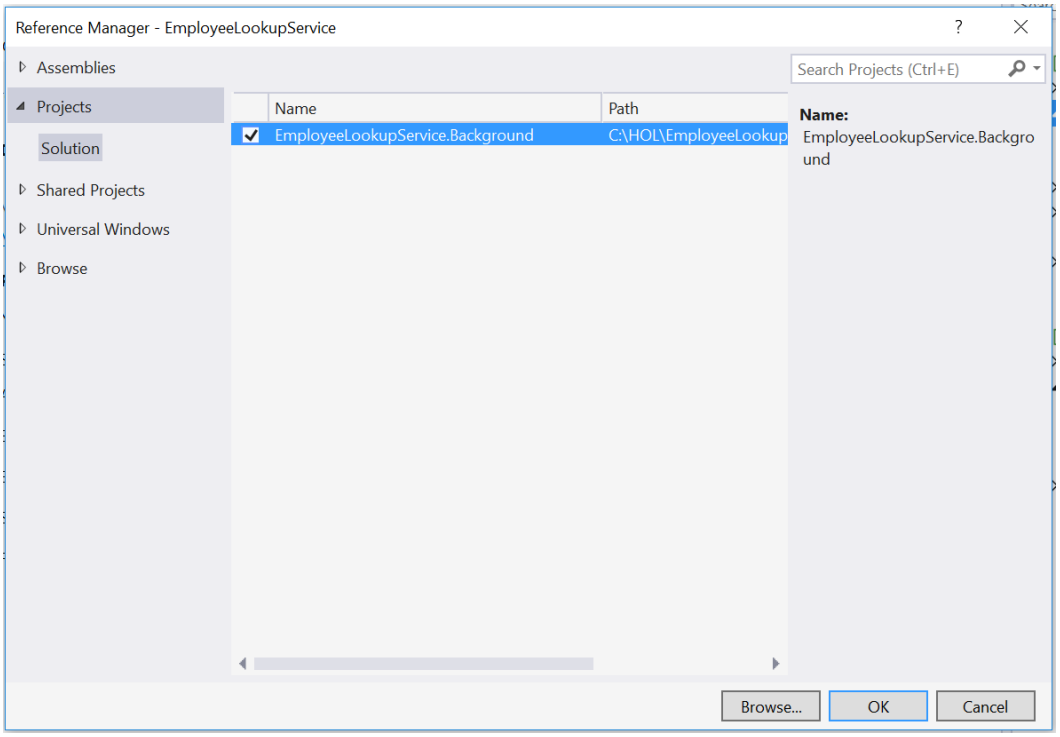

Figure 5

Add the WinRT component as a reference.

4. Build and run the EmployeeLookupService app to deploy your background task. The main page of the app will remain blank, because we didn't implement a UI for this app.

Note: We will explore ways to debug the background task in the next exercise.

5. Stop debugging and return to Visual Studio.

Exercise 2: Call the App Service from Another App

Now that you have created and registered an app service, you can call it from another app. In this exercise, you will create an app where you can input employee IDs, call the EmployeeLookup service, and receive results with the corresponding employee names.

Task 1 – Create a blank Universal Windows app

We will begin by creating a project from the Blank App template.

1. In a new instance of Visual Studio 2015, choose **File > New> Project** to open the New Project dialog. Navigate to **Installed > Templates > Visual C# > Windows > Universal** and select the **Blank App (Universal Windows)** template.

2. Name your project **AppServices** and select the file system location where you save your Hands-on Lab solutions.

   Leave the options selected to **Create new solution** and **Create directory for solution**. You may deselect both **Add to source control** and **Show telemetry in the Windows Dev Center** if you don't wish to version your work or use Application Insights. Click **OK** to create the project.

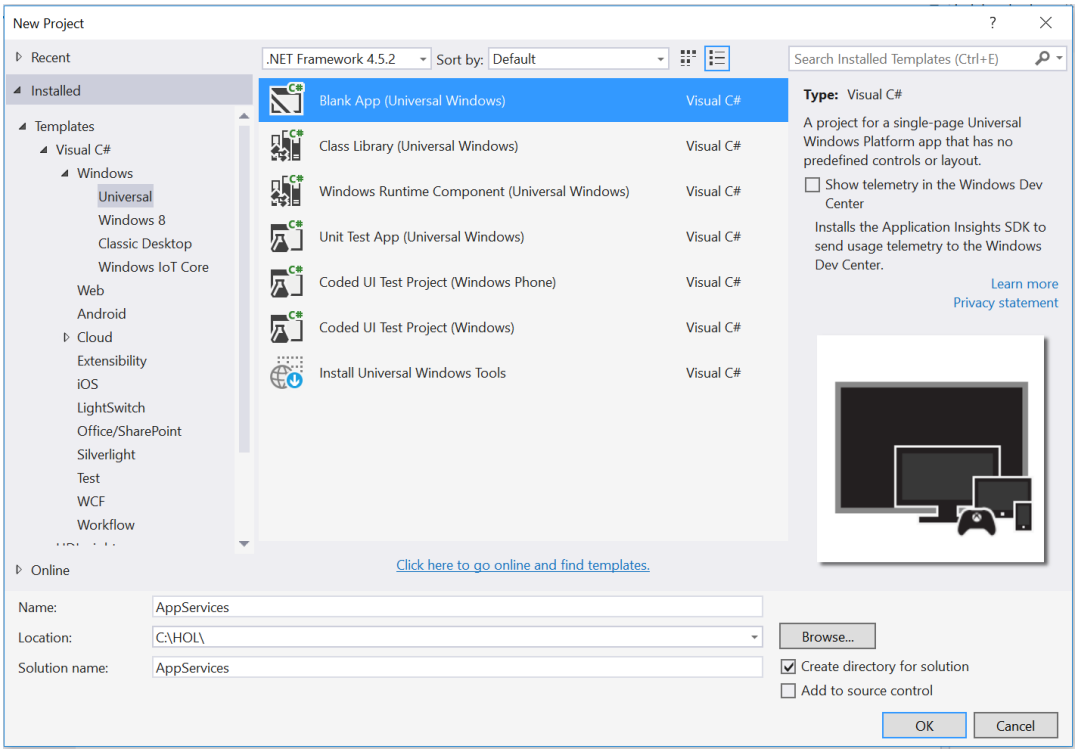

Figure 6

Create a new Blank App project in Visual Studio 2015.

3. Set your Solution Configuration to **Debug** and your Solution Platform to **x86**. Select **Local Machine** from the Debug Target dropdown menu.

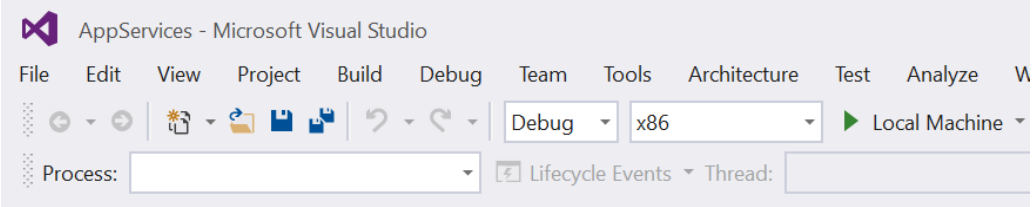

Figure 7

Configure your app to run on the Local Machine.

4. Build and run your app. You will see a blank app window with the frame rate counter enabled by default for debugging.

Figure 8

The blank universal app running in Desktop mode.

> **Note:** The frame rate counter is a debug tool that helps to monitor the performance of your app. It is useful for apps that require intensive graphics processing but unnecessary for the simple apps you will be creating in the Hands-on Labs.
>
> In the Blank App template, the preprocessor directive to enable or disable the frame rate counter is in **App.xaml.cs**. The frame rate counter may overlap or hide your app content if you leave it on. For the purposes of the Hands-on Labs, you may turn it off by setting **this.DebugSettings.EnableFrameRateCounter** to **false**.

5. Return to Visual Studio and stop debugging.

Task 2 – Create the UI and a simple model

We will create a simple employee class with Id and Name properties to assist in the display of the employee results from the lookup service. Then, we will create a simple UI to allow users to input IDs and query the app service.

1. Right-click on the AppServices project and choose **Add > Class**. Name the class **Employee.cs**.

2. Make the Employee class **public** and give it string properties of **Id** and **Name**.

C#

```
public class Employee
{
    public string Id { get; set; }

    public string Name { get; set; }
}
```

3. Save and close the Employee class.

4. Add a **StackPanel** containing a **TextBlock**, **TextBox**, and **Button** to the MainPage grid of the AppServices app.

XAML

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel>
        <TextBlock Text="Enter one employee ID per line." Margin="12" />
        <TextBox x:Name="EmployeeId" AcceptsReturn="True"
            Margin="12,0,12,12"/>
        <Button Content="Look up employee(s)" Margin="12,0,0,0"/>
    </StackPanel>
</Grid>
```

5. Add a click event on the button to call a **GetEmployeeById** handler. You will create the handler in the next step.

XAML

```
<Button Content="Look up employee(s)" Click="GetEmployeeById"
    Margin="12,0,0,0"/>
```

6. Create the **GetEmployeeById** handler in the MainPage code-behind.

C#

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

private void GetEmployeeById(object sender, RoutedEventArgs e)
{
```

```
}
```

Task 3 – Open the app service connection

The app service you created in Exercise 1 is set up to receive an incoming app service connection. In this task, you will open the app service connection and send a message to the app service with the employee lookup query.

1. Add the **Windows.ApplicationModel.AppService** namespace to the MainPage code-behind.

C#

```
using Windows.ApplicationModel.AppService;
```

2. Create an app service connection at the MainPage class level.

C#

```
public sealed partial class MainPage : Page
{
    private AppServiceConnection appServiceConnection;
```

3. In a separate instance of Visual Studio, load the **EmployeeLookupService** project and open its app manifest in the manifest editor. Under the **Packaging** tab, locate the **Package family name** and copy it to your clipboard.

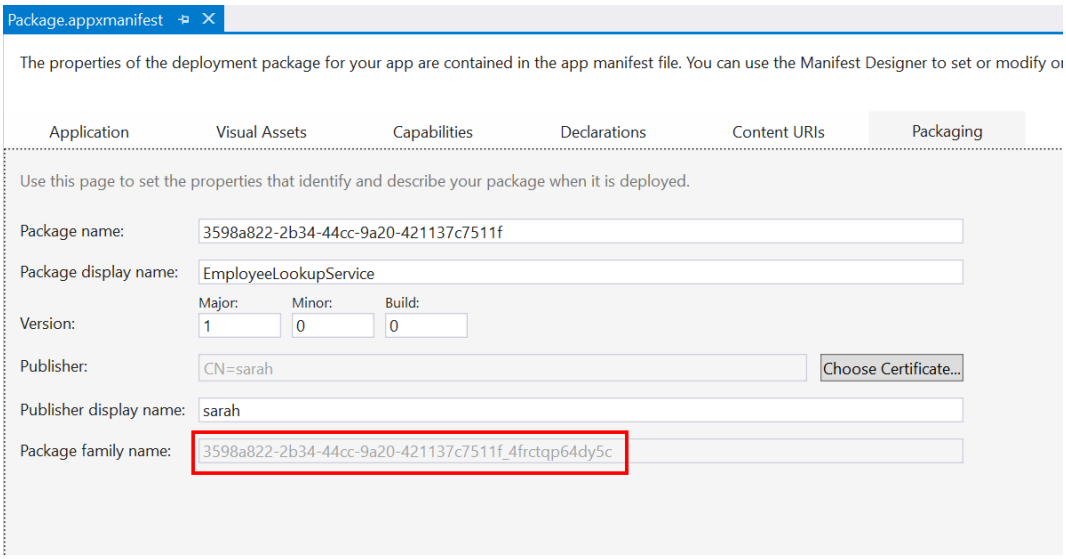

Figure 9

Copy the package family name from the EmployeeLookupService app manifest.

4. Set up the app service connection in the **GetEmployeeById()** handler to access your **EmployeeLookup** app service. You will need the service name you defined in the app service declaration and the package family name that you saved in the last step.

C#

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

    appServiceConnection = new AppServiceConnection
    {
        // Replace the AppServiceName and PackageFamilyName with those from
        // your EmployeeLookupServiceApp.

        AppServiceName = "EmployeeLookupService",
        PackageFamilyName =
            "3598a822-2b34-44cc-9a20-421137c7511f_4frctqp64dy5c"
    };
```

5. Make the **GetEmployeeById()** handler asynchronous and open the app service connection. We will based on the connection status to alert the user of any errors opening the connection. You will create the **LogError** method in the next step.

C#

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

    appServiceConnection = new AppServiceConnection
    {
        AppServiceName = "EmployeeLookup",
        PackageFamilyName =
            "3598a822-2b34-44cc-9a20-421137c7511f_4frctqp64dy5c"
    };

    var status = await appServiceConnection.OpenAsync();

    switch (status)
    {
        case AppServiceConnectionStatus.AppNotInstalled:
            await LogError("The EmployeeLookup application is not installed.
Please install it and try again.");
            return;
        case AppServiceConnectionStatus.AppServiceUnavailable:
            await LogError(
        "The EmployeeLookup application does not have the available feature");
            return;
        case AppServiceConnectionStatus.AppUnavailable:
            await LogError("The package for the app service is unavailable.
Have you added the WinRT component as a reference to the EmployeeLookupService
app?");
            return;
        case AppServiceConnectionStatus.Unknown:
            await LogError("Unknown Error.");
```

```
            return;
    }
```

6. Add the **System.Threading.Tasks** and **Windows.UI.Popups** namespaces to the MainPage code-behind.

C#

```
using System.Threading.Tasks;
using Windows.UI.Popups;
```

7. Add the **LogError** method to the code-behind to handle the status messages you created in Step 5. This method will open a message dialog with the status of the app service connection if it isn't opened successfully.

C#

```
private async Task LogError(string errorMessage)
{
    await new MessageDialog(errorMessage).ShowAsync();
}
```

8. In the **GetEmployeeById()** handler, parse the employee IDs from the **EmployeeId** input you created earlier on the MainPage view. You may remember the **AcceptsReturn="True"** attribute on that TextBox. We will allow users to to query the employee lookup service with multiple employee IDs as long as each ID is on its own line.

C#

```
    case AppServiceConnectionStatus.Unknown:
    await LogError("Unknown Error.");
    return;
}

var items = this.EmployeeId.Text.Split(new string[] { Environment.NewLine },
    StringSplitOptions.RemoveEmptyEntries);
```

9. Create a message consisting of a ValueSet() to send to the app service. Add each parsed employee ID to the message.

C#

```
var items = this.EmployeeId.Text.Split(new string[] { Environment.NewLine },
    StringSplitOptions.RemoveEmptyEntries);

var message = new ValueSet();

for (int i = 0; i<items.Length; i++)
{
    message.Add(i.ToString(), items[i]);
}
```

10. Send the message to the app service and await a response. Create a switch to handle potential issues with the status of the response.

```
C#
for (int i = 0; i<items.Length; i++)
{
    message.Add(i.ToString(), items[i]);
}

var response = await appServiceConnection.SendMessageAsync(message);

switch (response.Status)
{
    case AppServiceResponseStatus.ResourceLimitsExceeded:
        await LogError(
               "Insufficient resources. The app service has been shut down.");
        return;
    case AppServiceResponseStatus.Failure:
        await LogError("Failed to receive response.");
        return;
    case AppServiceResponseStatus.Unknown:
        await LogError("Unknown error.");
        return;
}
```

Task 4 – Display the results and debug the background task

Your UI can handle inputs and your app service is ready for queries. The final step is to display the query results and debug the background task if necessary. In this task, you will add a ListView to the UI to display one or more query results.

1. In **MainPage.xaml.cs**, add the System.Collections.ObjectModel namespace.

```
C#
using System.Collections.ObjectModel;
```

2. Create a new instance of an **ObservableCollection** that contains items of type **Employee**. You created the simple Employee class in Task 2 of this exercise.

```
C#
public sealed partial class MainPage : Page
{
    private AppServiceConnection appServiceConnection;
    public ObservableCollection<Employee> Items { get; set; } =
        new ObservableCollection<Employee>();
```

3. In the **GetEmployeeById** handler, add each item from the app service response to the Items collection.

C#

```
switch (response.Status)
{
    case AppServiceResponseStatus.ResourceLimitsExceeded:
        await LogError("Insufficient resources. The app service has been shut
                        down.");
        return;
    case AppServiceResponseStatus.Failure:
        await LogError("Failed to receive response.");
        return;
    case AppServiceResponseStatus.Unknown:
        await LogError("Unknown error.");
        return;
}

foreach (var item in response.Message)
{
    this.Items.Add(new Employee
    {
        Id = item.Key,
        Name = item.Value.ToString()
    });
}
```

4. Add a **ListView** to the **StackPanel** in MainPage.xaml. The ListView will display the employee id and name for each query result.

XAML

```
<StackPanel>
    <TextBlock Text="Enter one employee ID per line." Margin="12" />
    <TextBox x:Name="EmployeeId" AcceptsReturn="True" Margin="12,0,12,12"/>
    <Button Content="Look up employee(s)" Click="GetEmployeeById"
        Margin="12,0,0,0"/>
    <ListView ItemsSource="{x:Bind Items}" Grid.Column="1">
        <ListView.ItemTemplate>
            <DataTemplate x:DataType="local:Employee">
                <StackPanel>
                    <StackPanel Orientation="Horizontal" Margin="12">
                        <TextBlock Text="{x:Bind Id}" Margin="0,0,12,0" />
                        <TextBlock Text="{x:Bind Name}" />
                    </StackPanel>
                </StackPanel>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
```

```
</StackPanel>
```

Note: We are using x:Bind to set the **Items** data context for the ListView. For more on data binding, check out **the Data Binding** hands-on lab.

5. Build and run the **AppServices** app. Enter an employee ID in the text box and use the **Look up employee(s)** button to query the employee lookup service.

6. If your app service returns the employee name, congratulations! Your display will look something like this:

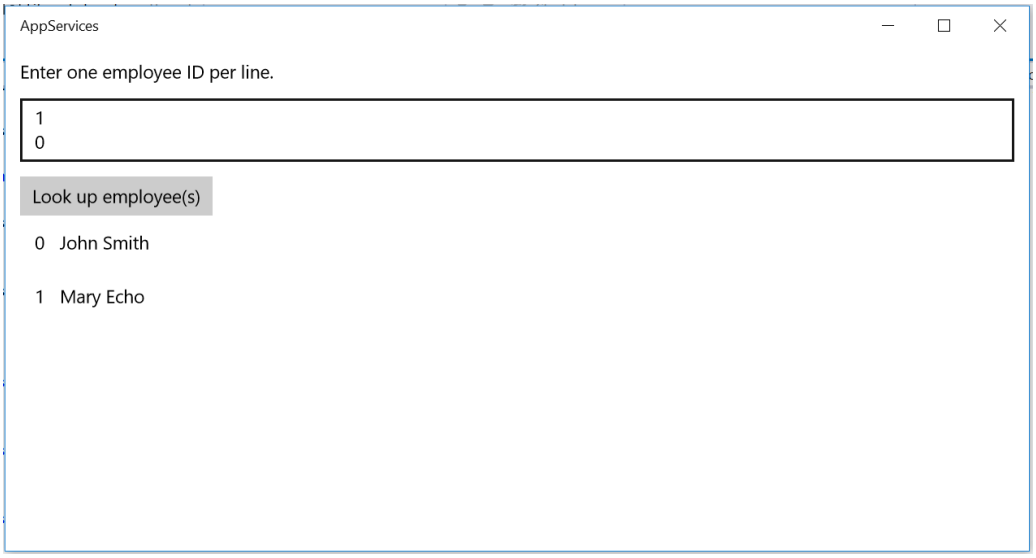

Figure 10
Results for a multiple-line employee lookup return from the app service.

7. Stop debugging and return to Visual Studio.

8. If you received an error when trying to do the lookup, let's do some troubleshooting. If you received the **AppUnavailable** status error, you may have forgotten to add the EmployeeLookupService.Background project as a reference to the EmployeeLookupService app.

   Alternatively, the **AppUnavailable** status error may indicate that your declared app service entrypoint does not match the class name in **the EmployeeLookup.cs** class.

9. To debug the background task itself, open the EmployeeLookupService solution.

10. Open the **EmployeeLookupService** project properties in the properties editor and browse to the **Debug** tab. Check the **Do not launch, but debug my code when it starts** option and save the properties file.

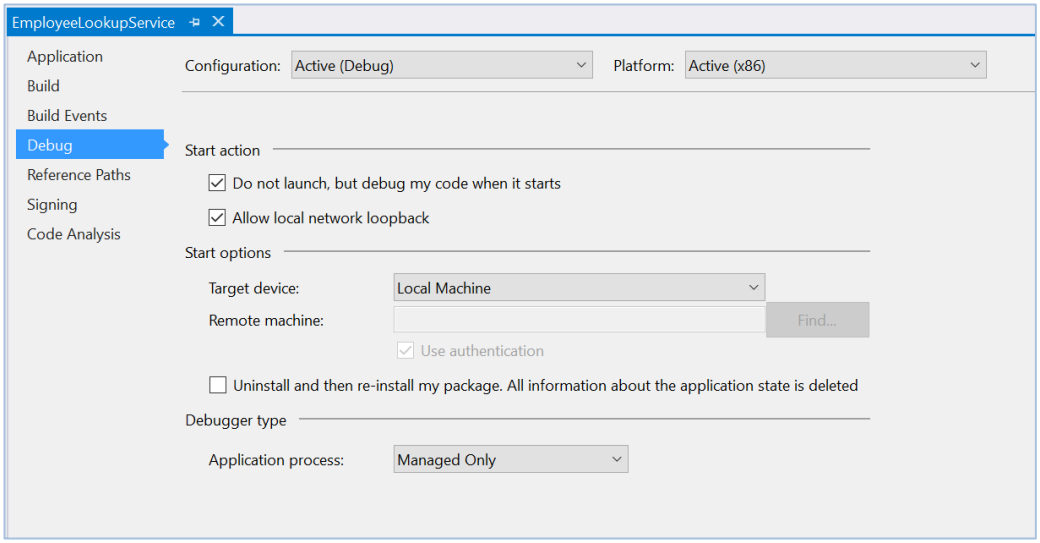

Figure 11

Set debug options to make it possible to debug the background task.

11. Set a breakpoint in your background task and use the **Start Debugging** button to attach the debugger. The EmployeeLookupService app will not launch.

12. Build and run the **AppServices** app and query the lookup service. You will hit the breakpoint in your background task.

Summary

In this lab, you learned how to create and register an app service and query it asynchronously from another app using an app service connection.

In an enterprise situation, you may want to consider wrapping your app service in an SDK and providing access through a class library. It is best practice to use REST-style versioning to update your app services and create new entrypoints for breaking changes.