Azure Developer Immersion

API App

In this walkthrough, you are going to take advantage of Azure App Service's API features in the web API presented by the rGroup app. Doing this will enable the use of an authentication model that works with both web sites and mobile apps, and which can support various third-party login providers, including Azure Active Directory (AAD), which you will be using, but also Google, Twitter, Facebook, and Microsoft Accounts. Furthermore, you can use API Apps from Azure Logic Apps, making it possible for your code to supply triggers that run workflows, or to be invoked from a workflow.

As you know, the app supports AAD login through the web site already, but adding mobile app support is non-trivial; it is considerably easier through Azure's App Service authentication features. Had the app been written to run in Azure from the start, a lot of the code it contains for working with AAD would not have been needed in the first place because it could just have let Azure App Service do the work.

API Apps are part of the suite of features collectively called Azure App Services. Although you can create separate Web, Mobile, and API Apps, you don't have to. Any of these three Azure App types can use any of the Azure App Service functionality. (In fact, the biggest difference between Web Apps, Mobile Apps, and API Apps is the icon the Azure Portal uses when showing these apps.) For example, you can take advantage of Mobile functionality in an API App. More importantly to our example, you can use API App features from inside a Web App. Therefore, although we're going to be using Azure's API App features in this walkthrough, you won't need to create a new API App to do so—you will just switch on API features in the Web App you created in the "lift and shift" walkthrough.

This walkthrough assumes you have completed the 'Getting Started' walkthrough and the first "lift and shift" walkthrough.

There are three exercises in this walkthrough:

1. API App
2. Create a Test Client App
3. Configure API Access

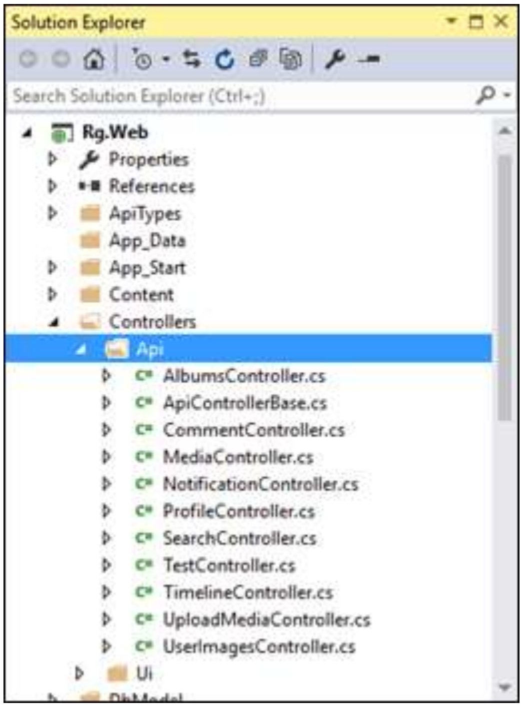
Expected duration: 45 minutes

Exercise 1: API App

In this first exercise, you will inspect and work with the API App for the rGroup application.

1. Your web browser should still be open to the Visual Studio Team Services Task Board for your Team Project. If not, open your web browser, access your Team Project, and access the Task Board.
2. Drag and drop the card **Complete the 02-API App Walkthrough** from the **To do** column to the **In progress** column.
3. In Visual Studio 2015, open the **Rg.Web** solution if you don't already have it open.

4. Select **View | Solution Explorer**.
5. In the **Rg.Web** project in **Solution Explorer**, expand the **Controllers** folder, and, inside that, expand the **Api** folder.

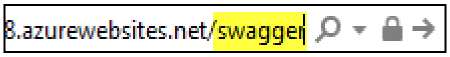


The various controller classes in this folder are all ordinary ASP.NET (4.6) Web API controllers. The web UI uses these from JavaScript. It would be hard to use these right now from a mobile app or desktop client because most of these controllers use the **[Authorize]** attribute to indicate that only authenticated callers can use them. Right now the only way for a user to authenticate is to go through the login UI in a web browser. Soon, we will fix this by enabling the Azure App Service authentication features, which can handle both web- and mobile-based logins.

6. To take full advantage of Azure's API Apps features, we need something that you will not find in all Web Apps, but which is present in **Rg.Web**. In **Solution Explorer**, look in the **App_Start** folder for the **SwaggerConfig.cs** file and open it.

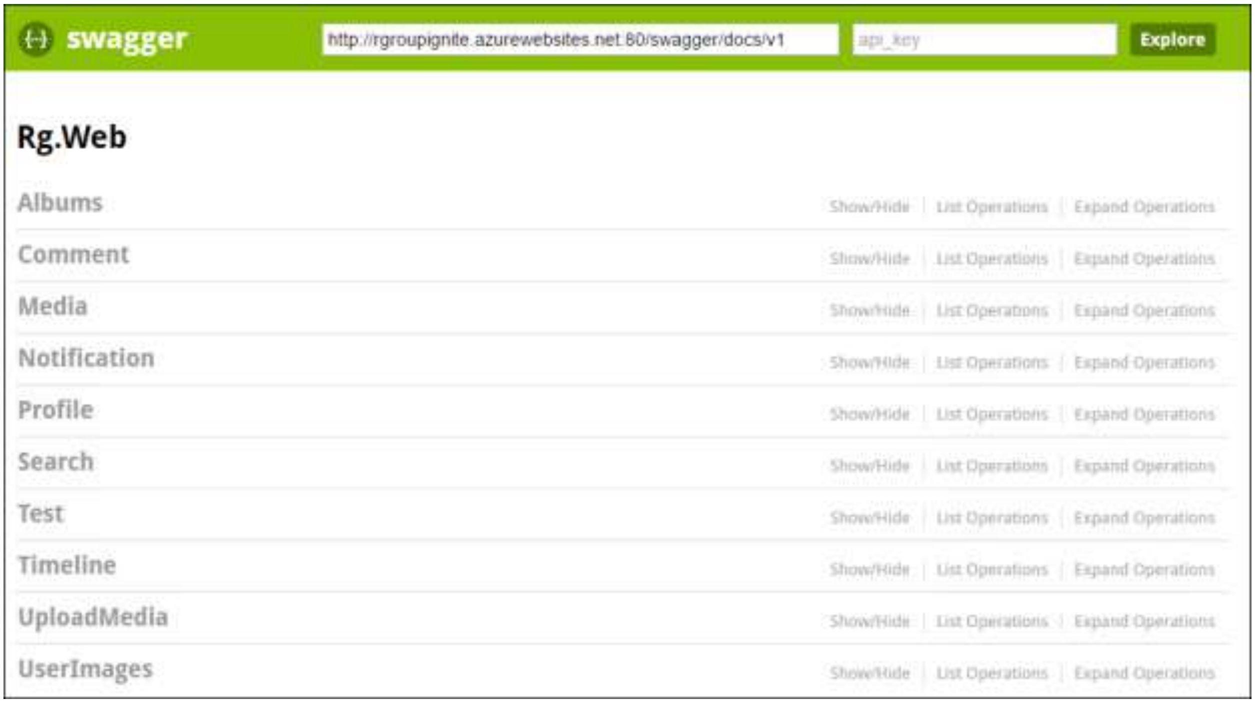
   Swagger is a system for describing web API. This code makes metadata describing our endpoints available. Most of this is commented out—the comments are there to show you how you can enable various optional features—but the basic functionality is enabled, as is an optional UI for browsing the API. This will enable Azure to discover everything it needs to know about our web API. You don't need to change anything; this has been present in the project from the start. Let's take a look at what it does.

7. In the web browser, if you still have a tab or window open on the web site you deployed to Azure in the "lift and shift" lab, return to it. If not, open a new browser tab and go to your site.
8. In the browser address bar, add **swagger** to the end of the address (right after the **.net/**) and press the **ENTER** key.

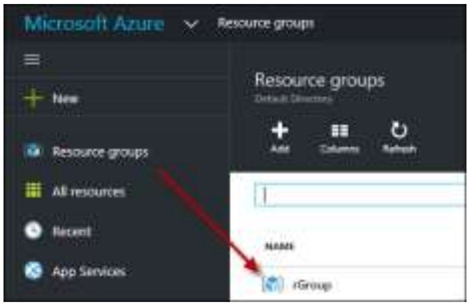


8.azurewebsites.net/swagger

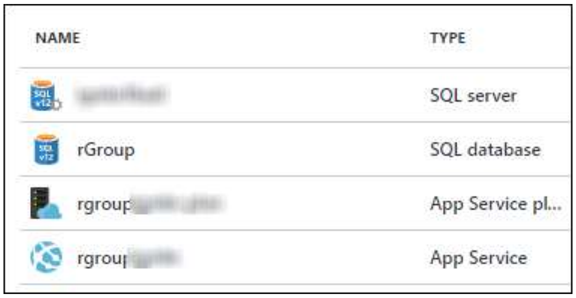
The browser will navigate to the test UI that enabled by the **SwaggerConfig.cs** file.


swagger
http://rgroupignite.azurewebsites.net:80/swagger/docs/v1
api_key
Explore
Rg.Web
Albums
Comment
Media
Notification
Profile
Search
Test
Timeline
UploadMedia
UserImages
Show/Hide | List Operations | Expand Operations

9. Leave this browser open; you will be coming back soon. Go to the Azure portal at https://portal.azure.com/ in a new browser tab.
10. Click on **Resource groups** from the list of actions on the left (which you can add from the **Browse** item if it's not already present), find the **rGroup** resource group there, and click on it.

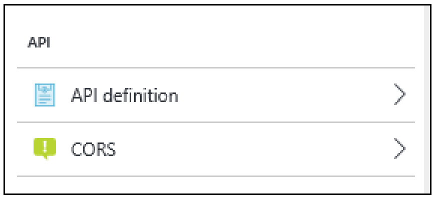


11. Click on the icon for the web app you created in the “lift and shift” walkthrough. The name that appears next to the icon will be whatever name you chose for the web app (**App Service**) when you created it.



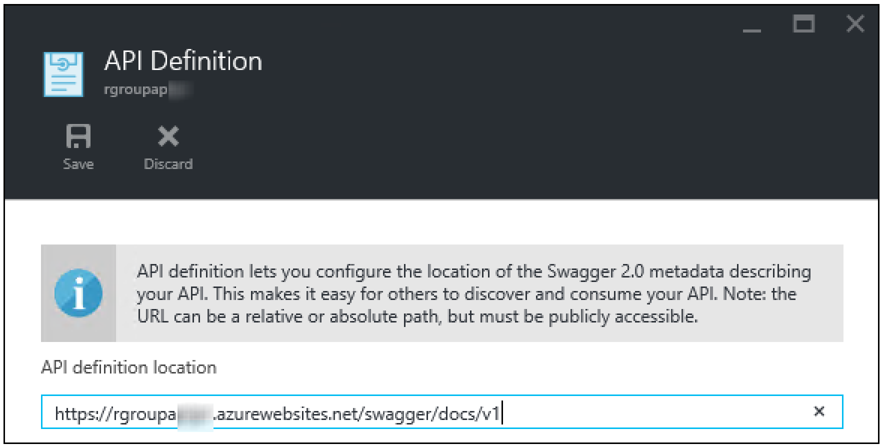
NAME	TYPE
[redacted]	SQL server
rGroup	SQL database
rgroup[redacted]	App Service pl...
rgrou[redacted]	App Service

12. In the Web App’s **Settings** select **API definition**.



Azure reads the machine-readable version of the information that you saw a moment ago through Swagger. It’s the same Swagger data you saw earlier in a browser UI, but in a downloadable JSON file.

Go back to the tab showing the Swagger UI, copy the address from the textbox at the top of the web page (not the browser address bar), then return to the Azure portal and paste it into the **API definition location** textbox. Click **Save**.



API definition lets you configure the location of the Swagger 2.0 metadata describing your API. This makes it easy for others to discover and consume your API. Note: the URL can be a relative or absolute path, but must be publicly accessible.

API definition location: https://rgroupa[redacted].azurewebsites.net/swagger/docs/v1

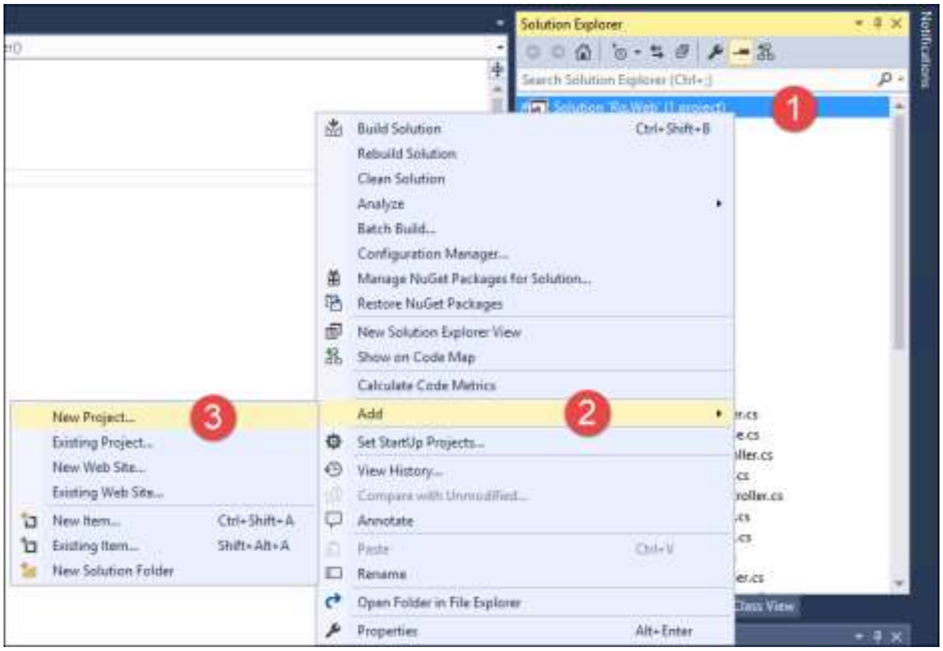
The Azure SDK can use this same information to make it easy to use API Apps from client code. In the next exercise, you will build a very simple client app that uses this API to demonstrate this.

13. Return to Visual Studio.

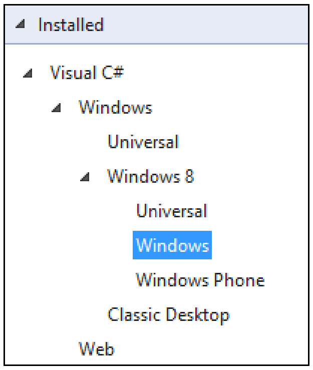
Exercise 2: Create a Test Client App

In this exercise, you will create a Windows Store app to be a test client of the API App.

1. Right-click on the Solution node in **Solution Explorer** and select **Add | New Project**.

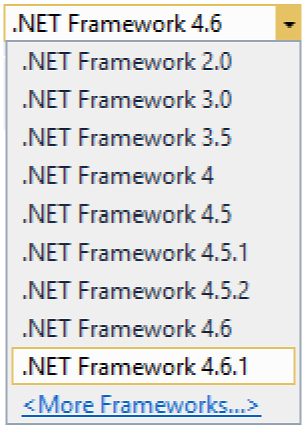


2. In the list on the left, expand **Visual C# | Windows | Windows 8 | Windows**.



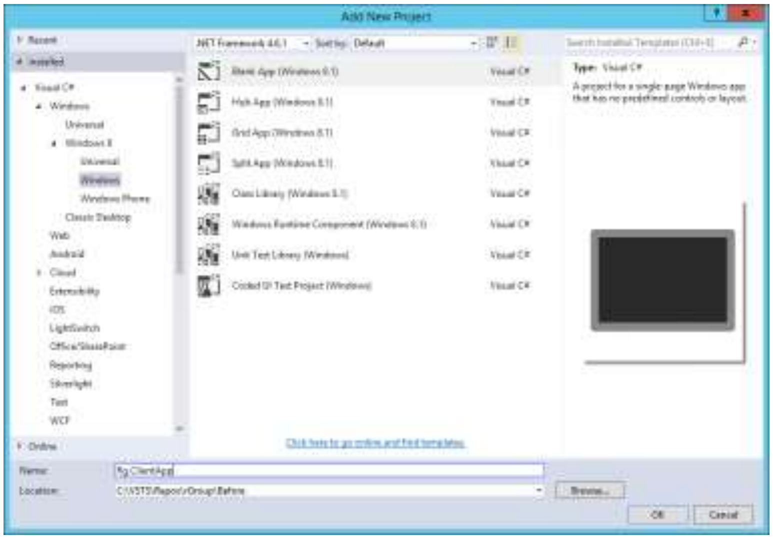
3. In the middle column of the dialog, select **Blank App (Windows 8.1)**. Make sure you do not select a Universal app template.

4. Make sure the Framework is set to **.NET Framework 4.6.1**.



.NET Framework 4.6

5. Set the **Name** of the app to **Rg.ClientApp**.



Add New Project

.NET Framework 4.6.1 Sort by: Default

Type: Visual C#

A project for a single-page Windows app that has no predefined controls or layout.

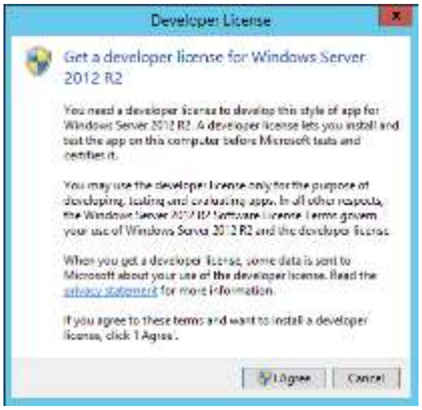
Name: Rg.ClientApp

Location: C:\VSTS\Repos\rGroup\Before

Recent
Installed
Visual C#
Windows
Universal
Windows 8
Universal
Windows
Windows Phone
Classic Desktop
Web
Android
Cloud
Extensibility
iOS
LightSwitch
Office/SharePoint
Reporting
Silverlight
Test
WCF
Online
Search Installed Templates (Ctrl+E)
Blank App (Windows 8.1) Visual C#
Hub App (Windows 8.1) Visual C#
Grid App (Windows 8.1) Visual C#
Split App (Windows 8.1) Visual C#
Class Library (Windows 8.1) Visual C#
Windows Runtime Component (Windows 8.1) Visual C#
Unit Test Library (Windows) Visual C#
Coded UI Test Project (Windows) Visual C#
Click here to go online and find templates.
Browse...
OK
Cancel

6. Click **OK** to add the project.

7. If you are running Windows Server 2012 R2 or Windows 8.1, you will most likely be prompted to get a developer license (see below).
Click **I Agree** if prompted.



Developer License

Get a developer license for Windows Server 2012 R2

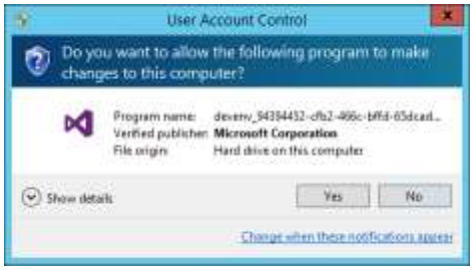
You need a developer license to develop this style of app for Windows Server 2012 R2. A developer license lets you install and test the app on this computer before Microsoft tests and certifies it.

You may use the developer license only for the purpose of developing, testing and evaluating apps. In all other respects, the Windows Server 2012 R2 Software License Terms govern your use of Windows Server 2012 R2 and the developer license.

When you get a developer license, some data is sent to Microsoft about your use of the developer license. Read the privacy statement for more information.

If you agree to these terms and want to install a developer license, click 'I Agree'.

8. Click **Yes** to the User Account Control dialog.



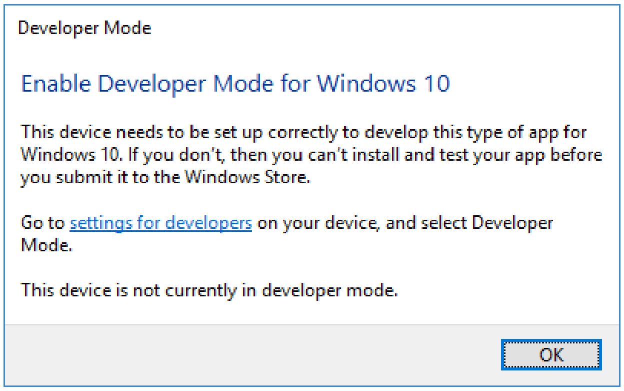
User Account Control

Do you want to allow the following program to make changes to this computer?

Program name: devenv_94394432-cfb2-466c-bffd-65dcad...
Verified publisher: **Microsoft Corporation**
File origin: Hard drive on this computer

When prompted, log in using your Microsoft account. Click **Close** once your developer license has been acquired.

If you are running Windows 10, you will see a slightly different prompt. It will tell you that you need to enable Developer Mode for Windows 10.



Developer Mode

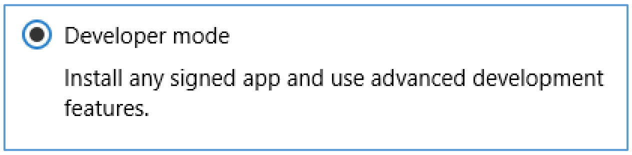
Enable Developer Mode for Windows 10

This device needs to be set up correctly to develop this type of app for Windows 10. If you don't, then you can't install and test your app before you submit it to the Windows Store.

Go to settings for developers on your device, and select Developer Mode.

This device is not currently in developer mode.

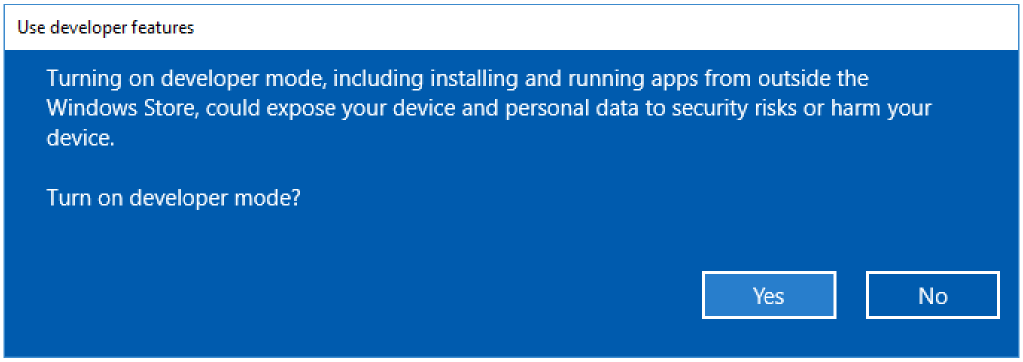
Click on the **settings for developers** link on this dialog. It will open the Windows settings app on the **For developers** page of the **UPDATE & SECURITY** section. Select the **Developer mode** radio button.



● Developer mode

Install any signed app and use advanced development features.

You may be shown a warning describing the risks of developer mode.



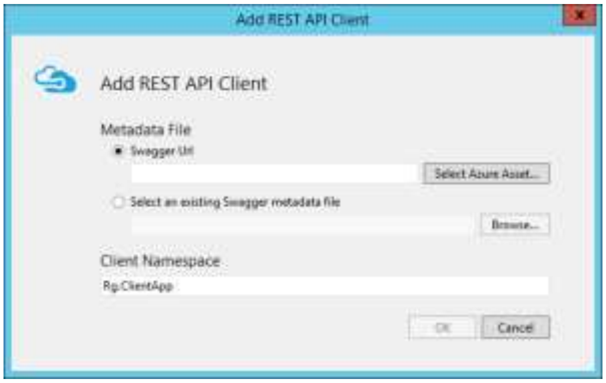
Use developer features

Turning on developer mode, including installing and running apps from outside the Windows Store, could expose your device and personal data to security risks or harm your device.

Turn on developer mode?

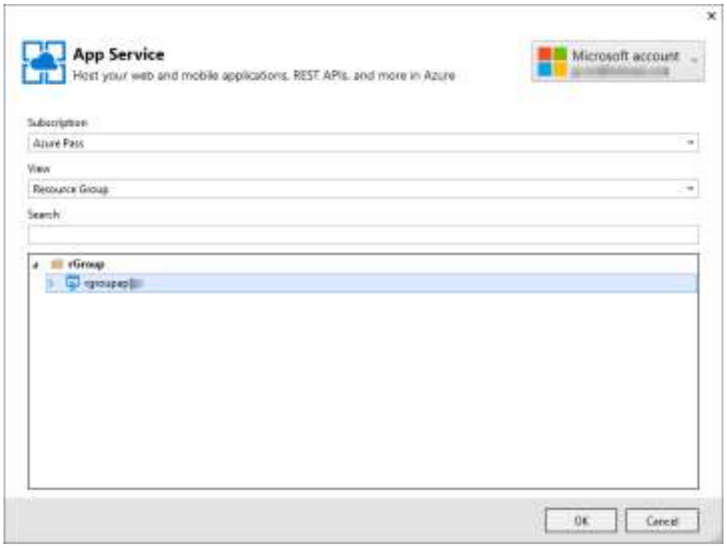
Click **Yes** to enable developer mode. Return to Visual Studio. Click OK to dismiss the dialog asking you to enable developer mode.

9. Select **Build | Build Solution**.
10. Right-click on your new project **Rg.ClientApp** in **Solution Explorer** and choose **Add | REST API Client**.
11. Ensure **Swagger Url** is selected under the **Metadata File** group. Click the **Select Azure Asset** button.



Client Namespace: Rg.ClientApp

12. In the **App Service** dialog, expand the **rGroup** folder and select your API app and then click **OK**.

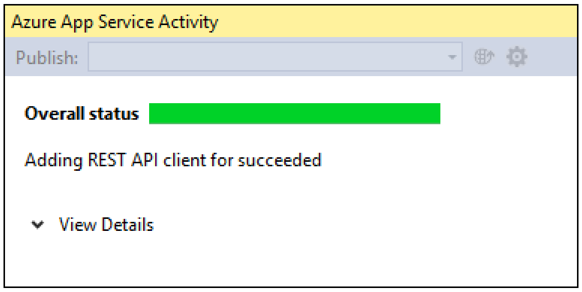


Subscription: Azure Pass

View: Resource Group

In the **Add REST API Client** dialog, notice the URL is the same as the one you looked at earlier in the portal.

13. Click **OK** to complete the **Add REST API Client** command and wait for it to finish.

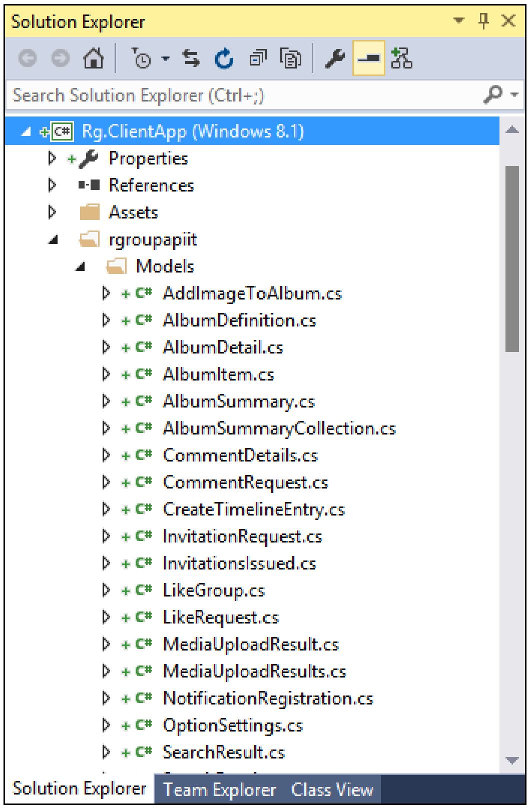


Adding REST API client for succeeded

Visual Studio downloads the metadata for the API and generates a set of types to enable you to use the API. It also adds various helper classes for using the API.

14. Look in the added folders and you will see all of these imported types

> Your folder names might be slightly different from the screen shot based on your service name.



```
Rg.ClientApp (Windows 8.1)
  Properties
  References
  Assets
  rgroupapiit
    Models
      AddImageToAlbum.cs
      AlbumDefinition.cs
      AlbumDetail.cs
      AlbumItem.cs
      AlbumSummary.cs
      AlbumSummaryCollection.cs
      CommentDetails.cs
      CommentRequest.cs
      CreateTimelineEntry.cs
      InvitationRequest.cs
      InvitationsIssued.cs
      LikeGroup.cs
      LikeRequest.cs
      MediaUploadResult.cs
      MediaUploadResults.cs
      NotificationRegistration.cs
      OptionSettings.cs
      SearchResult.cs
```

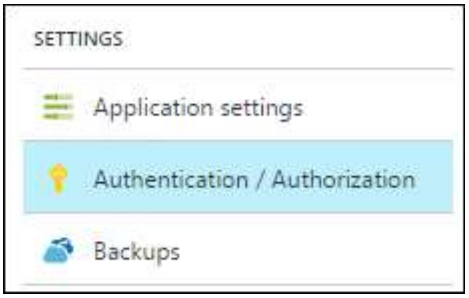
You are not quite ready to use the API yet because most of the methods need to know who the caller is and you have not yet enabled authentication on this API.

15. Now is a good time to commit your changes to your local Git repo. In **Solution Explorer**, right-click on the **Solution** node and select **Commit**.
16. At the **Changes** page, enter **Add client test app** as the commit message.
17. Click the **+** button in the **Related Work Items** section. In the box that opens, enter the Work Item ID for your current task (jump over to your open Visual Studio Team Services tab if you do not remember the ID) and click **Add**.
18. Click the **Commit All** button. If prompted to save your changes, select **Yes**. Doing a commit will quickly save your changes to your local repo but not push them up to Visual Studio Team Services.

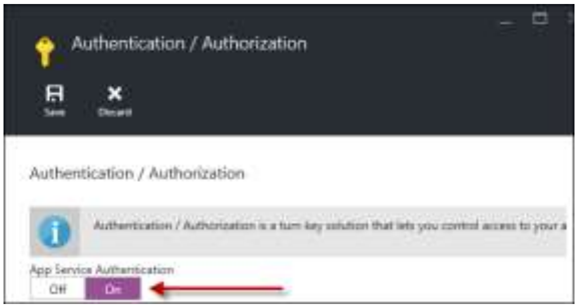
Exercise 3: API Authentication

In this last exercise for this walkthrough, you will configure authentication on your Web App in Azure and add some code to the client app to log the user in.

1. In the Azure portal, go to the pane for your Web App. If you don't already have this open, go back to the list of resources in your **rGroup** resource group and select the Web App.
2. In the **Settings** pane, select **Authentication / Authorization**.



3. In the **Authentication / Authorization** blade, change **App Service Authentication** to **On**.

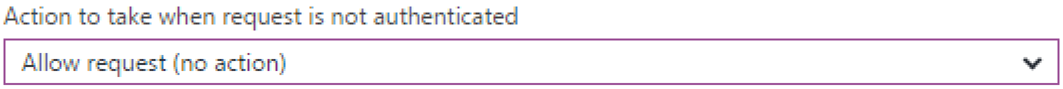

Authentication / Authorization
Save
Discard
Authentication / Authorization

Authentication / Authorization is a turn-key solution that lets you control access to your a

App Service Authentication

Off On

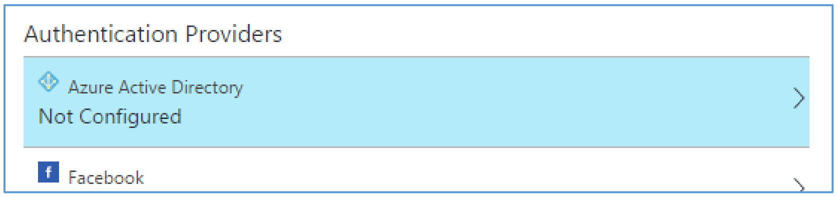
4. In the combo-box that appears, ensure that **Allow request (no action)** is selected. This enables our app to decide which pages and which API endpoints require authentication, and which are accessible to any user.



Action to take when request is not authenticated

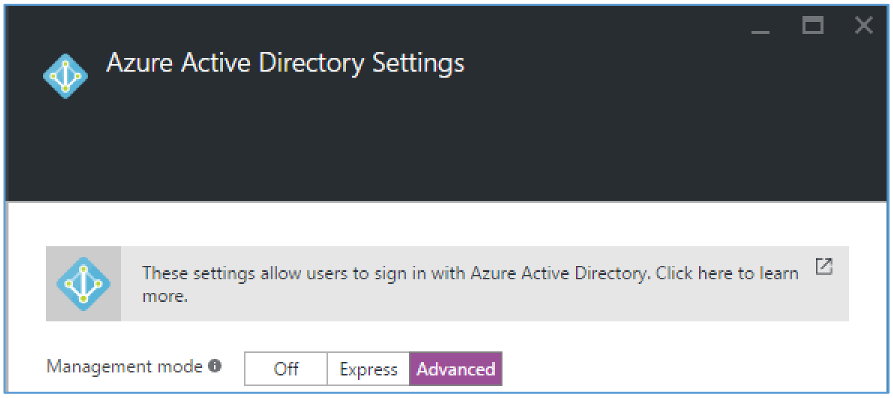
Allow request (no action)

5. In the **Authentication Providers** section, select **Azure Active Directory**.



Azure Active Directory
Not Configured

6. In the **Azure Active Directory Settings** pane, select **Advanced**.

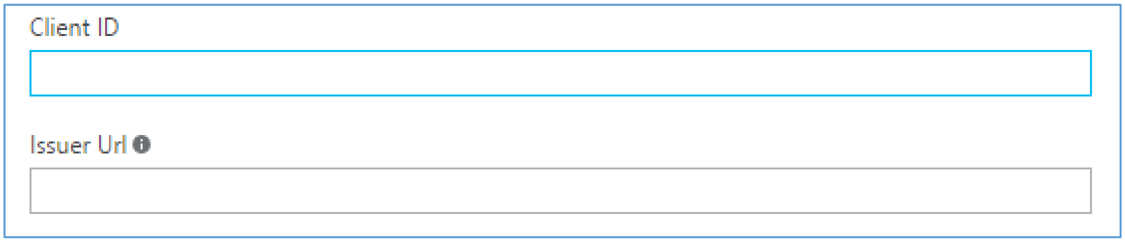


Azure Active Directory Settings

These settings allow users to sign in with Azure Active Directory. Click here to learn more.

Management mode: Off | Express | [✓ Advanced]

7. This will cause **Client ID** and **Issuer Url** fields to appear

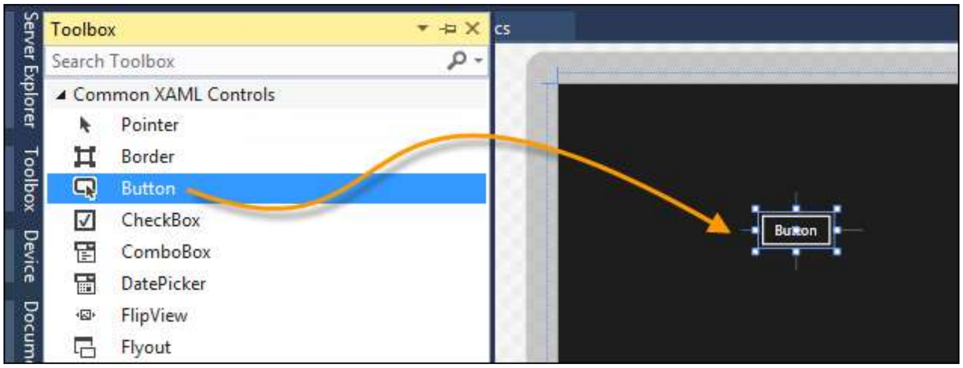


Client ID

Issuer Url

Fill these in with the **Client ID** for your AAD Application and the **Issuer Url** for your AAD Tenant. (We asked you to note these down in the 'Getting Started' module.)

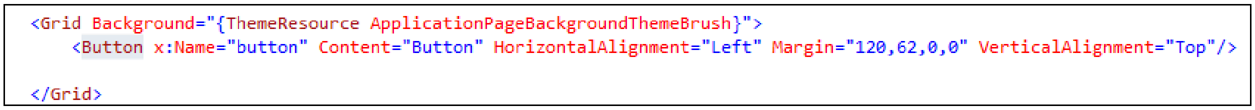
8. Click **OK** on the **Azure Active Directory Settings** pane.
9. Click **Save** on the **Authentication / Authorization** pane.
10. Return to Visual Studio.
11. In **Solution Explorer**, in the **Rg.ClientApp** app, double-click the **MainPage.xaml** file. Feel free to close or minimize the **Output** window to give yourself more room.
12. Open Visual Studio's **Toolbox** pane from the **View** menu.
13. In the toolbox, under **Common XAML Controls**, drag a **Button** onto the page.



14. In the XAML window below the design surface, replace the one line of XAML for the button with the following two lines:

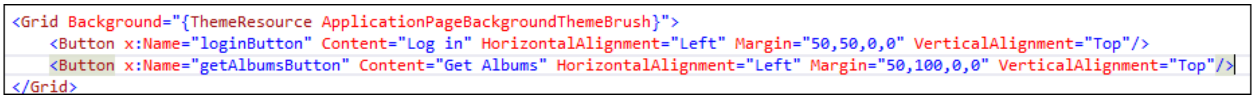
```
<Button x:Name="loginButton" Content="Log in" HorizontalAlignment="Left"
Margin="50,50,0,0" VerticalAlignment="Top"/>
<Button x:Name="getAlbumsButton" Content="Get Albums" HorizontalAlignment="Left"
Margin="50,100,0,0" VerticalAlignment="Top"/>
```

Before



```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Button x:Name="button" Content="Button" HorizontalAlignment="Left" Margin="120,62,0,0" VerticalAlignment="Top"/>

</Grid>
```

After



```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Button x:Name="loginButton" Content="Log in" HorizontalAlignment="Left" Margin="50,50,0,0" VerticalAlignment="Top"/>
    <Button x:Name="getAlbumsButton" Content="Get Albums" HorizontalAlignment="Left" Margin="50,100,0,0" VerticalAlignment="Top"/>
</Grid>
```

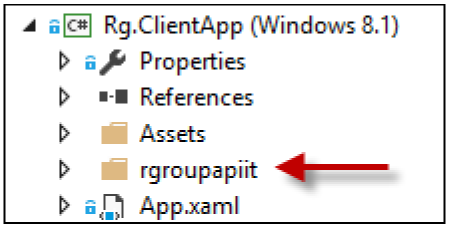
15. On the design surface, double-click the **Log in** button to add a click handler. Visual Studio should switch to a new tab showing the **MainPage.xaml.cs** code-behind file.

16. At the top of the **MainPage.xaml.cs** file, add these using directives below the existing ones:

```
using System.Net;
using System.Net.Http.Headers;
using Windows.Security.Authentication.Web;
using Windows.Storage;
using Rg.ClientApp.Models;
```

Next you are going to add a field in the code behind. The field's data type will depend on the name you gave your API App. When you used the client code generation earlier, it created a class based on the name of the API App.

For example, if your web site was called rgroupapitest, you will find a class called **Rgroupapitest**. However, since your App will have been called something else because Azure Web Apps must have unique names, you will need to use whatever is the right name for your API. A quick way to find the name of your web app is to look at the folder above the App.xaml file in the Solution Explorer. In the example below the folder is rgroupapiit. Thus, the type name is Rgroupapiit (note the capital R).

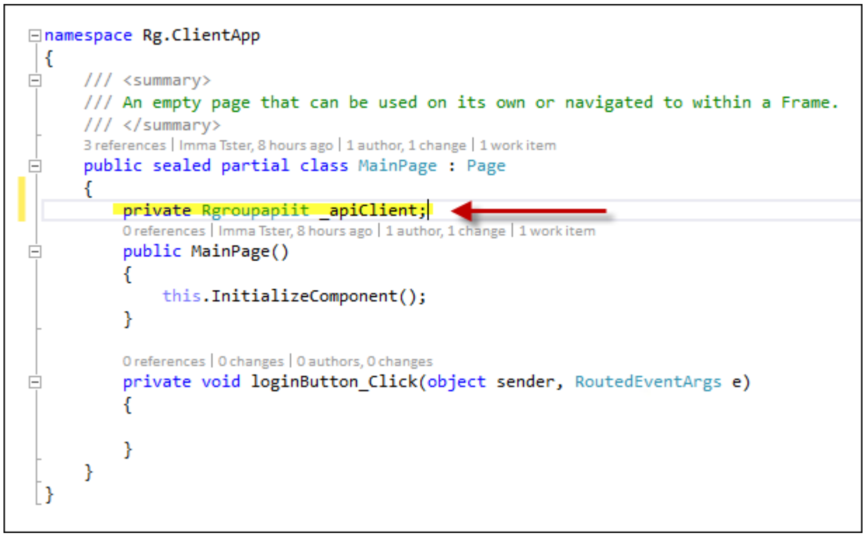


Rg.ClientApp (Windows 8.1)
Properties
References
Assets
rgroupapiit
App.xaml

You'll want to name the field **_apiClient**. See the screen shot below where you should put your field. Remember to use the correct type name.

17. Add your field. See below where it should go in the source file.

Rgroupapiit is an *example* type name—yours will be different.



```
namespace Rg.ClientApp
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    3 references | Imma Tster, 8 hours ago | 1 author, 1 change | 1 work item
    public sealed partial class MainPage : Page
    {
        private Rgroupapiit _apiClient;
        0 references | Imma Tster, 8 hours ago | 1 author, 1 change | 1 work item
        public MainPage()
        {
            this.InitializeComponent();
        }

        0 references | 0 changes | 0 authors, 0 changes
        private void loginButton_Click(object sender, RoutedEventArgs e)
        {

        }
    }
}
```

18. Inside the class below this new field, add string constants like below but using the URL for your web app, which you can get from the Azure portal, and the AAD Tenant and Client IDs which you noted down when creating the AAD Application in the 'Getting Started' walkthrough.

Make sure you use **https** in this constant. The URL value in the portal will be http.

```
private const string WebAppUrl = "https://YOURWEBAPP.azurewebsites.net";
private const string TenantId = "YOUR AAD TENANT ID HERE";
private const string ClientId = "YOUR AAD APPLICATION CLIENT ID HERE";
```

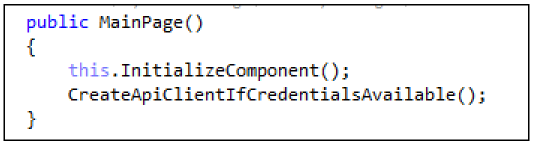
19. Add the following method to the class:

```
private void CreateApiClientIfCredentialsAvailable()
{
    object apiTokenValue;
    if (!ApplicationData.Current.LocalSettings.Values.TryGetValue(
          "ApiToken", out apiTokenValue))
    {
        getAlbumsButton.IsEnabled = false;
        return;
    }
    string apiToken = (string) apiTokenValue;

    _apiClient = new YOURTYPENAME(new Uri(WebAppUrl));
    _apiClient.HttpClient.DefaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue("Bearer", apiToken);

    getAlbumsButton.IsEnabled = true;
}
```

You need to fix the highlighted line and replace YOURTYPENAME with the name of the type you used for the first field you added to this class (see step **17**).

20. In **MainPage.xaml.cs** tab, after the line of code **this.InitializeComponent();**, add this code:

```
CreateApiClientIfCredentialsAvailable();
```



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

21. Add the **async** keyword before the **void** keyword for the **loginButton** click handler. It should look like the following:

```
private async void loginButton_Click(object sender, RoutedEventArgs e)
```

22. In the **loginButto**n click handler, add this code:

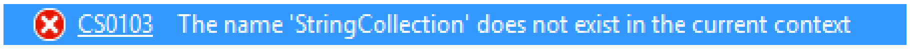
```
string nonce = Guid.NewGuid().ToString("D");
string redirectUrl = WebAppUrl;
var wa = await WebAuthenticationBroker.AuthenticateAsync(
    WebAuthenticationOptions.None,
    new Uri(
$"https://login.microsoftonline.com/{TenantId}/oauth2/authorize?response_type=id_toke
n&client_id={ClientId}&scope=openid&nonce={nonce}&response_mode=fragment&redirect_uri
={WebUtility.UrlEncode(redirectUrl)}"),
    new Uri(redirectUrl));

string d = wa.ResponseData;
int fragmentPos = d.IndexOf('#');
if (fragmentPos >= 0)
{
    string fragment = d.Substring(fragmentPos + 1);
    string[] elements = fragment.Split('&');
    Dictionary<string, string> fragmentValues =
        (from element in elements
            let parts = element.Split('=')
            where parts.Length == 2
            select new { key = parts[0], value = parts[1] })
            .ToDictionary(p => p.key, p => p.value);
    string aadToken;
    if (fragmentValues.TryGetValue("id_token", out aadToken))
    {

        ApplicationData.Current.LocalSettings.Values["ApiToken"] = aadToken;
    }
}

CreateApiClientIfCredentialsAvailable();
```

When run, this code shows the AAD login UI (displayed in a small web browser hosted in the app). On completion, it extracts the access token from the response. This will then be picked up by the CreateApiClientIfCredentialsAvailable method, which will then work on subsequent runs because it stores these values persistently.

23. Go back to the **MainPage.xaml** designer and double-click the **Get Albums** button to add a click handler. Again, you might have to manually switch to the new tab for MainPage.xaml.cs.
24. Add the **async** keyword in front of the new handler's **void** keyword.
25. Add this code to the button's click handler:

```
IList<string> testResult = await _apiClient.Test.GetAsync();
IList<AlbumSummary> albums = await _apiClient.Albums.GetAllAsync();
```

26. Select **Build | Build Solution**.
27. You will find it reports an error in a file called **Test.cs**.



CS0103 The name 'StringCollection' does not exist in the current context

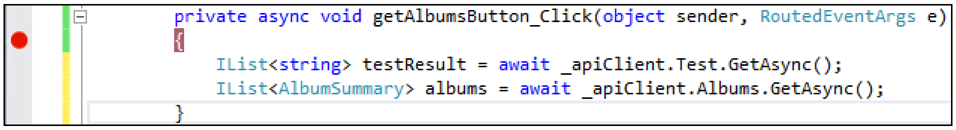
28. Double-click on the error in the **Error List** window to go to the file.

    This seems to be a bug in the current API client import tooling.

29. You can fix it by adding the following using directive to the top of that file:

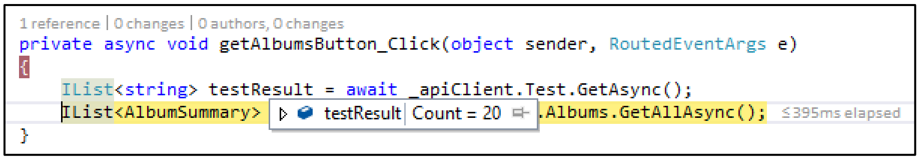
```
using Rg.ClientApp.Models;
```

30. Select **Build | Build Solution** again.
31. In the **Solution Explorer**, right-click on the **Rg.ClientApp** project and choose **Set as StartUp Project**.
32. In the **MainPage.xaml.cs** file, set a breakpoint on the first line of the **getAlbumsButton** click handler by placing your cursor on the line and pressing the **F9** key.



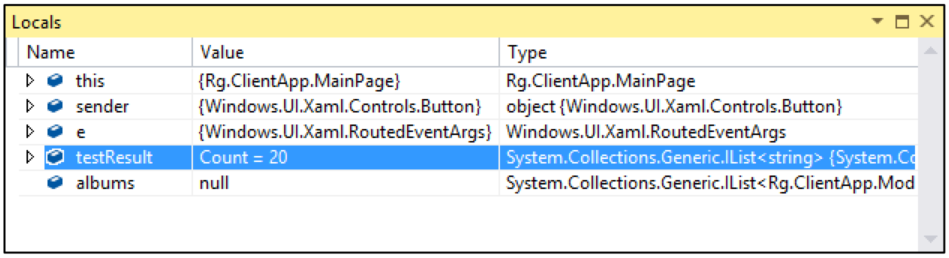
```
private async void getAlbumsButton_Click(object sender, RoutedEventArgs e)
{
    IList<string> testResult = await _apiClient.Test.GetAsync();
    IList<AlbumSummary> albums = await _apiClient.Albums.GetAsync();
}
```

33. Press **F5** to run the app.
34. Click the **Log in** button.
35. When a login prompt appears, sign into your Microsoft account.
36. Click the **Get Albums** button.
37. The debugger will stop at the click handler.
38. Step over the line that fetches test results by pressing **F10** twice.
39. Inspect the **testResult** variable by selecting it and then clicking the down arrow of the tool that appears.



```
1 reference | 0 changes | 0 authors, 0 changes
private async void getAlbumsButton_Click(object sender, RoutedEventArgs e)
{
    IList<string> testResult = await _apiClient.Test.GetAsync();
    IList<AlbumSummary> ▷ testResult Count = 20 .Albums.GetAllAsync();  ≤395ms elapsed
}
```

    You can also view details in the Locals window, which might be easier.



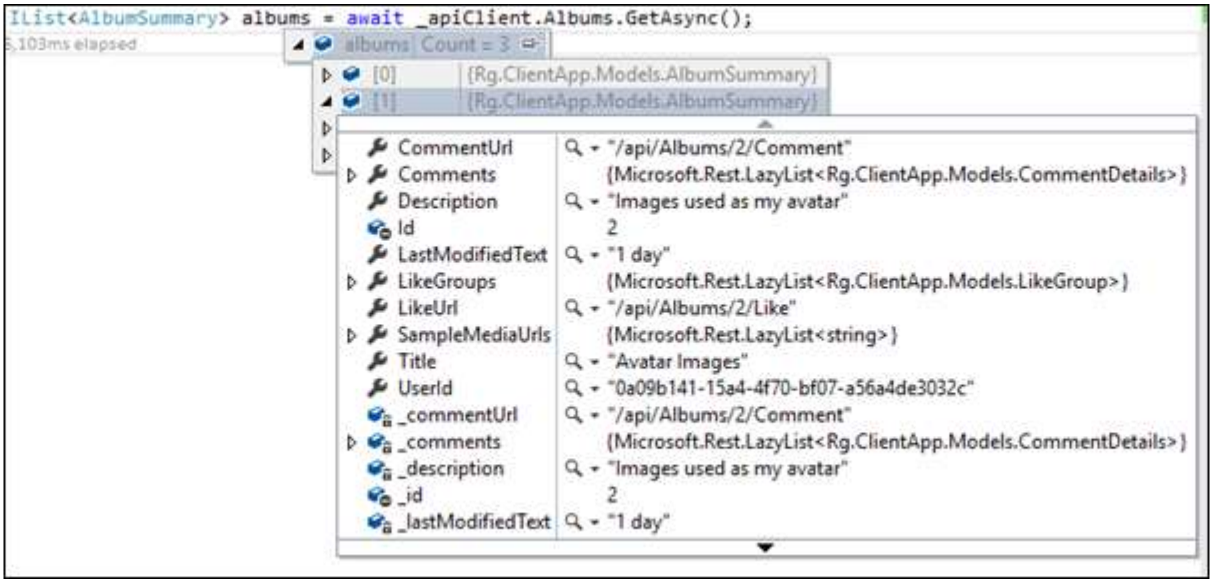
Locals

Name	Value	Type
▷ this	{Rg.ClientApp.MainPage}	Rg.ClientApp.MainPage
▷ sender	{Windows.UI.Xaml.Controls.Button}	object {Windows.UI.Xaml.Controls.Button}
▷ e	{Windows.UI.Xaml.RoutedEventArgs}	Windows.UI.Xaml.RoutedEventArgs
▷ testResult	Count = 20	System.Collections.Generic.IList<string> {System.Co
albums	null	System.Collections.Generic.IList<Rg.ClientApp.Mod

    If the **testResult** list contains a single item with an error message, stop debugging, go to the Azure portal, find the Web App, and click its **Restart** button and try again. Sometimes changes to authentication settings do not percolate fully through the system until you restart. If this happens, stop debugging and try again. Once it is working you should see something like above.

This is the same information as the test endpoint reported when you tried it from the Swagger UI. This verifies that the user has been identified successfully when logging in through the client app.

40. Step over the next line by pressing **F10** again. Inspect the **albums** variable. It should contain a list with one entry for each of your albums. If you expand one of the entries, you will see information about the album like this:



```
IList<AlbumSummary> albums = await _apiClient.Albums.GetAsync();
5,103ms elapsed
  albums  Count = 3
    [0]   {Rg.ClientApp.Models.AlbumSummary}
    [1]   {Rg.ClientApp.Models.AlbumSummary}
      CommentUrl         "/api/Albums/2/Comment"
      Comments           {Microsoft.Rest.LazyList<Rg.ClientApp.Models.CommentDetails>}
      Description        "Images used as my avatar"
      Id                 2
      LastModifiedText   "1 day"
      LikeGroups         {Microsoft.Rest.LazyList<Rg.ClientApp.Models.LikeGroup>}
      LikeUrl            "/api/Albums/2/Like"
      SampleMediaUrls    {Microsoft.Rest.LazyList<string>}
      Title              "Avatar Images"
      UserId             "0a09b141-15a4-4f70-bf07-a56a4de3032c"
      _commentUrl        "/api/Albums/2/Comment"
      _comments          {Microsoft.Rest.LazyList<Rg.ClientApp.Models.CommentDetails>}
      _description       "Images used as my avatar"
      _id                2
      _lastModifiedText  "1 day"
```

41. This verifies that your API App is able to return user-specific data. You could, of course, go on to display some of this data in the app, but the focus of this lab is authenticated access to the API, so we're going to stop now that we've demonstrated that this is working.

42. Select **Debug | Stop Debugging**.

43. In the **Solution Explorer**, right-click on the **Solution** node for **Rg.Web** and select **Commit**.

44. Enter a message like **Completed configuring API app and test client**.

45. Under **Related Work Items** click **+**.

46. Enter the ID for your task (if necessary, go back to the Task Board to get the ID). It will most likely be **4** if you created a new account today.

47. Once you have entered it, click **Add**.

48. When ready, click **Commit All and Push**.

49. Go to the Task Board and drag and drop your task to **Done**.

    You have now completed the second walkthrough. Stand up, take a quick break, and move on to the next one.