Windows 10

Hands-on lab

Lab: Connecting a UWP Client to Azure Mobile Apps

September 2015

Microsoft

CONTENTS

Overview

Azure App Service is a fully managed Platform as a Service (PaaS) offering for professional developers that brings a rich set of capabilities to web, mobile and integration scenarios. Mobile Apps in Azure App Service offer a highly scalable, globally available mobile application development platform for Enterprise Developers and System Integrators that brings a rich set of capabilities to mobile developers.

With Mobile Apps you can:

- Build native and cross platform apps - whether you're building native iOS, Android, and Windows apps or cross-platform Xamarin or Cordova (Phonegap) apps, you can take advantage of App Service using native SDKs.
- Connect to your enterprise systems - with Mobile Apps you can add corporate sign on in minutes, and connect to your enterprise on-premises or cloud resources.
- Connect to SaaS APIs easily - with more than 40 SaaS API connectors, you can easily integrate your app with SaaS APIs your enterprise uses today.
- Build offline-ready apps with sync - make your mobile workforce productive by building apps that work offline and use Mobile Apps to sync data in the background when connectivity is present with any of your enterprise data sources or SaaS APIs.
- Push Notifications to millions in seconds - engage your customers with instant push notifications on any device, personalized to their needs, sent when the time is right.

In this lab, you will connect a UWP app to an Azure App Service Mobile App backend. You will first run the backend services locally on your development PC and explore the code to understand the fundamentals of how a client app and a mobile backend service interact. You will modify the client app to authenticate against Azure Active Directory and connect it to the mobile backend hosted in Microsoft Azure. Finally, you will modify the client app to implement offline sync capability, enabling one of the most powerful capabilities offered by Azure App Service Mobile App.

Objectives

This lab will show you how to:

- Run a Windows 10 UWP app that is coded to sync data with Azure App Service Mobile App backend, running the backend services locally on your PC.
- Connect the client app to the mobile backend running in Azure.

- Implement authentication against Azure Active Directory in the client app.
- Modiy the client app to enable offline operation, storing data locally in SQLite, and then re-synchronizing with the mobile backend when connectivity is restored.

System requirements

You must have the following to complete this lab:

- Microsoft Windows 10
- Microsoft Visual Studio 2015

Optional add-ons

If you wish to complete the optional steps in this lab, you will need:

- Windows 10 Mobile Emulator or a physical phone running Windows 10 Mobile

---

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. Choose a custom install and ensure that the Universal Windows App Development Tools are selected from the optional features list.
3. Optional: Install the Windows 10 Mobile Emulator.

Exercises

This Hands-on lab includes the following exercises:

1. Getting Started with Azure App Service Mobile Apps
2. Enable authentication for your app and connect to the Cloud
3. Enable offline sync for your app

Estimated time to complete this lab: **45 to 75 minutes**.

Exercise 1: Getting Started with Azure App Service Mobile Apps

In this exercise, you will run a Windows 10 UWP client app which is similar to the ToDo app you can download from the Azure Portal when you create an Azure mobile app backend. You will connect the client app to Azure mobile app backend which you will run locally on your PC in the Azure development environment. You will connect to the real Azure backend running in the cloud in the following exercise.

Task 1 – Understand the Azure Portal support for Azure

We will begin by understanding how to create an Azure App Service Mobile App backend in the Azure Portal.

1. You do not need to create an Azure App Service mobile app backend for this hands-on lab as we have pre-created one for you.
   If you want to learn how to create a new Azure mobile app backend, follow the instructions in the section **Create a new Azure mobile app backend** at https://azure.microsoft.com/en-us/documentation/articles/app-service-mobile-dotnet-backend-windows-store-dotnet-get-started-preview/ .

   That tutorial walks you through how to use the wizard in Azure Portal to create a new Azure mobile app backend:

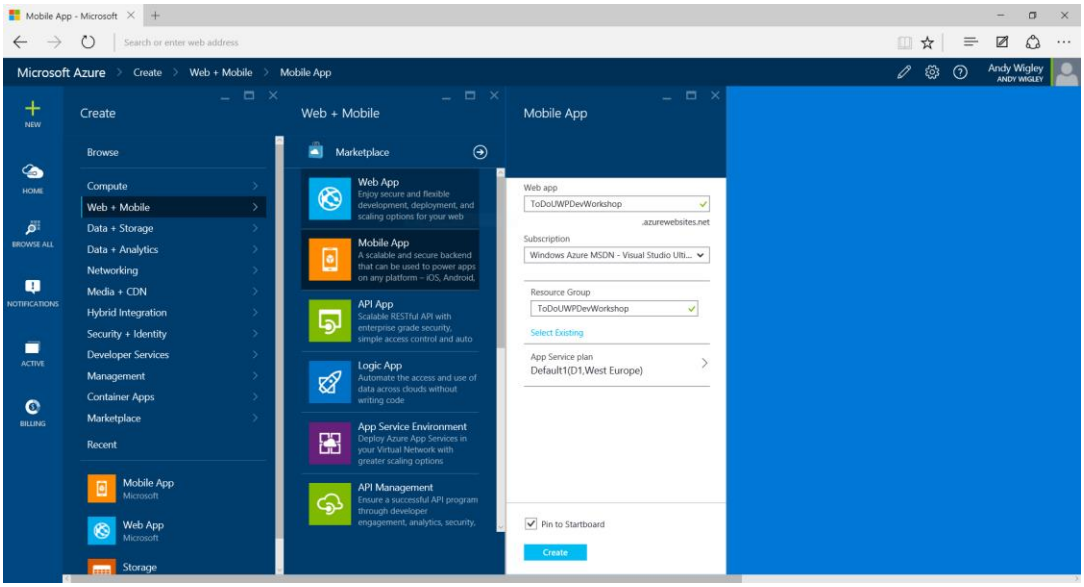

Figure 1

Starting to create a new Azure mobile app backend in Azure Portal.

2. If you had created your own Azure mobile apps backend, you would see a blade in the Azure Portal with information about your mobile app backend similar to this:

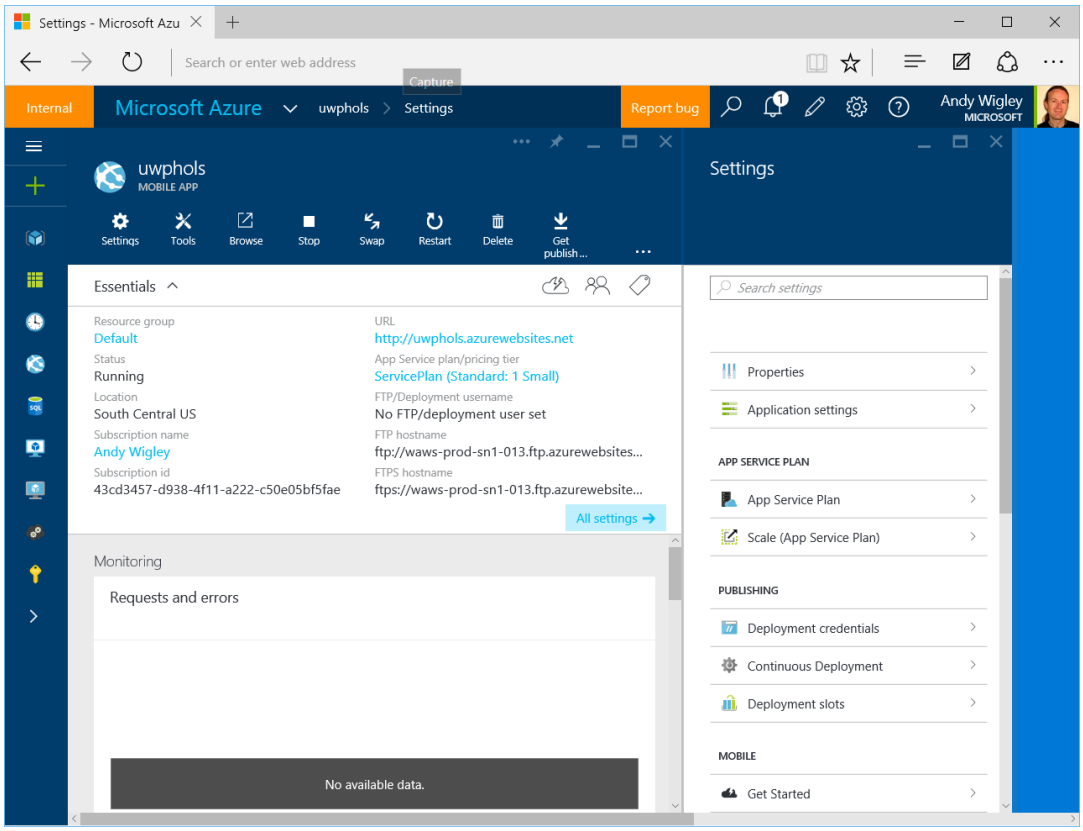

Figure 2

The blade for an Azure mobile app backend in Azure Portal.

3. On the **Settings** pane, you will find a section called Mobile containing a number of options including **Get Started**. If you were to click on this, you would open the **Quickstart** pane where you can get instructions on connecting an existing app to your Azure mobile apps backend, or you can download projects for an example server project and an example mobile app client. You can choose between a number of different clients, including Windows (C#), iOS (Objective-C), iOS (Swift), HTML/JavaScript, Xamarin.Android, Xamarin.iOS or Xamarin.Forms.

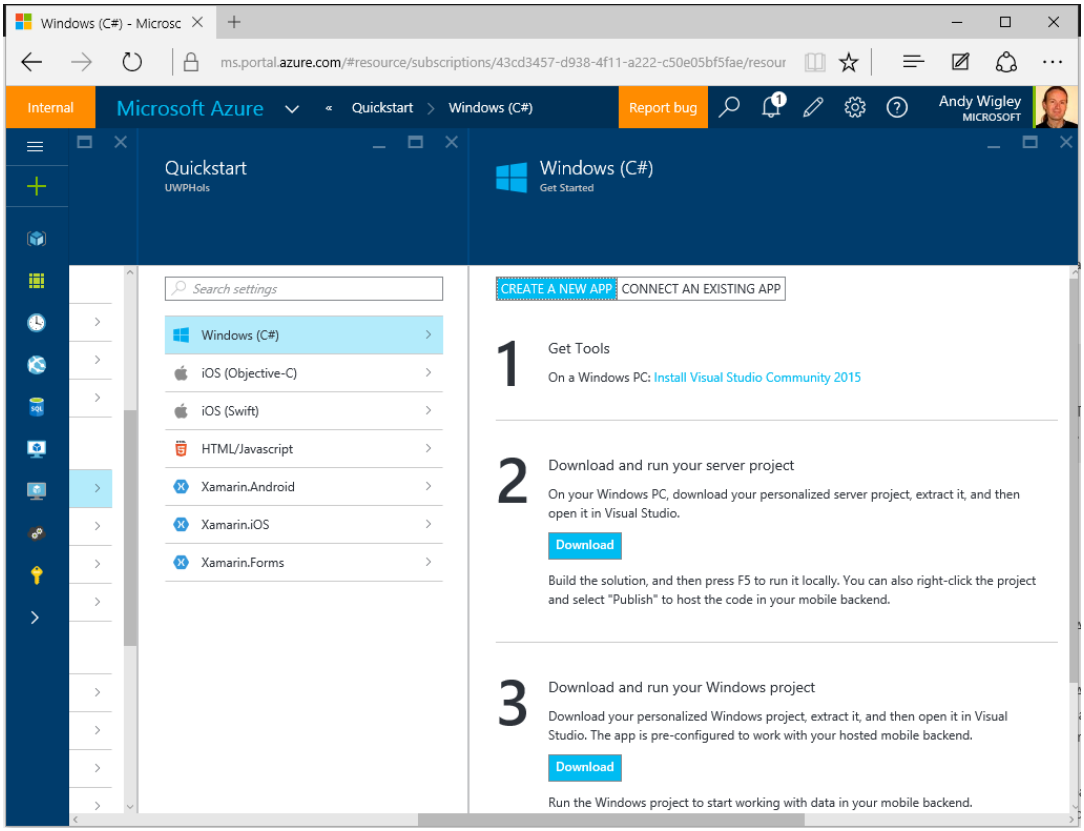

Figure 3

Azure mobile app Quickstart options in Azure Portal.

4. At the time of this writing (September 2015), the Quickstart download for Windows (C#) is a Windows 8.1 universal app project. For the purposes of this lab, we have created a UWP version of this client.

Task 2 – Run the server project on your PC

We will examine the server code which was downloaded when the Azure mobile apps backend that you will use later in this lab was created. You will then run it locally on your PC.

1. Open **Visual Studio 2015**. On the **File** menu, click **Open Project/Solution**. Navigate to the folder where you have saved the code for this lab, and open **Exercise 1\Begin-Cloud\UWPHolsService.sln**.

2. Build the project to restore the NuGet packages.

> **Note:** This project contains the C# code for a .NET backend to your Azure mobile app backend. You can instead choose to have your backend logic implemented in NodeJS. See the Azure App Service mobile apps documentation for more information.

3. This project contains the C# code for the backend logic. In the **DataObjects** folder, open **ToDoItem.cs**. This class defines the data objects that the cloud service stores in a SQL Azure database:

C#

```
public class TodoItem : EntityData
{
    public string Text { get; set; }

    public bool Complete { get; set; }
}
```

4. In the **Controllers** folder, open **TodoItemController.cs**. This class contains the code that implements the CRUD operations offered by the cloud service REST API:

C#

```
public class TodoItemController : TableController<TodoItem>
{
    protected override void Initialize(
                                HttpControllerContext controllerContext)
    {
        base.Initialize(controllerContext);
        UWPHolsContext context = new UWPHolsContext();
        DomainManager =
           new EntityDomainManager<TodoItem>(context, Request);
    }

    // GET tables/TodoItem
    public IQueryable<TodoItem> GetAllTodoItems()
    {
        return Query();
    }

    // GET tables/TodoItem/48D68C86-6EA6-4C25-AA33-223FC9A27959
    public SingleResult<TodoItem> GetTodoItem(string id)
    {
        return Lookup(id);
    }

    // PATCH tables/TodoItem/48D68C86-6EA6-4C25-AA33-223FC9A27959
    public Task<TodoItem> PatchTodoItem(string id, Delta<TodoItem> patch)
    {
        return UpdateAsync(id, patch);
    }

    // POST tables/TodoItem
    public async Task<IHttpActionResult> PostTodoItem(TodoItem item)
    {
```

```
            TodoItem current = await InsertAsync(item);
            return CreatedAtRoute("Tables", new { id = current.Id }, current);
        }

        // DELETE tables/TodoItem/48D68C86-6EA6-4C25-AA33-223FC9A27959
        public Task DeleteTodoItem(string id)
        {
            return DeleteAsync(id);
        }
    }
```

5. In the **App_Start** folder, open file **Startup.MobileApp.cs**. This file contains configuration code for the service, including the **Seed** method in the **UWPHolsInitializer** class. This method runs the first time a request is made to the REST service, and inserts the two data items created in this method into the database (just for the purposes of this Quickstart):

C#

```
    protected override void Seed(UWPHolsContext context)
    {
        List<TodoItem> todoItems = new List<TodoItem>
        {
            new TodoItem { Id = Guid.NewGuid().ToString(),
                           Text = "First item", Complete = false },
            new TodoItem { Id = Guid.NewGuid().ToString(),
                           Text = "Second item", Complete = false },
        };

        foreach (TodoItem todoItem in todoItems)
        {
            context.Set<TodoItem>().Add(todoItem);
        }

        base.Seed(context);
    }
```

6. Set your Solution Configuration to Debug and your Solution Platform to Any CPU. Select Microsoft Edge from the Debug Target dropdown next to the Start Debugging Button.

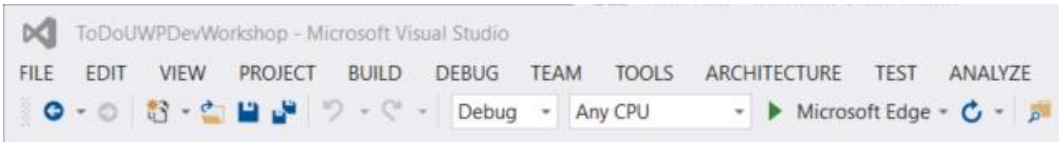

Figure 4

Configure your website to run on the Local Machine.

> **Note:** ▶ is the Start Debugging button.

7. Build and run your app. You will see a web browser window showing the default 'running' information screen for the Azure mobile app backend.

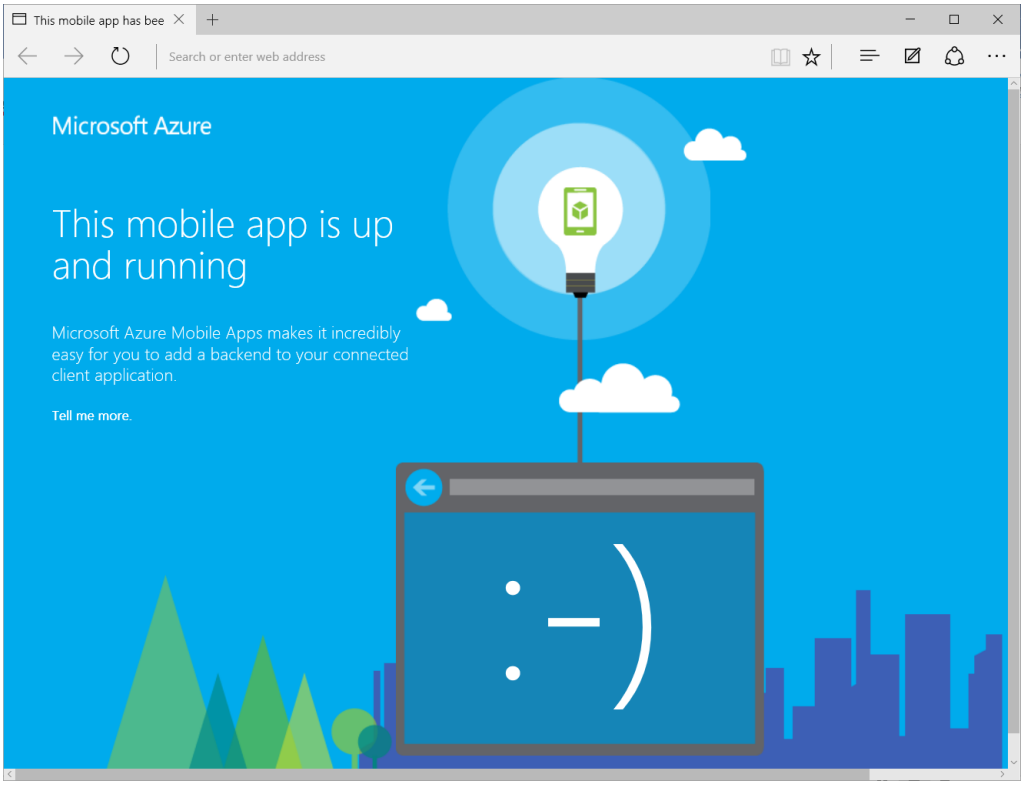

Figure 5

The Azure mobile app backend running locally on your PC.

> **Note:** Running your backend service locally gives you one way of debugging problems with your application.

Task 3 – Open and run the client app project

We will examine the Windows UWP app code that connects to an Azure mobile app backend and will run it configured to connect to the backend service that is running locally on your PC.

1. Open a second instance of **Visual Studio 2015**. On the **File** menu, click **Open Project/Solution**. Navigate to the folder where you have saved the code for this lab, and open **Exercise 1\Begin-Client\ToDoUWPDevWorkshop.sln**.

2. Set your Solution Configuration to **Debug** and your Solution Platform to **x86**. Select **Local Machine** from the Debug Target dropdown next to the Start Debugging Button.

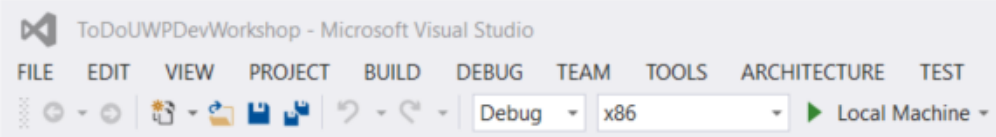

Figure 6

Configure your client app to run on the Local Machine.

3. Build the project to restore the NuGet packages.

4. Open **App.xaml.cs**. Notice how at the top of this class, there is code to instantiate a **MobileServiceClient** object. The first declaration is the form you use to connect to the cloud – leave this commented out. The second declaration that is NOT commented out is the form that connects to the local version of the cloud service that you just started running in the previous steps.

C#

```
    sealed partial class App : Application
    {

        // Uncomment this code for configuring the MobileServiceClient to
        // communicate with the Azure Mobile Service and
        // Azure Gateway using the application key. You're all set to start
        // working with your Mobile Service!
        //public static MobileServiceClient MobileService =
        //    new MobileServiceClient(
        //    "https://uwpdevhols.azurewebsites.net",
        //    "",
        //    ""
        //);

        // Use this code for configuring the MobileServiceClient to
        // communicate with your local
        // test project for debugging purposes.
        public static MobileServiceClient MobileService =
            new MobileServiceClient(
            "http://localhost:50781"
        );
    ...
```

5. Open **MainPage.xaml.cs**. In this class you can see the way that your application code communicates with the cloud backend service to download objects from the service and to insert, update and delete objects.
   At the top of the class, the field **todoTable** of type **IMobileServiceTable<TodoItem>** is initialized by calling **App.MobileService.GetTable<TodoItem>()**. This object is used throughout the class to perform strongly typed data operations for that table.
   For example, to insert a new data item, you use the **InsertAsync** method of that **todoTable** object:

C#

```
    private IMobileServiceTable<TodoItem> todoTable =
         App.MobileService.GetTable<TodoItem>();

    ...

    private async Task InsertTodoItem(TodoItem todoItem)
    {
        // This code inserts a new TodoItem into the database. When the
        // operation completes and Mobile Services has assigned an Id,
        // the item is added to the CollectionView
        await todoTable.InsertAsync(todoItem);
        items.Add(todoItem);

    }
    ...
```

6. Build and run your app. When your client app runs, code in the **OnNavigatedTo** method of MainPage calls **RefreshTodoitems** which calls your backend service to retrieve any ToDo items stored in the backend database.

> **Note**: If your app fails with an HttpRequestException, it is probably because you stopped debugging your UWPHolsService at the end of the previous task. Run the backend service project again, leave it running, and then run the client app in a separate copy of Visual Studio.

7. As this is the first time the service has been accessed, it will automatically configure the database and run the Seed method that you looked at previously, which inserts two items. After a few moments, you should see the two items displayed in the client app UI.

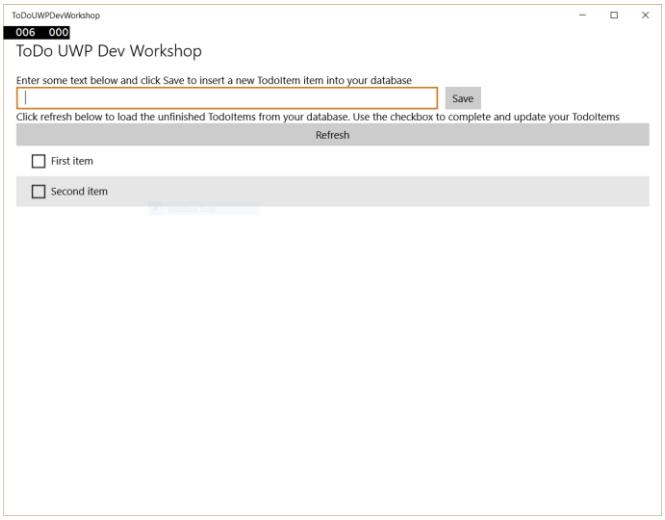

Figure 7

The client app retrieves the two 'Seed' data items from the backend service.

8. Insert some new items in addition to the two items that were added by the Seed method in the service.

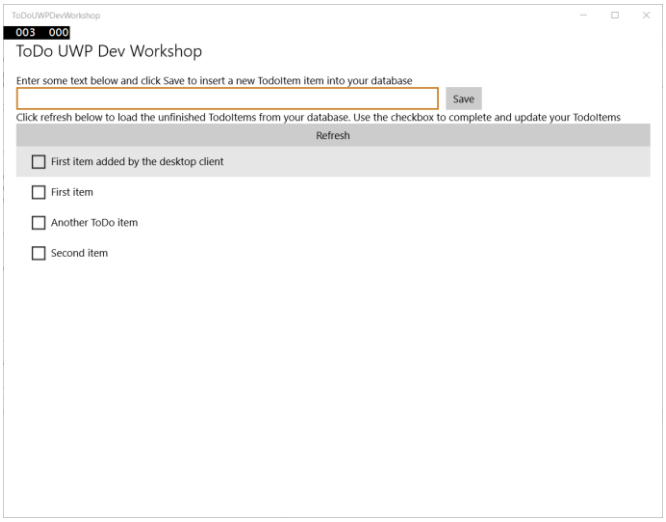

Figure 8

Adding new data items.

9. Stop debugging.

10. If your development PC has the Windows 10 Mobile emulators installed, select one of the **Mobile Emulator …** options from the Debug Target dropdown next to the Start Debugging Button. If you have a real phone device connected and enabled for Developer Mode, select **Device** from the Debug Target dropdown. If neither of these options are available, you will have to skip the rest of this exercise.

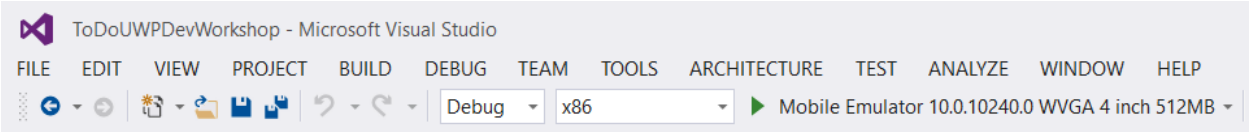

Figure 9

Configure your debug target to a Mobile emulator or a real device, if available.

11. Start debugging. The app will start on your Mobile device or emulator and show the same items you entered into the desktop client and which you stored in your mobile backend service database (which is running locally on your PC at present).

Figure 10

The client app running on a Windows 10 Mobile device uses the same backend service.

12. Stop debugging in both the client app project and also in the service project.

13. You have completed this exercise, and have run a mobile app over two different devices which shares data through a mobile backend service. You have created a *connected mobile experience* for the users of this app.

Exercise 2: Enable authentication for your app and connect to the Cloud

This exercise shows you how to add authentication functionality to a UWP app that is connected to an Azure Mobile App backend. After being successfully authenticated and authorized by your Mobile App, the user ID value is displayed. The UWP app is slightly different from the one you used in exercise 1 as it is configured to connect to an Azure App Service Mobile App service running in Microsoft Azure instead of to one running locally. The Azure Mobile App service has already been configured to support authentication against Azure Active Directory.

Task 1 – Add authentication to the app

We will modify the UWP app to authenticate users before requesting resources from your App Service.

1. Open **Visual Studio 2015**. On the **File** menu, click **Open Project/Solution**. Navigate to the folder where you have saved the code for this lab, and open **Exercise 2\Begin-Client\ToDoUWPDevWorkshop.sln**.

2. Set your Solution Configuration to **Debug** and your Solution Platform to **x86**. Select **Local Machine** from the Debug Target dropdown next to the Start Debugging Button.

3. Build the project to restore the NuGet packages.

4. Open **App.xaml.cs**. In this exercise, the code to instantiate a **MobileServiceClient** object has been changed so that the first declaration is not commented out – this is the form you use to connect to the cloud. The second declaration is the form that connects to the local version of the cloud service you used in the previous exercise – this is now commented out.

C#

```
    sealed partial class App : Application
    {

        // Uncomment this code for configuring the MobileServiceClient to
        // communicate with the Azure Mobile Service and
        // Azure Gateway using the application key. You're all set to start
        // working with your Mobile Service!
        public static MobileServiceClient MobileService =
            new MobileServiceClient(
            "https://uwpdevhols.azurewebsites.net",
          "https://default43cd3457d9384f11a222c50e05bf5fae.azurewebsites.net",
            ""
        );

        // Use this code for configuring the MobileServiceClient to
        // communicate with your local
        // test project for debugging purposes.
        //public static MobileServiceClient MobileService =
        //    new MobileServiceClient(
        //    "http://localhost:59989"
        //);
    ...
```

5. In a browser, go to the url of the Azure App Service Mobile App: https://uwpdevhols.azurewebsites.net . The standard 'Running' page should display indicating that the service is up and running in the cloud.

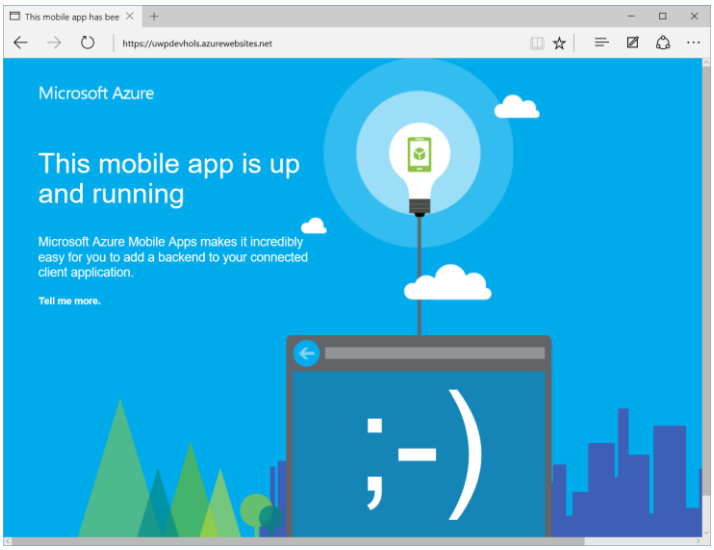

Figure 11

The Azure mobile app service running in Microsoft Azure.

6. This cloud service differs slightly from the one you ran locally in the previous exercise. It has been configured to require that all access is through authenticated users. To demonstrate this, run your **TodoUWPDevWorkshop** client app. The app will fail to load items from the service because the user of the app has not authenticated against a supported identity provider.

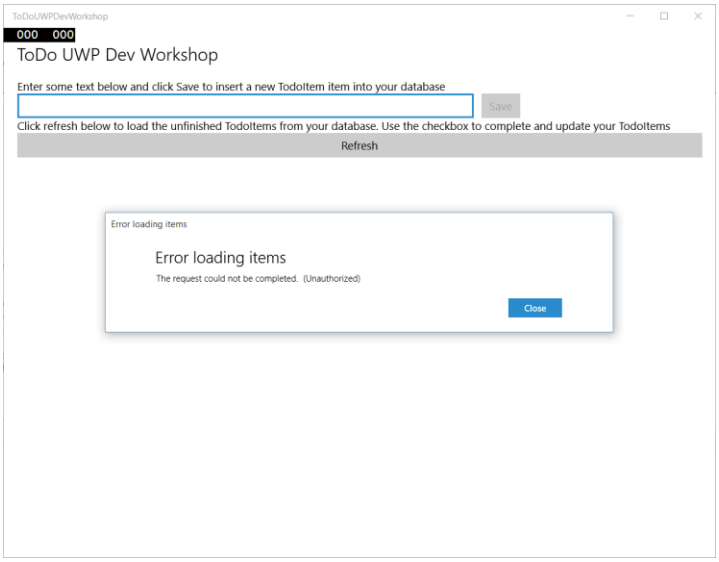

Figure 12

The Azure mobile app service requires authenticated user access.

Note: Azure App Service Mobile App supports authentication against - Azure Active Directory - Facebook - Google

- Microsoft
- Twitter

The service you are connecting to has been configured to support authentication against Azure Active Directory.

To follow step by step instructions on how to configure one of these identity providers in an Azure App Service Mobile App, see the tutorial https://azure.microsoft.com/en-us/documentation/articles/app-service-mobile-dotnet-backend-windows-store-dotnet-get-started-users-preview/ , topic **Authenticate Users**.

7. You will now add authentication to the app. Open **MainPage.xaml.cs**. Define a member variable for storing the signed-in user, and a method for performing the authentication process.

C#

```
// Define a member variable for storing the signed-in user.
private MobileServiceUser user;

// Define a method that performs the authentication process
// using an Azure Active Directory sign-in.
private async System.Threading.Tasks.Task AuthenticateAsync()
{
    while (user == null)
    {
        string message;
        // This sample uses the Azure Active Directory provider.
        var provider = "AAD";

        try
        {
            // Sign-in using AAD authentication.
            user = await App.MobileService.LoginAsync(provider);
            message =
                string.Format("You are now signed in - {0}", user.UserId);
        }
        catch (InvalidOperationException)
        {
            message = "You must log in. Login Required";
        }

        var dialog = new MessageDialog(message);
        dialog.Commands.Add(new UICommand("OK"));
        await dialog.ShowAsync();
    }
}
```

8. Comment-out or delete the call to the **RefreshTodoItems** method in the existing **OnNavigatedTo** method override. This prevents the data from being loaded before the user is authenticated. Next, you will add a Sign in button to the app that triggers authentication.

9. Add the following method to the **MainPage** class:

C#

```
private async void ButtonLogin_Click(object sender, RoutedEventArgs e)
{
    // Login the user and then load data from the mobile app.
    await AuthenticateAsync();

    // Hide the login button and load items from the mobile app.
    this.ButtonLogin.Visibility = Windows.UI.Xaml.Visibility.Collapsed;
    await RefreshTodoItems();
}
```

10. Open the **MainPage.xaml** project file and add the following **Button** element into the **StackPanel** in Row 0 of the Grid, just before the **TextBlock** element:

XAML

```
<StackPanel Grid.Row="0" Grid.ColumnSpan="2" >

    <Button Name="ButtonLogin" Click="ButtonLogin_Click"
        Visibility="Visible">Sign in</Button>

    <TextBlock Style="{StaticResource BodyTextBlockStyle}" Text="Enter some
text below and click Save to insert a new TodoItem item into your database"
TextWrapping="Wrap"/>
</StackPanel>
```

11. Before running your app, you need to find out which username and password to use when you authenticate against Azure Active Directory. A custom AAD has been created just for these labs which has been provisioned with 500 users. To find out which username and password to use, go to this website: http://uwpholsusers.azurewebsites.net/ . You will see a page similar to this, telling you the username and password to use to authenticate against AAD

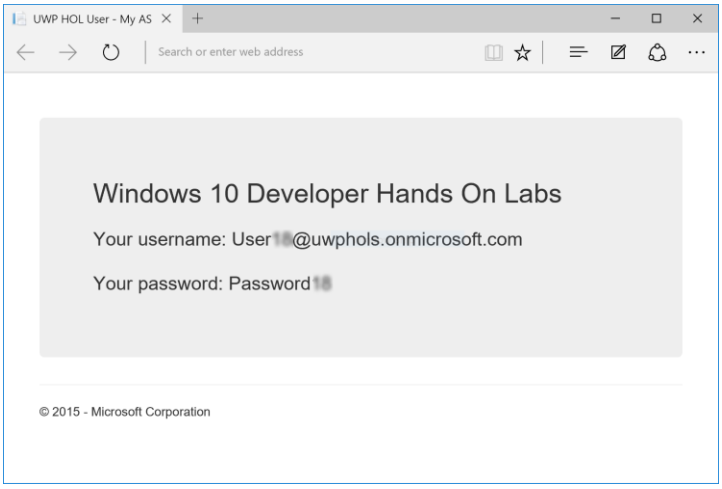

Figure 13

Getting the credentials to use.

> **Note**: Every request to this webpage returns a different username and password from the 500 that have been pre-provisioned for this lab. You should have sole usage of this username for the duration of this lab, but this is not guaranteed. Since the user identity is used to partition data for each user in the SQL Azure database, if someone else is using the same username and password on the same day that you work through this lab, you will see ToDo items created by the other person(s).
> Of course, in a real application, credentials should never be shared in this way!

12. Press the **F5** key to run the app, click the **Sign in** button, and sign into the app using your credentials.

13. When you are successfully logged-in, the app should run without errors, and you should be able to query your Mobile App and make updates to data.

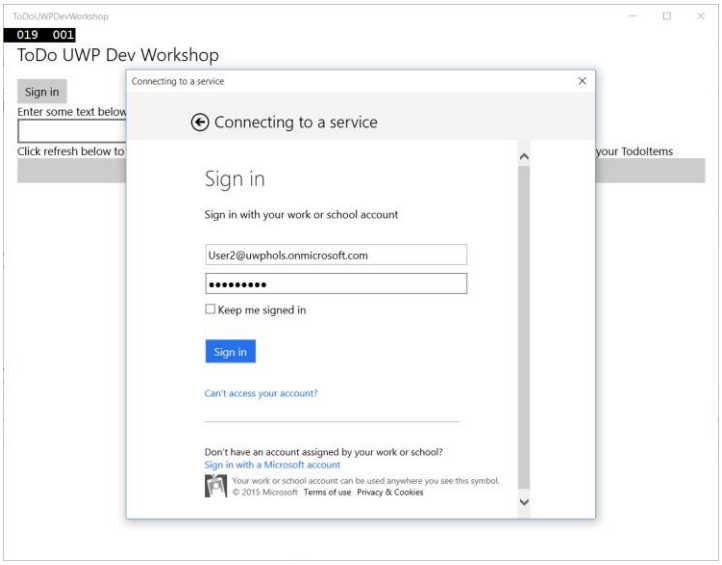

Figure 14

Authenticating against Azure Active Directory.

Note: The Azure App Service Mobile App you are connected to is shared across all users of the service. All data for all users is stored in the same SQL Azure database, and all users access the same REST service hosted in Azure.

To support multiple different users, a few changes were required to the code in the cloud service compared to the local service you ran in exercise 1.

You can examine the code that is running in the service you are connecting to by opening **Exercise 2\Cloud\UwpDevHols.sln**. The changes to support separation by users are:

- In **DataObjects\TodoItem.cs**: addition of a **UserId** field to store the identity of the authenticated user against every ToDo item.
- In **Controllers\TodoItemController.cs**: in method **GetAllTodoItems** code to get the current User ID and to use that to change the query to only return Todo items that are associated with that user; and in method **PostTodoItem**, logic to get the current User identity and to store that in the UserId field of every new Todo item stored.
- Addition of the **[Authorize]** attribute to the **TodoItemController** class, which restricts access to only authenticated users.

Task 2 – Store the authentication token on the client

The previous task showed a standard sign-in, which requires the client to contact both the identity provider and the App Service every time that the app starts. Not only is this method inefficient, you can

run into usage-relates issues should many customers try to start your app at the same time. A better approach is to cache the authorization token returned by your App Service and try to use this first before using a provider-based sign-in.

1. Continue with the **ToDoUWPDevWorkshop.sln** from the previous task.

2. In **MainPage.xaml.cs**, add the following **using** statements:

C#

```
using System.Linq;
using Windows.Security.Credentials;
```

3. Replace the **AuthenticateAsync** method with the following code:

C#

```
// Define a method that performs the authentication process
// using an Azure Active Directory sign-in.
private async System.Threading.Tasks.Task AuthenticateAsync()
{
    string message = string.Empty;
    // This sample uses the Azure Active Directory provider.
    var provider = "AAD";

    // Use the PasswordVault to securely store and access credentials.
    PasswordVault vault = new PasswordVault();
    PasswordCredential credential = null;

    while (credential == null)
    {
        try
        {
            // Try to get an existing credential from the vault.
            credential = vault.FindAllByResource(provider).FirstOrDefault();
        }
        catch (Exception)
        {
            // When no matching resource an error occurs, which we ignore.
        }

        if (credential != null)
        {
            // Create a user from the stored credentials.
            user = new MobileServiceUser(credential.UserName);
            credential.RetrievePassword();
            user.MobileServiceAuthenticationToken = credential.Password;

            // Set the user from the stored credentials.
            App.MobileService.CurrentUser = user;

            try
```

```
            {
                // Try to return an item now to determine if the
                // cached credential has expired.
                await App.MobileService.GetTable<TodoItem>().Take(1)
                      .ToListAsync();
            }
            catch (MobileServiceInvalidOperationException ex)
            {
                if (ex.Response.StatusCode ==
                        System.Net.HttpStatusCode.Unauthorized)
                {
                    // Remove the credential with the expired token.
                    vault.Remove(credential);
                    credential = null;
                    continue;
                }
            }
        }
        else
        {
            try
            {
                // Login with the identity provider.
                user = await App.MobileService
                    .LoginAsync(provider);

                // Create and store the user credentials.
                credential = new PasswordCredential(provider,
                    user.UserId, user.MobileServiceAuthenticationToken);
                vault.Add(credential);
            }
        catch (InvalidOperationException)
        {
            message = "You must log in. Login Required";
        }

        var dialog = new MessageDialog(message);
        dialog.Commands.Add(new UICommand("OK"));
        await dialog.ShowAsync();
    }
}
```

In this version of **AuthenticateAsync**, the app tries to use credentials stored in the PasswordVault to access the Mobile App. A simple query is sent to verify that the stored token is not expired. When a 401 is returned, a regular provider-based sign-in is attempted. A regular sign-in is also performed when there is no stored credential.

Note: This app tests for expired tokens during login, but token expiration can occur after authentication when the app is in use. For a solution to handling authorization errors related to expiring tokens, see the post Caching and handling expired tokens in Azure Mobile Services managed SDK.

4. Restart the app twice.
   Notice that on the first start-up, sign-in with the provider is again required. However, on the second restart the cached credentials are used and sign-in is bypassed.

5. **[Optional]** A great feature of **PasswordVault** is that credentials that you store in it are roamed automatically through the cloud to all your devices where you have the same app installed. If you have a second device running Windows 10 or Windows 10 Mobile, such as a Phone, tablet or PC, and that device is associated with the same Microsoft account as your development PC, you can try this out by installing the same UWP app onto that device.
   If you wait for a while so that the credential roaming has had time to complete, when you log onto the app on one of your other devices, it will find the same stored credentials in **PasswordVault** and you will not have to log in again, unless of course the access token has expired.

Exercise 3: Enable offline sync for your app

This exercise shows you how to add offline support to a UWP app using an Azure Mobile App backend. Offline sync allows end-users to interact with a mobile app--viewing, adding, or modifying data--even when there is no network connection. Changes are stored in a local database; once the device is back online, these changes are synced with the remote backend.

In this exercise, you will update the Windows app project to support the offline features of Azure Mobile Apps.

Task 1 – Update the client app to support offline features

Azure Mobile App offline features allow you to interact with a local database when you are in an offline scenario. To use these features in your app, you initialize a **MobileServiceClient.SyncContext** to a local store. Then reference your table through the **IMobileServiceSyncTable** interface. In this tutorial we use SQLite for the local store.

The first task is to open the ToDoUWPDevWorkshop solution you created in the previous exercise.

1. Navigate to the file location where you saved your **ToDoUWPDevWorkshop** app in Exercise 2. Open **ToDoUWPDevWorkshop.sln** in Visual Studio 2015.

2. Install the SQLite runtime for Universal Windows Platform.

a. In Visual Studio, on the **Tools** menu, click **Extensions and Updates**

b. In the left pane of the Extensions and Updates wizard, click **Online**

c. In the search box at the top right of the window, enter **SQLite**

d. When the Search Results display, scroll down until you see **SQLite for Universal App Platform**. If this SDK is not already installed on your system, select this item, and then click the **Download** button

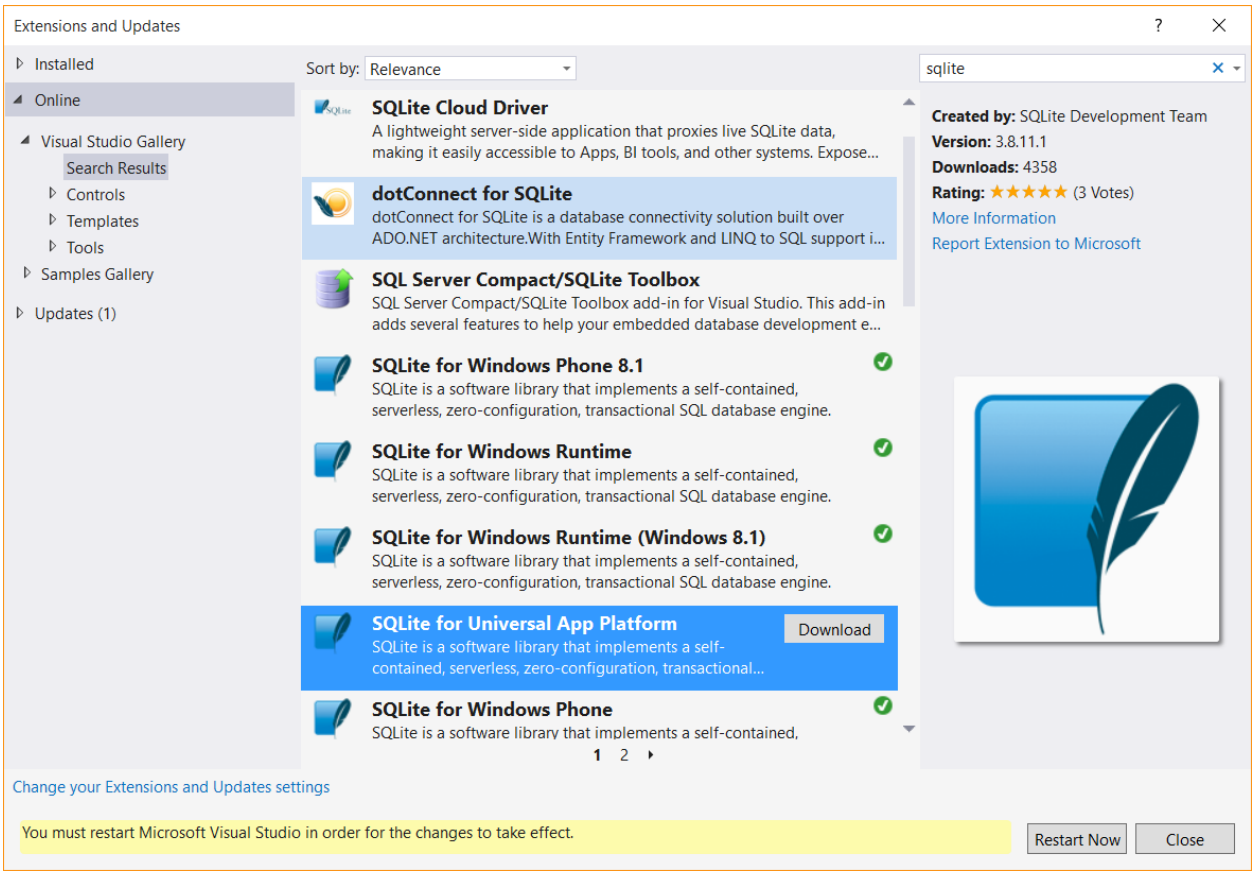

Figure 15

Download and install the SQLite for Universal App Platform SDK.

e. When the UAC prompt displays, click **OK.**

f. In the VSIX Installer window, click **Install**. After the extension installs, click **Close**.

g. Click the **Restart Now** button on the Extensions and Updates window and wait for Visual Studio 2015 to restart.

3. Add a reference to the SQLite runtime dll to your project.

a. In Solution Explorer, right click the References node in the project tree and click **Add Reference** to run the Reference Manager.

b. In the "Universal Windows" category, select the option "Extensions" in the navigation pane at the left.

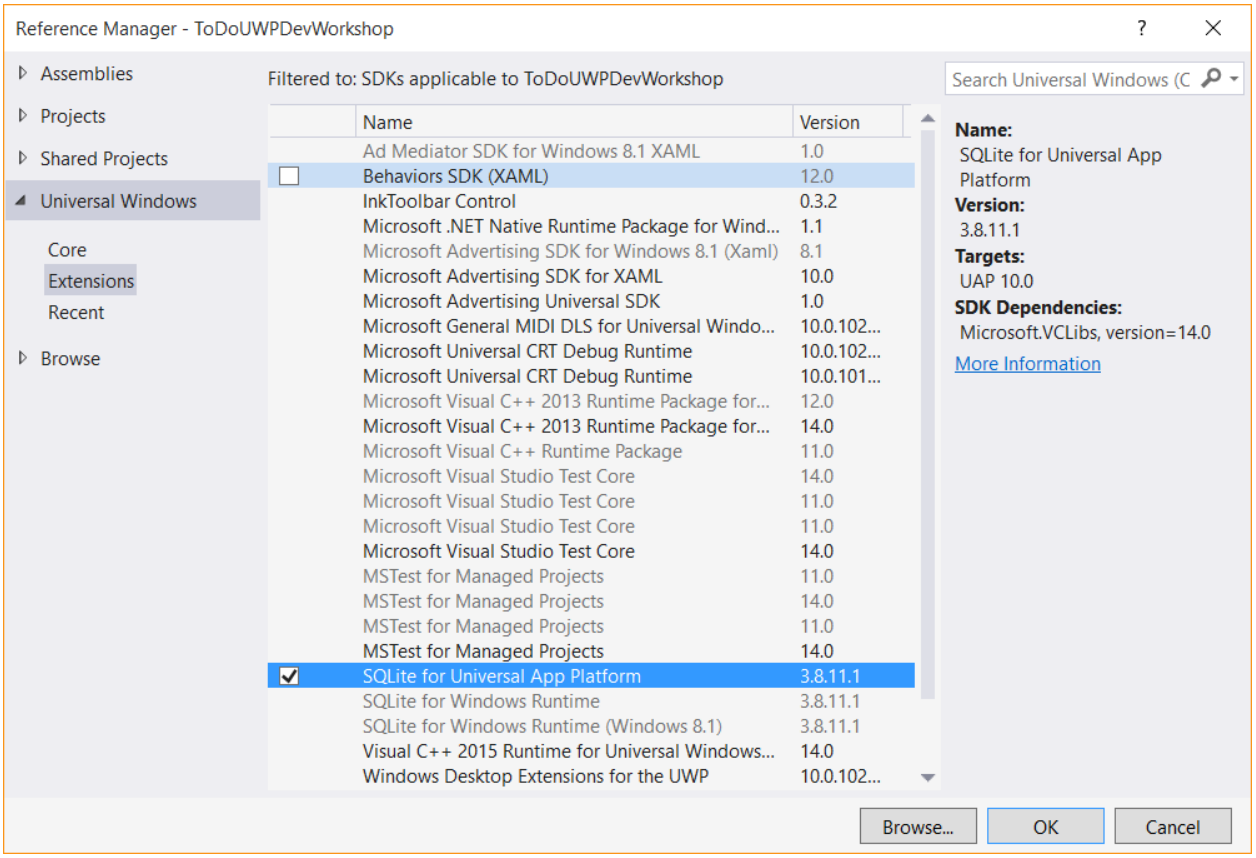

Figure 16

Add a reference to the SQLite for Universal App Platform dll to your project.

c. Select **SQLite for Universal App Platform**, and then click **OK**.

4. Install the WindowsAzure.MobileServices.SQLiteStore NuGet package.

a. In Solution Explorer, right click the project and click **Manage Nuget Packages** to run NuGet Package Manager.

b. In the "Online" tab, select the option "Include Prerelease" in the dropdown at the top. Search for **SQLiteStore** to locate the 2.0.0-beta of **WindowsAzure.MobileServices.SQLiteStore**.

c. Then, click **Install** to add the NuGet reference to the project.

d. Click **I Accept** on the License Acceptance window.

5. In Solution Explorer, open the **MainPage.cs** file. Uncomment the following using statements at the top of the file:

C#

```
using Microsoft.WindowsAzure.MobileServices.SQLiteStore;  // offline sync
using Microsoft.WindowsAzure.MobileServices.Sync;         // offline sync
```

6. In MainPage.cs, comment the line of code that initializes `todoTable` as an `IMobileServiceTable`. Uncomment the line of code that initializes `todoTable` as an `IMobileServiceSyncTable`:

C#

```
//private IMobileServiceTable<TodoItem> todoTable =
App.MobileService.GetTable<TodoItem>();
private IMobileServiceSyncTable<TodoItem> todoTable =
                App.MobileService.GetSyncTable<TodoItem>(); // offline sync
```

7. In **MainPage.cs**, in the region marked **`Offline sync`**, uncomment the methods **`InitLocalStoreAsync`** and **`SyncAsync`**. The method InitLocalStoreAsync initializes the client sync context with a SQLite store. In Visual Studio, you can select all commented lines and use the Ctrl+K+U keyboard shortcut to uncomment.

   Notice in **SyncAsync** a push operation is executed off the `MobileServiceClient.SyncContext` instead of the `IMobileServicesSyncTable`. This is because the context tracks changes made by the client for all tables. This is to cover scenarios where there are relationships between tables. For more information on this behavior, see Offline Data Sync in Azure Mobile Apps.

C#

```
private async Task InitLocalStoreAsync()
{
    if (!App.MobileService.SyncContext.IsInitialized)
    {
        var store = new MobileServiceSQLiteStore("localstore.db");
        store.DefineTable<TodoItem>();
        await App.MobileService.SyncContext.InitializeAsync(store);
    }

    await SyncAsync();
}

private async Task SyncAsync()
```

```
{
    await App.MobileService.SyncContext.PushAsync();
    await todoTable.PullAsync("todoItems", todoTable.CreateQuery());
}
```

Note: In this `PullAsync` example, we retrieve all records in the remote `todoTable`, but it is also possible to filter records by passing a query. The first parameter to `PullAsync` is a query ID that is used for incremental sync, which uses the `UpdatedAt` timestamp to get only records modified since the last sync. The query ID should be a descriptive string that is unique for each logical query in your client application. To opt-out of incremental sync, pass `null` as the query ID. This will retrieve all records on each pull operation, which is potentially inefficient.

8. In the `OnNavigatedTo` event handler, uncomment the call to `InitLocalStoreAsync`:

C#

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    await InitLocalStoreAsync(); // offline sync
    // await RefreshTodoItems();
}
```

9. Uncomment the 3 calls to `SyncAsync` in the methods `InsertTodoItem`, `UpdateCheckedTodoItem`, and `ButtonRefresh_Click`:

C#

```
private async Task InsertTodoItem(TodoItem todoItem)
{
    await todoTable.InsertAsync(todoItem);
    items.Add(todoItem);

    await SyncAsync(); // offline sync
}

...

private async Task UpdateCheckedTodoItem(TodoItem item)
{
    await todoTable.UpdateAsync(item);
    items.Remove(item);
    ListItems.Focus(Windows.UI.Xaml.FocusState.Unfocused);

    await SyncAsync(); // offline sync
}

private async void ButtonRefresh_Click(object sender, RoutedEventArgs e)
{
    ButtonRefresh.IsEnabled = false;
```

```
    await SyncAsync(); // offline sync
    await RefreshTodoItems();

    ButtonRefresh.IsEnabled = true;
}
```

10. Modify the code in the SyncAsync method to add exception handlers. In an offline situation a MobileServicePushFailedException will be thrown with PushResult.Status == CancelledByNetworkError.

C#

```
private async Task SyncAsync()
{
    String errorString = null;

    try
    {
        await App.MobileService.SyncContext.PushAsync();
        // first param is query ID, used for incremental sync
        await todoTable.PullAsync("todoItems", todoTable.CreateQuery());
    }

    catch (MobileServicePushFailedException ex)
    {
        errorString = "Push failed because of sync errors. " +
                      "You may be offine.\nMessage: " +
                      ex.Message + "\nPushResult.Status: " +
                      ex.PushResult.Status.ToString();
    }
    catch (Exception ex)
    {
        errorString = "Pull failed: " + ex.Message +
          "\n\nIf you are still in an offline scenario, " +
          "you can try your Pull again when connected with " +
          "your Mobile Service.";
    }

    if (errorString != null)
    {
        MessageDialog d = new MessageDialog(errorString);
        await d.ShowAsync();
    }
}
```

Note: The MobileServicePushFailedException can occur for both a push and a pull operation. It can occur for a pull because the pull operation internally executes a push to make sure all tables along with any relationships are consistent.

11. In Visual Studio, press the F5 key to rebuild and run the client app. The app will behave the same as it did before the offline sync changes, because it does a sync operation on the insert, update, and refresh operations. However, it will populate a local database which can be used in an offline scenario. We will test the offline scenario in the next section now that the local database is populated.

Task 2 – Testing the offline behavior of the client app

In this task, you will run the client app in an offline scenario. When you add data items, your exception handler will inform you that the app is operating in offline mode with **PushResult.Status == CancelledByNetworkError**. Items added will be held in the local store, but not synced to the mobile app backend until you are online again and execute a successful push to the Azure Mobile App backend.

1. Make changes to ensure the app runs in an offline state:

    a. **Option 1:** To test offline operation using the app running on your PC, click the Notifications icon near the bottom right of your Task Bar to open Notification Center. From the Quick tasks buttons, click **Flight Mode** to put the device offline.

    b. **Option 2**: Change your debug target to be a Windows 10 Mobile emulator. Deploy the app to the emulator. When the emulator is running, click the **>>** button on the command bar to show the emulator additional features window.
    On the **Network** tab, check **Enable network simulation** and set **Network speed** to **No network**.

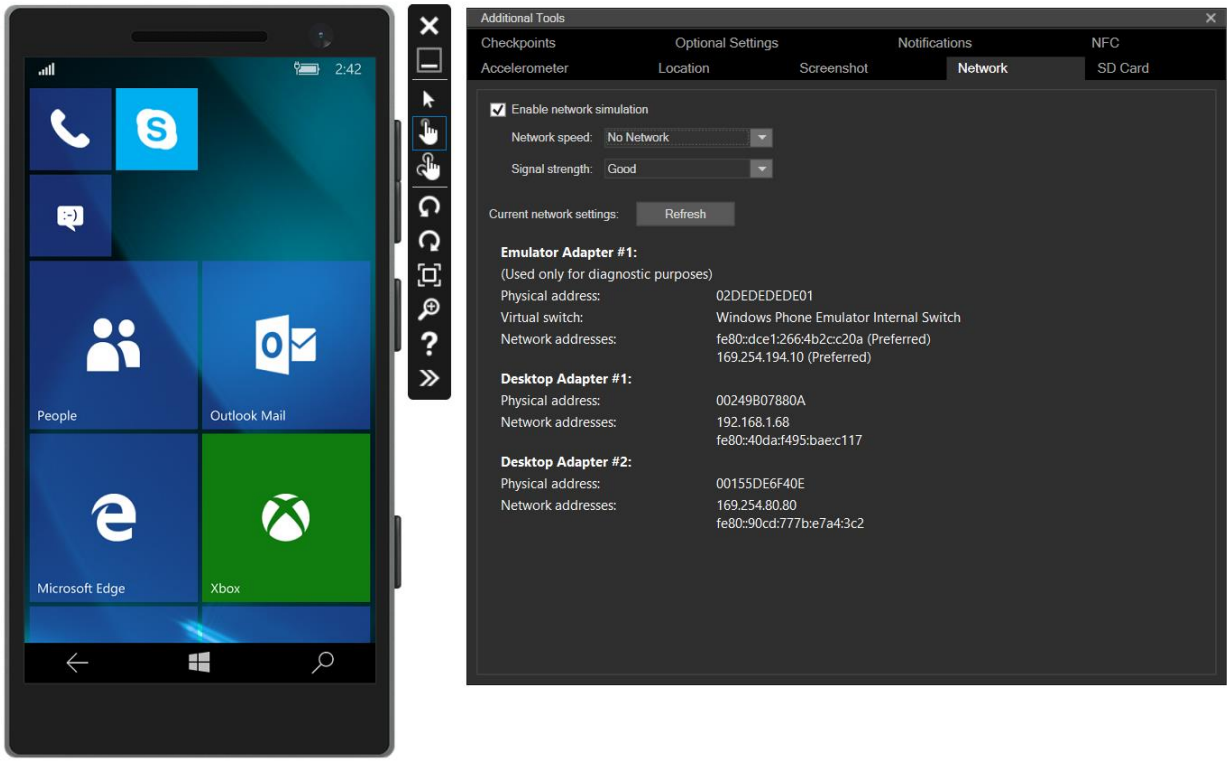

Figure 17

Setting the Windows 10 Mobile emulator to operate offline.

2. Press F5 to build and run the app. Notice your sync failed on refresh when the app launched.

3. Enter some new todo items and click **Save** for each one. Push fails for each one with a **PushResult.Status=CancelledByNetworkError**. The new todo items exist only in the local store until they can be pushed to the mobile app backend.
   You could suppress the exception dialog for **PushResult.Status=CancelledByNetworkError**, then client app would behave as if its connected to the mobile app backend supporting all create, read, update, delete (CRUD) operations seamlessly.

4. Close the app and restart it to verify that the new items you created previously are still visible in the app because they were persisted to the local store.

Task 3 – Reconnect the client app to the mobile app backend

In this task you reconnect the app to the mobile app backend. The app moves from an offline state to an online state with the mobile app backend. When you first run the application, the **OnNavigatedTo** event handler will call **InitLocalStoreAsync**. This will in turn call **SyncAsync** to sync your local store with the backend database. So the app will attempt to sync on start up.

1. Put the app back into an online state.

a. **Option 1:** If using the desktop target, from the Notifications Center, click the **Flight Mode** button to turn off Flight Mode and put the device back online.

b. **Option 2**: If you are testing offline behavior using the Mobile emulator, click the **>>** button on the command bar to show the emulator additional features window. On the **Network** tab, uncheck **Enable network simulation**.

2. Press the **F5** key to rebuild and run the app. The app syncs your local changes with the Azure Mobile App backend using push and pull operations as soon as the **OnNavigatedTo** event handler executes.

3. **[Optional]** If you have another device with this app installed, launch the app on that device and sign in with the same credentials as previously, if prompted. Notice the Todo items that were previously created offline have now been synchronized to the Azure Mobile App backend database and are now available on this second device.

4. In the app, click the check box beside a few items to complete them in the local store.

   UpdateCheckedTodoItem calls SyncAsync to sync the complete each item with the Mobile App backend. SyncAsync calls both push and pull. However, you should note that **whenever you execute a pull against a table that the client has made changes to, a push on the client sync context will always be executed first automatically**. This is to ensure all tables in the local store, along with relationships remain consistent. So in this case we could have removed the call to PushAsync because it is executed automatically when executing a pull. This behavior may result in an unexpected push if you are not aware of it. For more information on this behavior, see Offline Data Sync in Azure Mobile Apps.

Summary

Azure App Service Mobile App is a . In this lab, you evolved a blank app created from a template into a Hello World app that displays device-specific information across all Windows 10 devices. You also learned how to leverage sample data in Blend to quickly start building and visualizing your app UI. In the next lab, you will learn how to navigate within a UWP app, handle back navigation with the shell-drawn back button, and implement custom back and forward navigation controls.