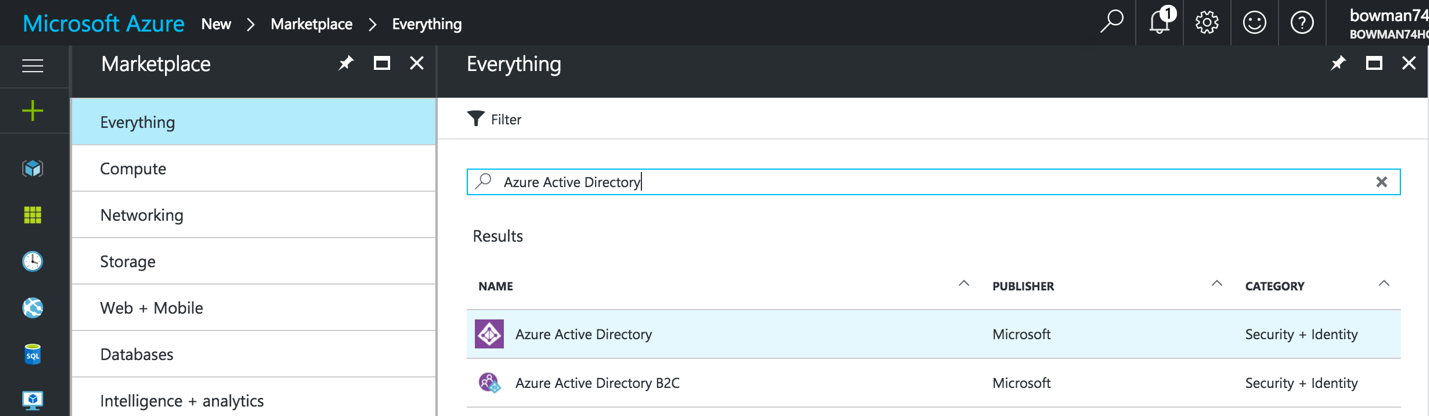
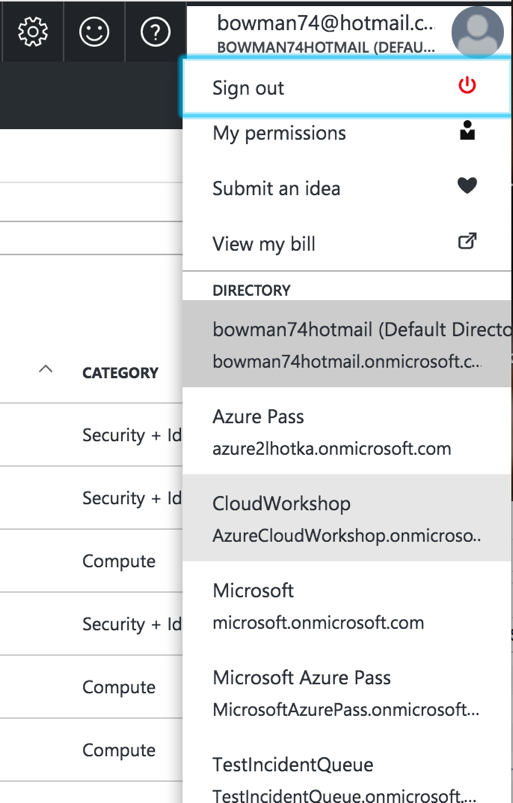
# Hands on Lab 1: Set up Azure AD and tie to Azure App Service

**Step 1:** Ensure you can log into the Azure Portal at:

<https://portal.azure.com>

If you cannot work with the version of Azure AD you have by default for whatever reason, create a new one. Most of you likely can use the default instance.

If you do need to create a new Azure active directory make sure you switch to it by clicking on your account information in the upper right.



**Step 2:** Ensure you have a GitHub Account.

<https://github.com/>

If you do not have an account click on Sign Up for Github.

**Step 3:** Fork the repository for this project into your own account.

<https://github.com/Bowman74/CloudDevelopmentWorkshop>

The fork button is in the upper right.

**Step 4:** Download the solutions from your forked repository. You can use Visual Studio for this or your favorite Git client like SourceTree.

**Step 5:** Create an account for Visual Studio Mobile Center.

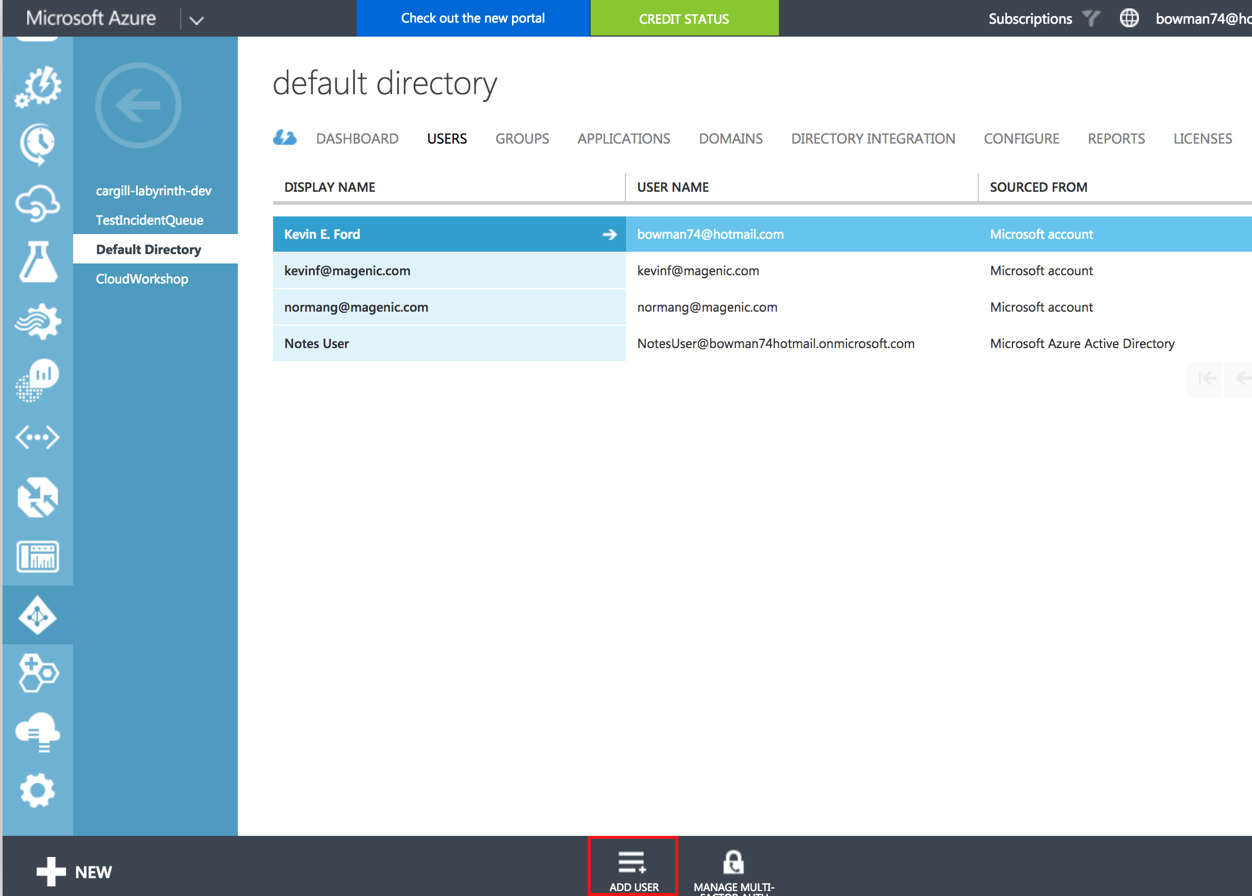
<https://www.visualstudio.com/vs/mobile-center/>

If you don’t have an account click on Get Started for Free to start a free trial.

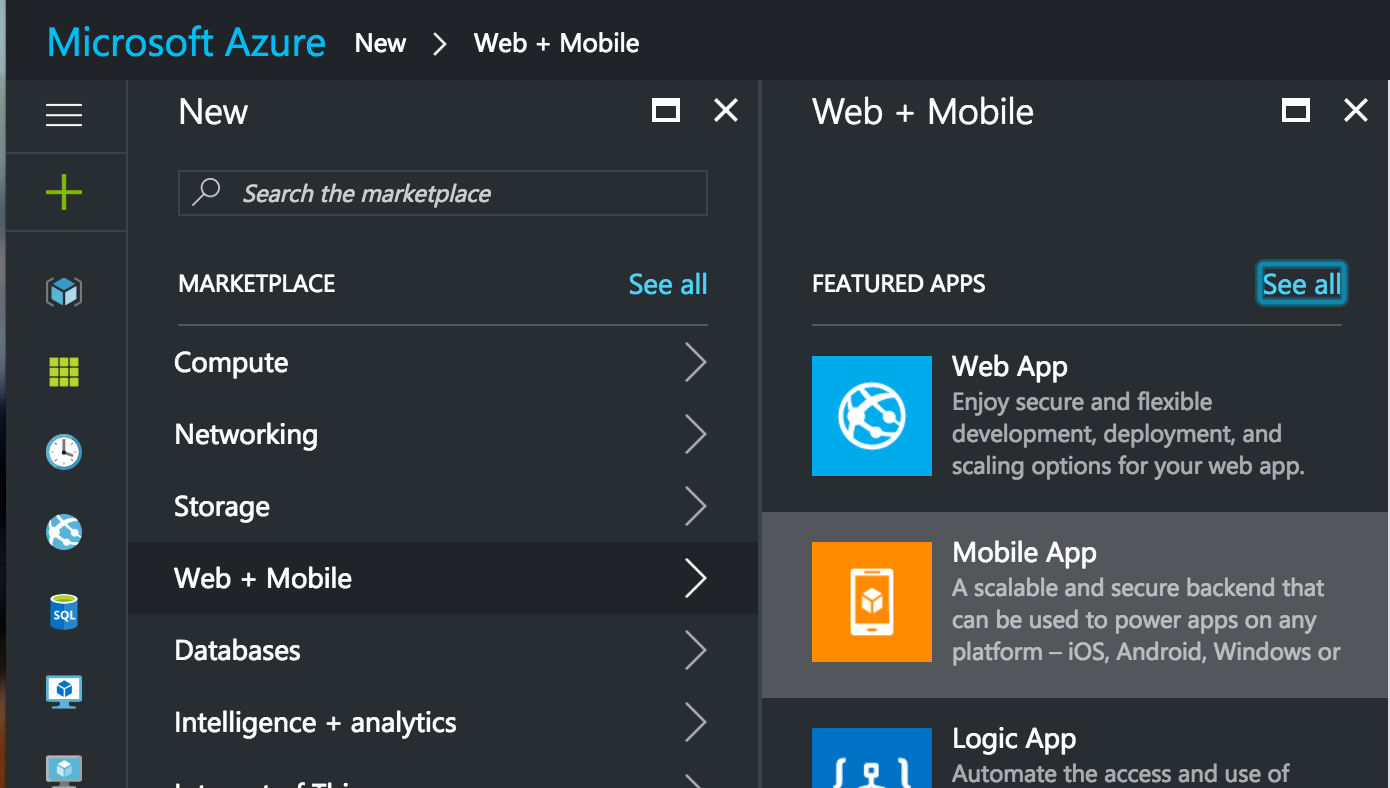
**Step 6:** Create a user in your active directory. You will have to do this in the old portal at:

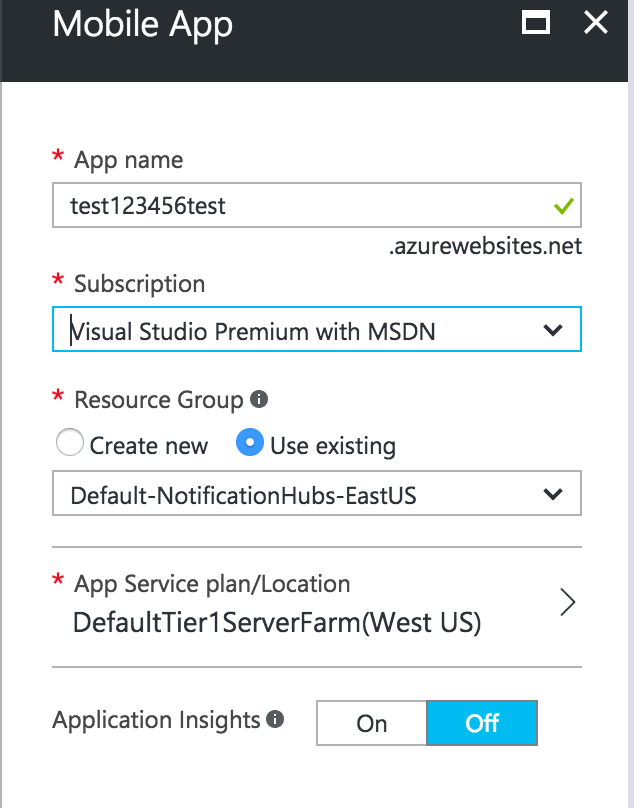
<https://manage.windowsazure.com/>

Select Active Directory and your specific directory (may be default) and use the Add User button.

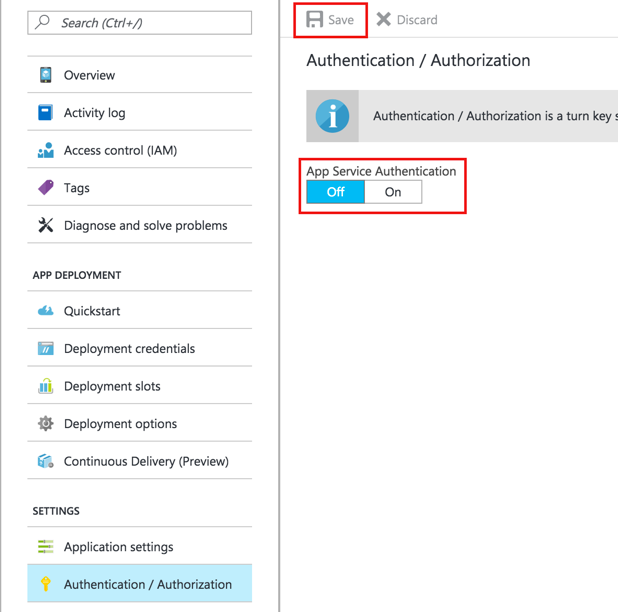


**Step 7:** Got back into the new Azure portal and create a new Mobile App. Click on the + sign in the upper left (For ease of use you may select the option to include it on your dashboard):

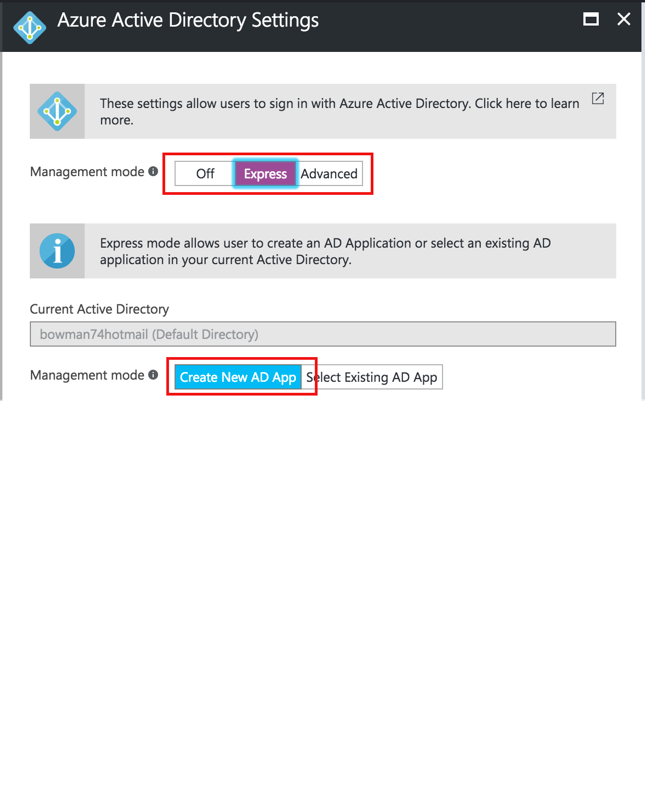




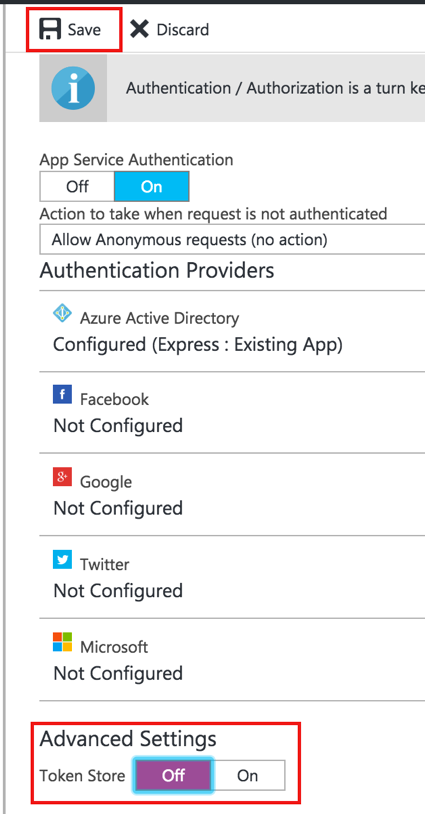
**Step 8:** Attach the mobile app you just created to Azure AD. Open the mobile app you just created and click on Authentication / Authorization. Inside the blade that shows up set App Server Authentication to the “On” position the press the save button at the top of the blade.



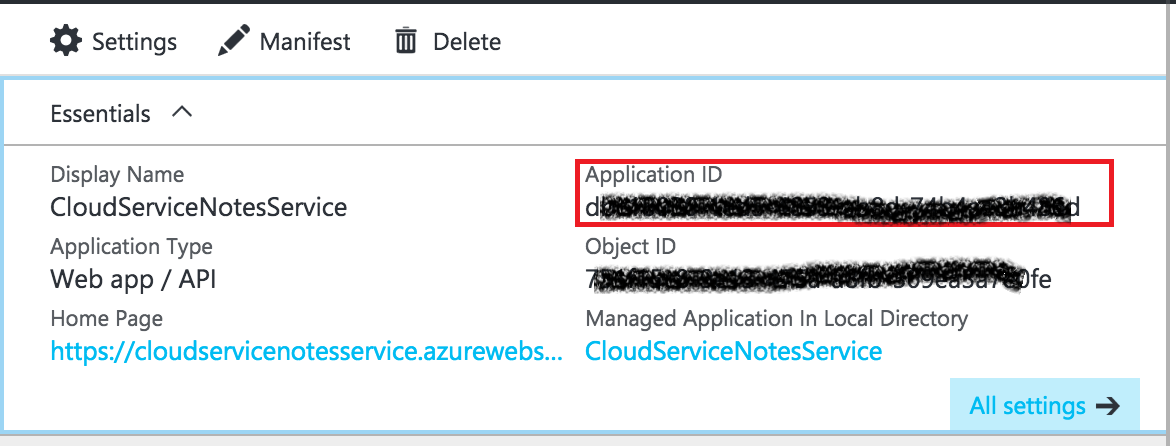
A new set of options will appear. Click on Azure Active Directory. Select the Management mode of Express and leave Create New AD App selected. Press the OK button at the bottom of the blade.



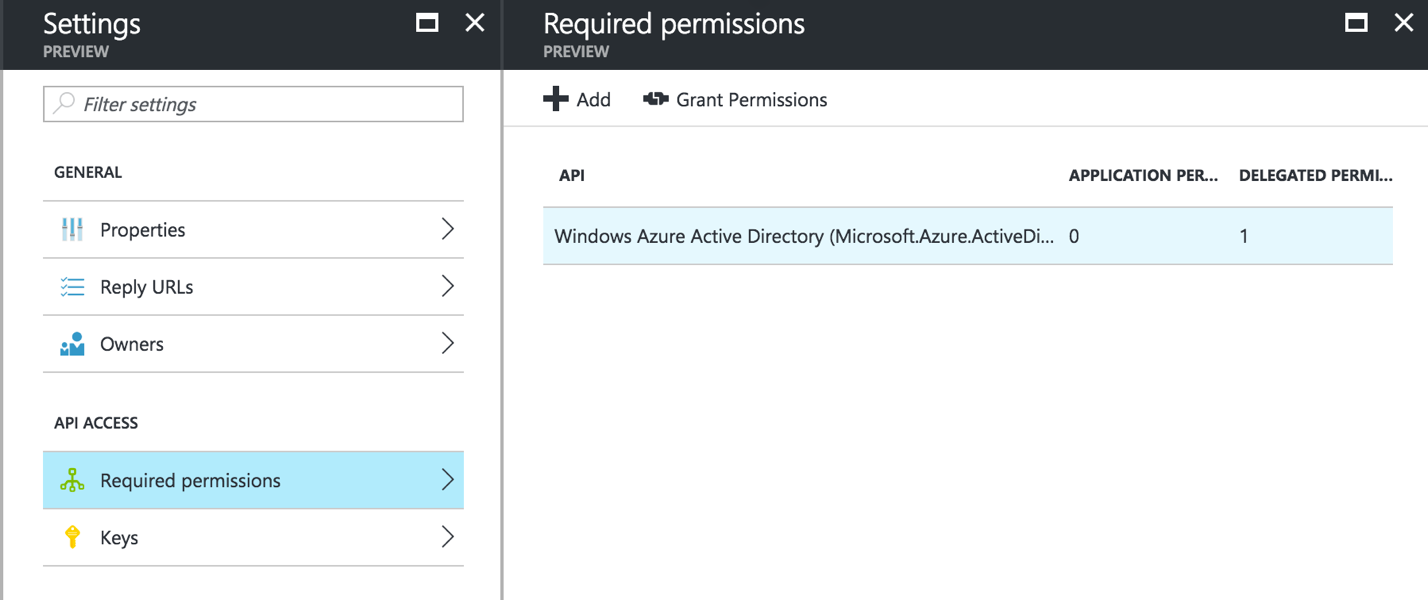
When the add blade closes, turn on token store at the bottom and press the Save button at the top of the screen.



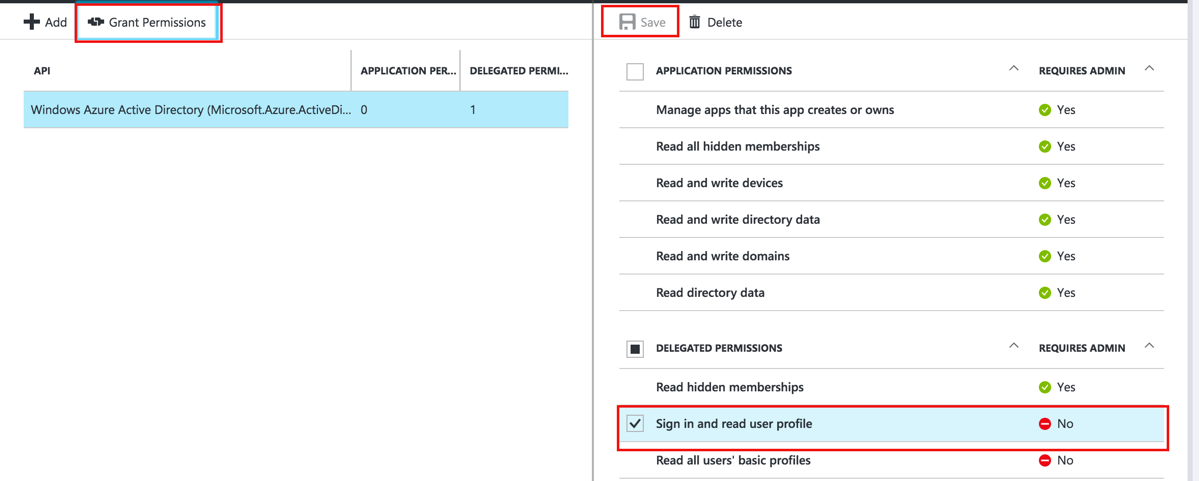
**Step 9:** Ensure that your app has all the necessary permissions. Go into the Active Directory blade and select App Registrations. Select your application from the resulting list. Take the Application Id and write it down or place it in notepad, we are going to need it in a little bit. We will refer to it as the ClientId.



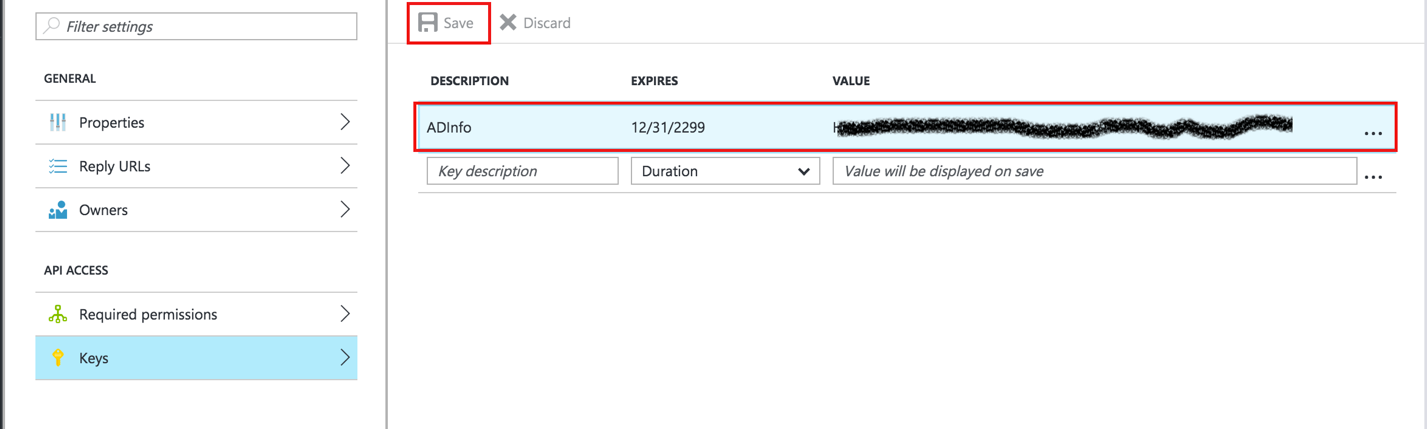
Click on Required Permissions and select Windows Azure Active Directory.



Make sure the sign in and read user profile option is selected and press save is necessary. Then press the Grant Permissions button to grant any app users permissions. This should only be your user for now for now.



Select the keys option and add a new key called ADInfo. You can set it never to expire and when you save it, write down or copy to notepad the key that is displayed. We will refer to this as the client secret.



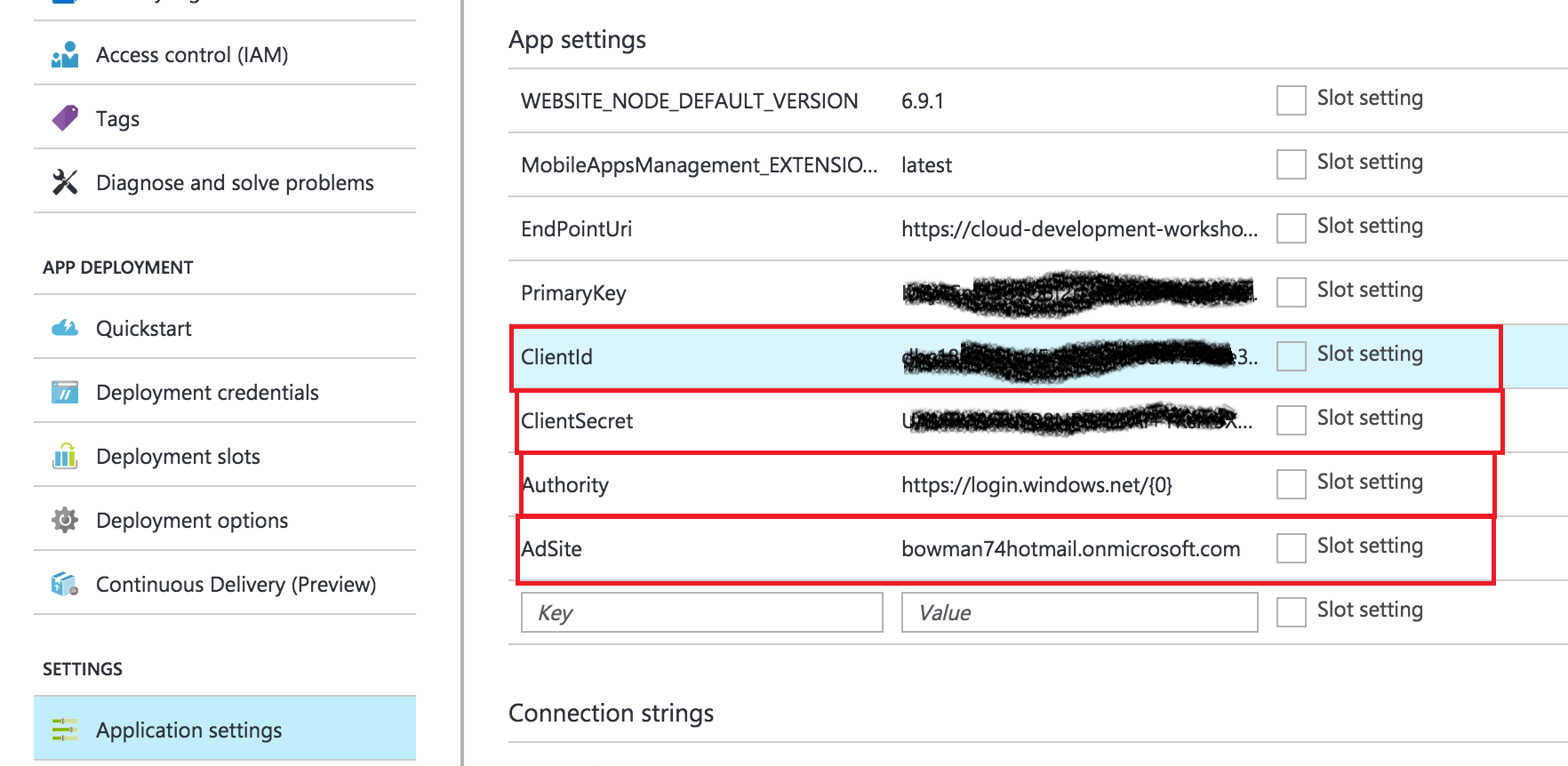
**Step 10:** Go to your Mobile App and go to Application Settings. We need to enter in four new keys.

ClientId – The client Id we saved from the Azure AD app.

ClientSecret – The secret we saved from the AzureAD app.

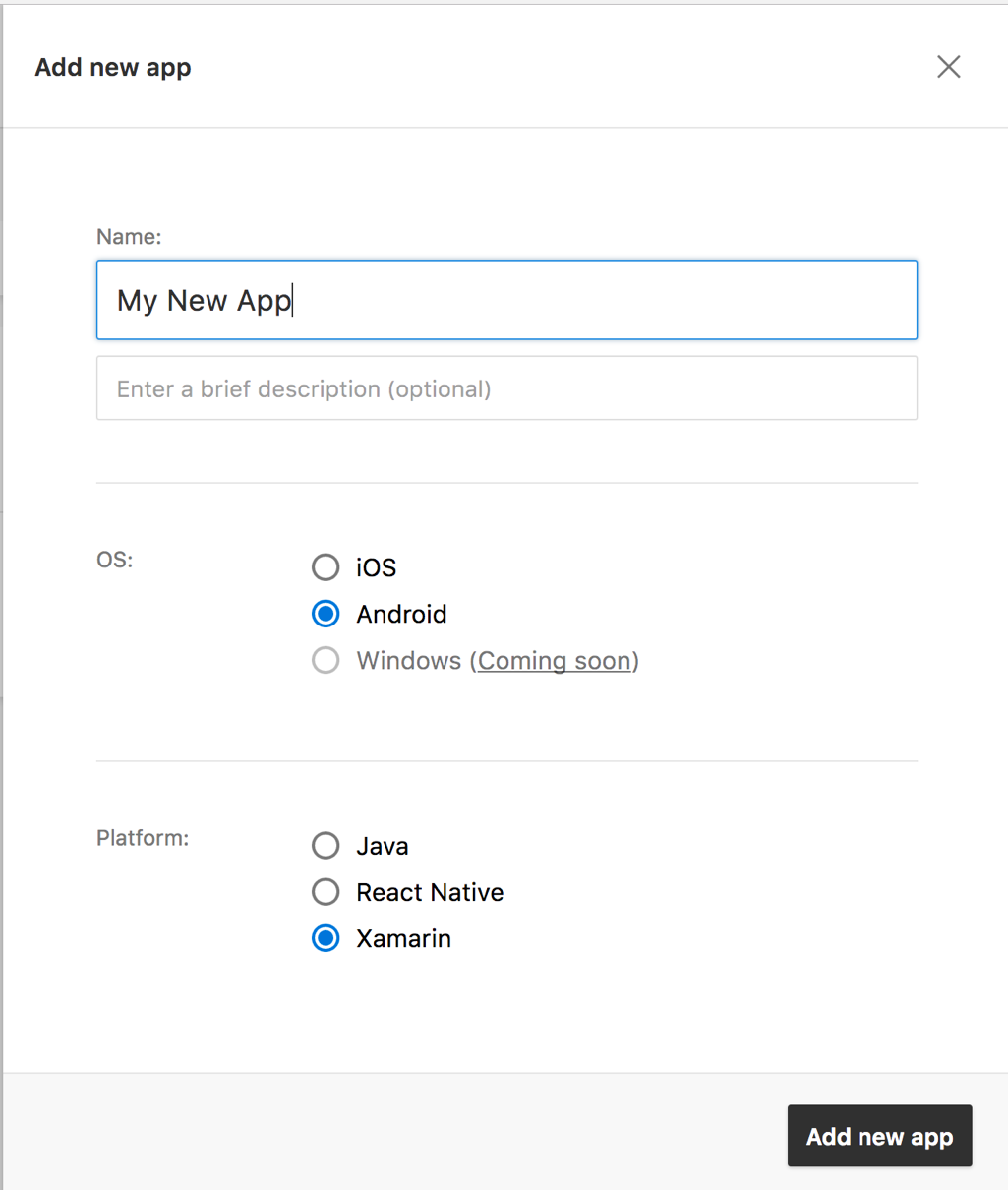
Authority – [https://login.windows.net/{0}](https://login.windows.net/%7b0%7d)

AdSite – Your AD site name from the upper right of the portal plus .onmicrosoft.com

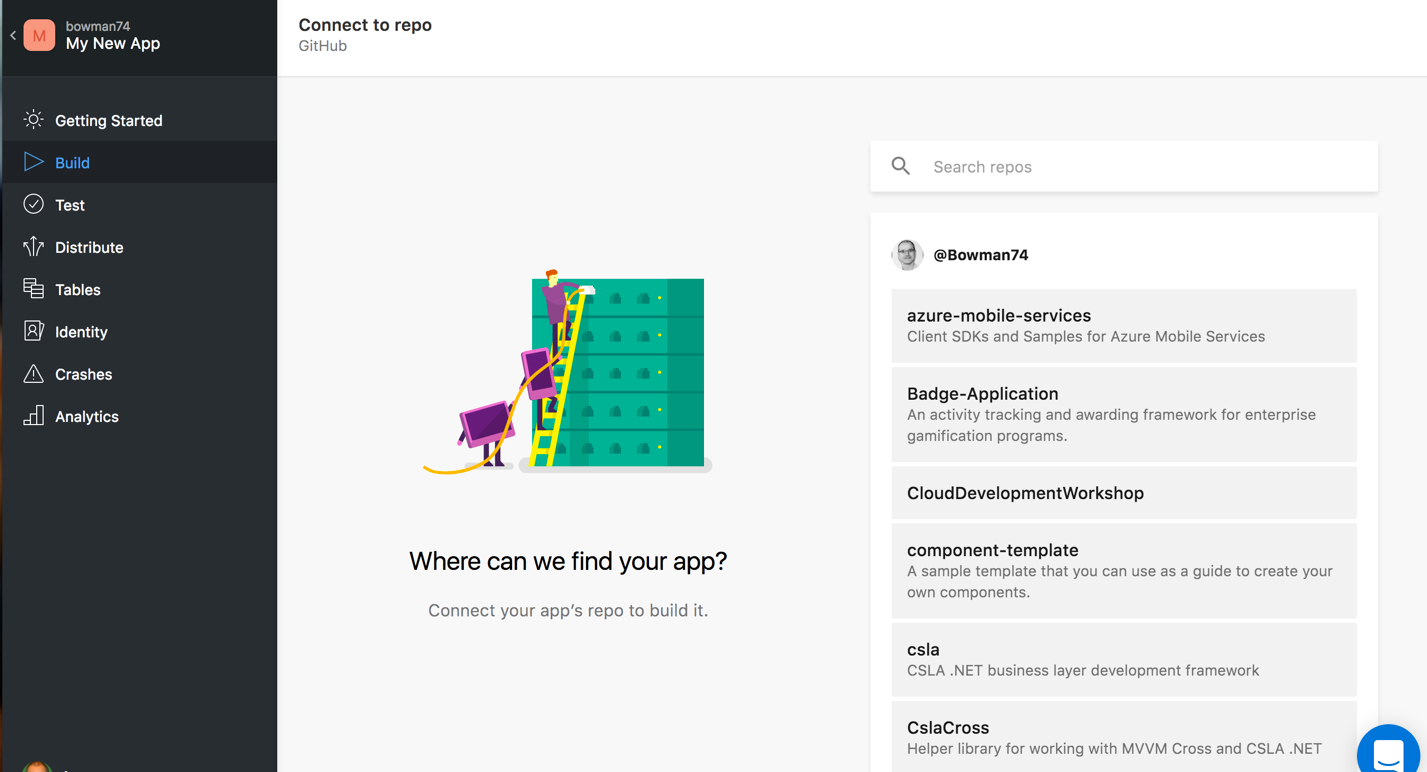


Step 11: Go into the Visual Studio Mobile Center and add click on Add New App. Give it a name and select the OS as Android and the platform as Xamarin.

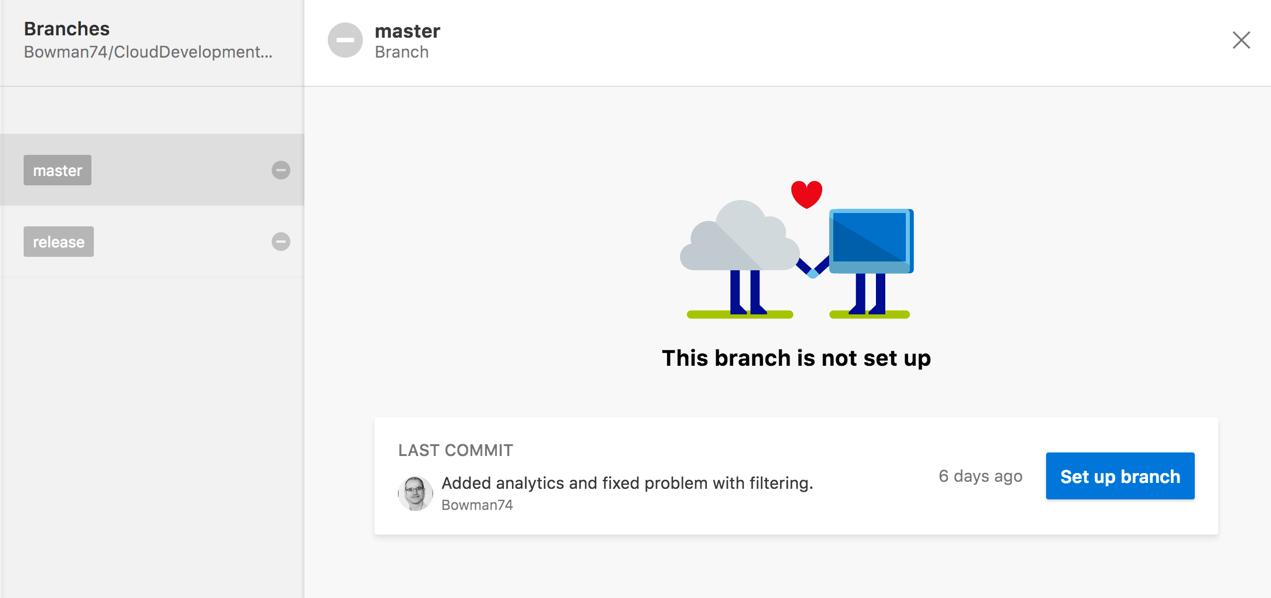
Click Add New App at the bottom when finished.



**Step 12:** Click on build and connect to your GitHub account and select the repository you forked.

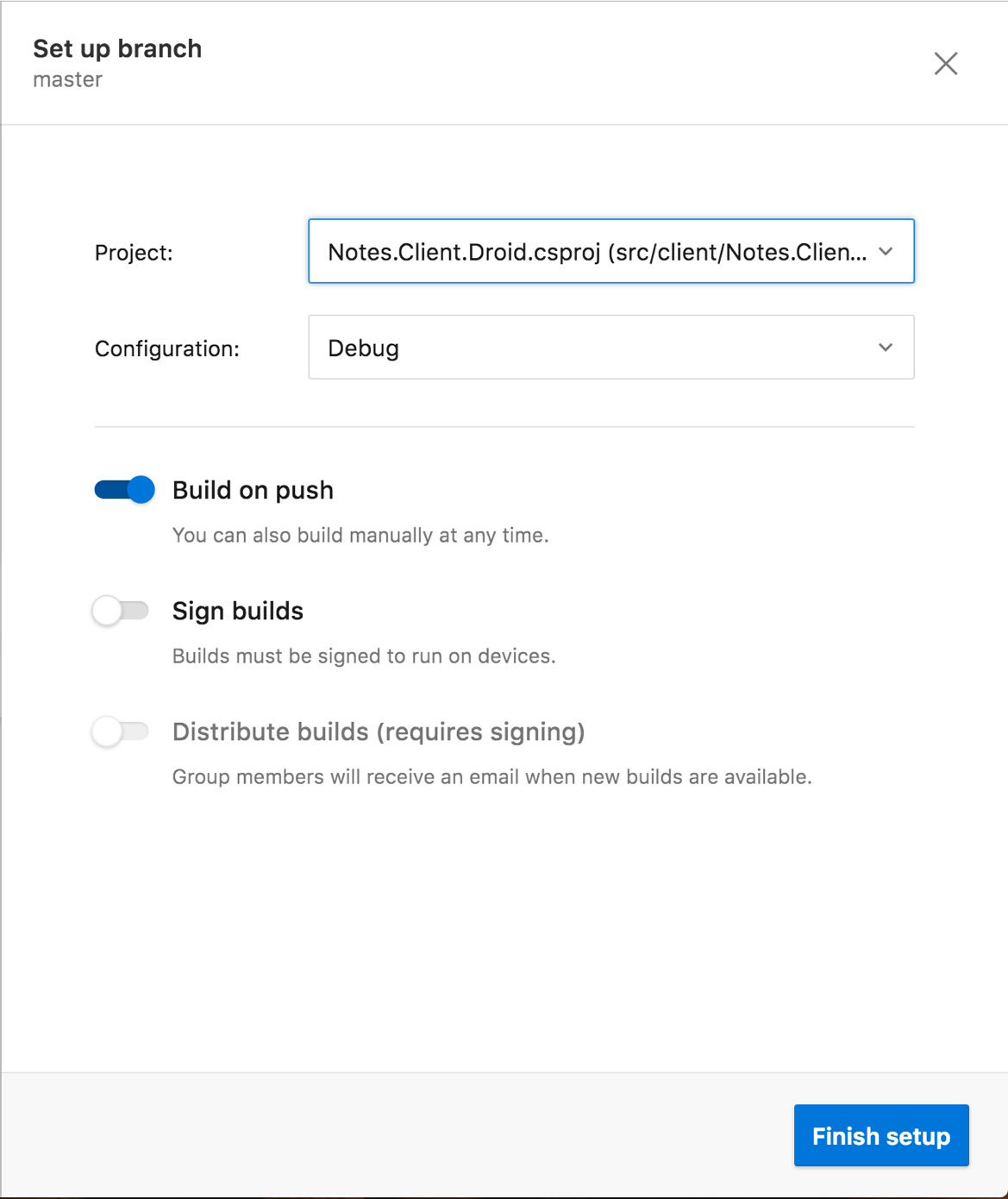


Select the Master Branch and Set up Branch.



Select the Notes.Client.Droid project from the Start directory and leave the configuration as debug and Build on Push selected. Press Finish Setup and a build should be kicked off. This will now cause a build to occur anytime the master branch is updated.

**Step 13:** Create another app for iOS and follow the steps above for Android except choose iOS and the OS.



# Hands on Lab 2: Making a quick Xamarin Forms app to login to Azure Back end

## Steps:

**Step 1:** Open the Notes.Client solution and add the following Nuget packages to the Notes.Client, Notes.Client.Droid, Notes.Client.Ios and Notes.Client.Uwp projects:

Microsoft.Azure.Mobile – Mobile Center Communication

Microsoft.Azure.Mobile.Analytics – Collection of custom event information

Microsoft.Azure.Mobile.Client – App Service Login, API communication

Microsoft.Azure.Mobile.Crashes – Collection of app crash information

**Step 2:** Add a new interface called IAzureService to the Notes.Client project (I usually put such files in a folder labeled interfaces).

The interface should have two methods. One for logging into the service and another to return a reference to the service.

Task<MobileServiceUser> LoginAsync();

IMobileServiceClient GetMobileServicesClinet();

These interfaces will be used with dependency injection to get client specific versions of the Azure service.

**Step 3:** In the same project edit the HelperConstants class to set the ClientAddress to be the address of your mobile service.

**Step 4:** Add a new class to the Notes.Client.Droid project to implement the Azure Service. Make the class implement the IAzureService interface we just created. I usually put this in a Services folder.

Add the following code above the namespace to register this class with the Xamrin Forms dependency service:

[assembly: Xamarin.Forms.Dependency(typeof(AzureService))]

Add a new static public property to save the current Android activity, the type should be FormsAppCompatActivity. Call this property Context.

Add a new private static attribute to the class called \_client that is of type IMobileServiceClient.

In the GetMobileServiceClient method check if the \_client is null and if so create a new instance of the MobileServiceClient and store it in the \_client variable. Return that value.

if (\_client == null)

{

\_client = new MobileServiceClient(Helpers.Constants.ClientAddress);

}

return \_client;

Add code to the LoginAsync method to call GetMobileServiceClient and call its LoginAsync method using the Context property with the login provider being AzureAD .

var client = GetMobileServicesClinet();

return await client.LoginAsync(Context, MobileServiceAuthenticationProvider.WindowsAzureActiveDirectory);

**Step 5:** In the Notes.Client.Droid project edit the MainActivity’s OnCreate method to set the Context property of the AzureService we just created to the activity. I do this right after calling the base class’s OnCreate method.

AzureService.Context = this;

**Step 6:** Add a new class to the Notes.Client.Ios project to implement the Azure Service. Make the class implement the IAzureService interface we just created. I usually put this in a Services folder.

Add the following code above the namespace to register this class with the Xamrin Forms dependency service:

[assembly: Xamarin.Forms.Dependency(typeof(AzureService))]

Add a new private static attribute to the class called \_client that is of type IMobileServiceClient.

In the GetMobileServiceClient method check if the \_client is null and if so create a new instance of the MobileServiceClient and store it in the \_client variable. Return that value.

if (\_client == null)

{

\_client = new MobileServiceClient(Helpers.Constants.ClientAddress);

}

return \_client;

Add code to the LoginAsync method to call GetMobileServiceClient and call its LoginAsync method using the Context property with the login provider being AzureAD .

var client = GetMobileServicesClinet();

return await client.LoginAsync(UIApplication.SharedApplication.KeyWindow.RootViewController, MobileServiceAuthenticationProvider.WindowsAzureActiveDirectory);

**Step 7:** Add a new class to the Notes.Client.Uwp project to implement the Azure Service. Make the class implement the IAzureService interface we just created. I usually put this in a Services folder.

Add the following code above the namespace to register this class with the Xamrin Forms dependency service:

[assembly: Xamarin.Forms.Dependency(typeof(AzureService))]

Add a new private static attribute to the class called \_client that is of type IMobileServiceClient.

In the GetMobileServiceClient method check if the \_client is null and if so create a new instance of the MobileServiceClient and store it in the \_client variable. Return that value.

if (\_client == null)

{

\_client = new MobileServiceClient(Helpers.Constants.ClientAddress);

}

return \_client;

Add code to the LoginAsync method to call GetMobileServiceClient and call its LoginAsync method using the Context property with the login provider being AzureAD .

var client = GetMobileServicesClinet();

return await client.LoginAsync(MobileServiceAuthenticationProvider.WindowsAzureActiveDirectory);

**Step 8:** In the Notes.Client project’s MainPage.axaml.cs add a new Boolean to keep track of if the user is already logging in and to stop recursion:

private bool LoggingIn = false;

**Step 9:** In the Notes.Client project’s MainPage.axaml.cs add a new LoginAsync method that returns a Task. This method will keep showing the login screen until the user successfully logs in.

When the user does successfully login we save the user’s token and id so we can try and use them later if the app is restarted and the token is still valid.

private async Task LoginAsync(AppSettingsService appSettingsService, IAzureService azureService)

{

LoggingIn = true;

do

{

try

{

var user = await azureService.LoginAsync();

appSettingsService.Add(AuthenticaitonTokenKey, user.MobileServiceAuthenticationToken);

appSettingsService.Add(UserIdKey, user.UserId);

await GetCurrentUserInfoAsync(azureService);

}

catch (Exception) { }

} while (UserInformation.User == null);

}

**Step 10:** Add an AuthenticateAsync method that will handle when the user first enters the system. It will load the user’s id and token if save in app data and attempt to make a call. If the call fails or if there are no tokens saved, the login process will be initiated.

private async Task AuthenticateAsync()

{

var appSettingsService = new AppSettingsService();

var azureService = DependencyService.Get<IAzureService>();

if (azureService.GetMobileServicesClinet().CurrentUser == null)

{

string userId;

string token;

if (appSettingsService.TryGetValue(AuthenticaitonTokenKey, out token) &&

appSettingsService.TryGetValue(UserIdKey, out userId))

{

var user = new MobileServiceUser(userId);

user.MobileServiceAuthenticationToken = token;

azureService.GetMobileServicesClinet().CurrentUser = user;

if (await GetCurrentUserInfoAsync(azureService))

{

return;

}

}

await LoginAsync(appSettingsService, azureService);

}

}

**Step 11:** Add a method to get the current user information from AD like the user name. This will be calling a method on our API that was provided in the start project. It returns a Boolean indicating if the call was successful.

private async Task<bool> GetCurrentUserInfoAsync(IAzureService azureService)

{

bool returnValue;

try

{

UserInformation.User = await azureService.GetMobileServicesClinet().InvokeApiAsync<User>("Profile", HttpMethod.Get, null);

Title = $"Tasks for: {UserInformation.User.FullName}";

returnValue = true;

}

catch (MobileServiceInvalidOperationException)

{

returnValue = false;

}

return returnValue;

}

**Step 12:** Add a logout method that will remove any user information in memory or stored in app local storage. When the user is logged out of Azure, the form’s login method is again called as a user is required for this app to work.

private async Task LogoutAsync()

{

UserInformation.User = null;

var appSettingsService = new AppSettingsService();

var azureService = DependencyService.Get<IAzureService>();

appSettingsService.Remove(AuthenticaitonTokenKey);

appSettingsService.Remove(UserIdKey);

await azureService.GetMobileServicesClinet().LogoutAsync();

await LoginAsync(appSettingsService, azureService);

}

**Step 13:** Modify the form’s constructor when adding a toolbar item in the Device.Onplatform call for the default case. The action should call the LogoutAsync method.

ToolbarItems.Add(new ToolbarItem("Logout", "", async () => { await LogoutAsync(); }, ToolbarItemOrder.Secondary, 2));

**Step 14:** Modify the OnAppearing method to call AuthenticateAsync before GetNotesForUserAsync. This will make sure that when the form is displayed a user is attached to the Azure client and if not, they will be forced to login.

await AuthenticateAsync();

Step 15: Modify the ActionsClick method to make a call to LogoutAsync when the logout case happens:

await LogoutAsync();

Step 16: In the OnStart method of the Notes.Client App file we want to start crash reporting and analytics. Add these two lines to make that happen:

MobileCenter.Start(typeof(Analytics), typeof(Crashes));

Analytics.Enabled = true;

**Step 17:** In the Notes.Client.Droid project’s MainActivity we want to initialize error logging and analytics and tie it to our Android App in Visual Studio Mobile Center. In the OnCreate method place the configuration line after the call to Forms.Init.

MobileCenter.Configure("Replace with your app secret from the Manage App section");

**Step 18:** In the Notes.Client.ios project’s AppDelegate we want to initialize error logging and analytics and tie it to our Android App in Visual Studio Mobile Center. In the FinishedLaunching method place the configuration line after the call to Forms.Init.

MobileCenter.Configure("Replace with your app secret from the Manage App section");

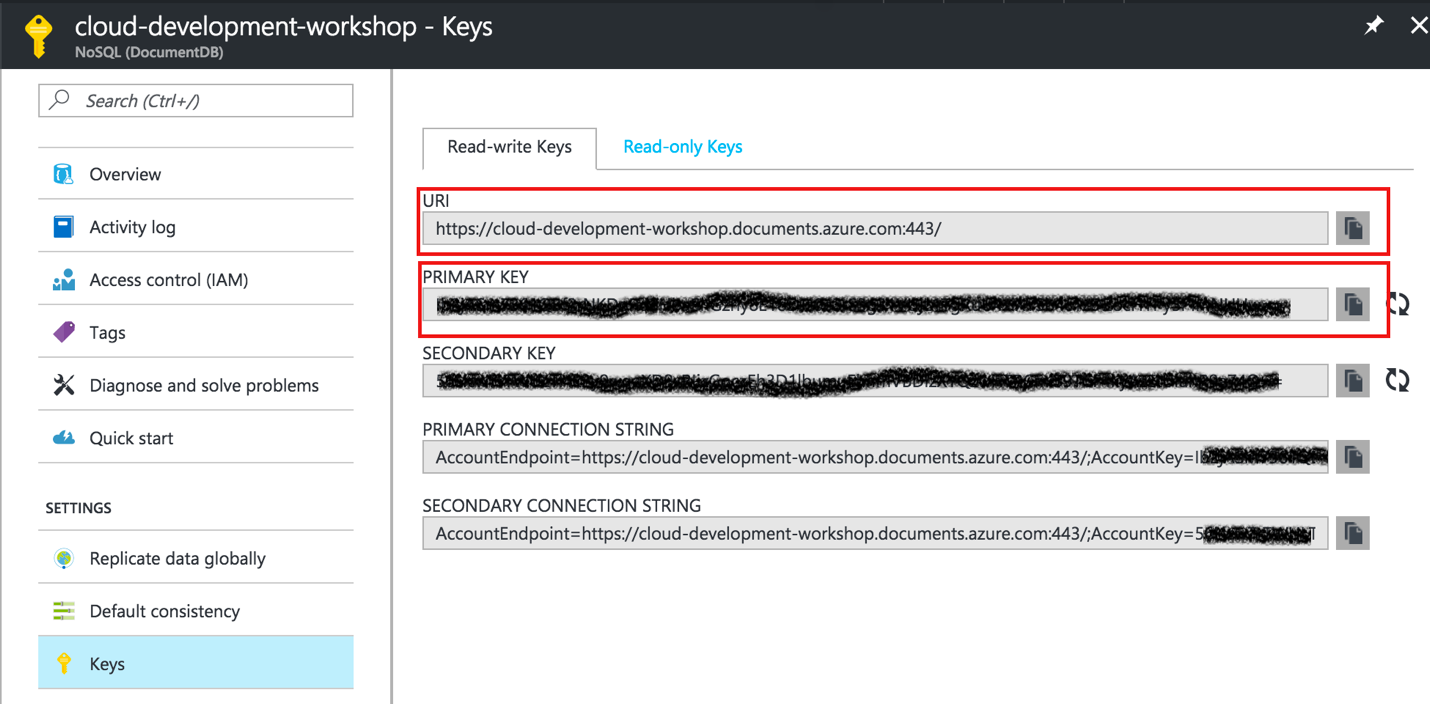
**Step 19:** Run the app and make sure it works

# Hands on Lab 3: Set up an Azure DB and Azure App API and display records in mobile client.

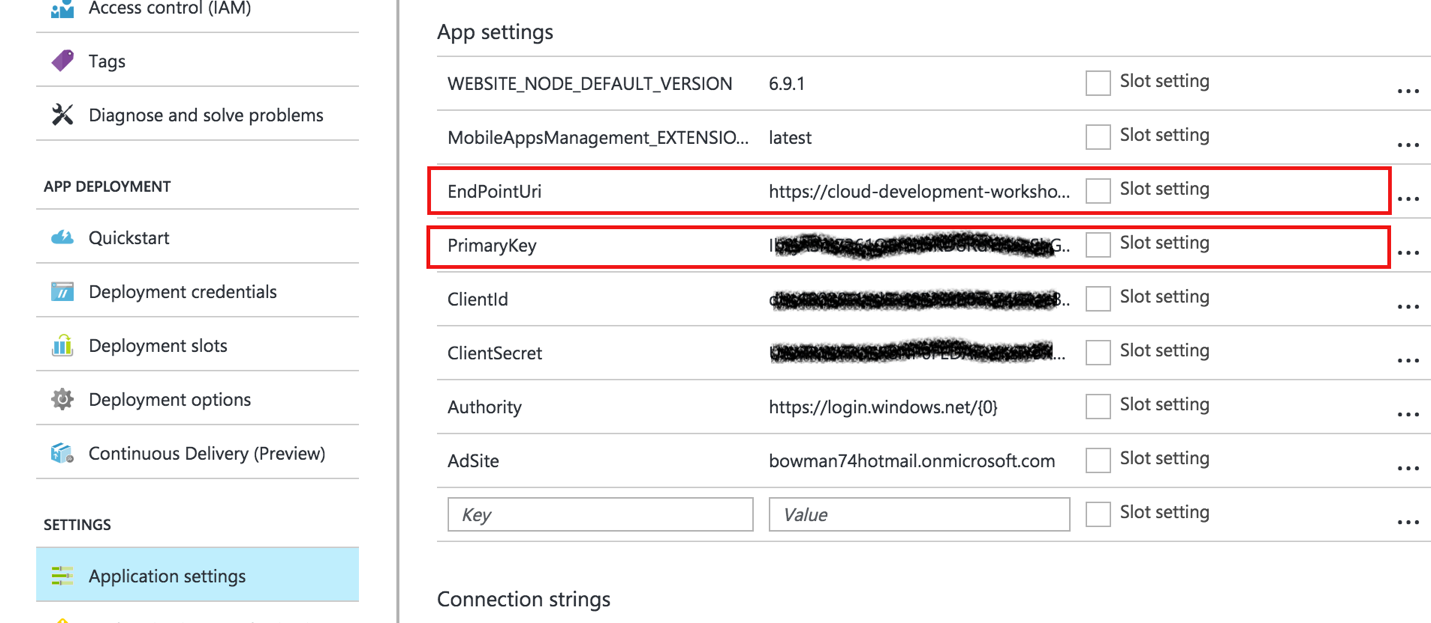
## Steps:

**Step 1:** Add a new document database in the Azure portal by selecting ‘+’\Databases\NoSQL (DocumentDB).

Make sure to save the URI and primary key from the keys section.



In your Azure Mobile App service add two new keys under Application Settings with these values.



**Step 2:** Create a new class in the service project (NotesService project in the NotesService solution) to open the document database connection and create the database and collection if they don’t currently exist.

**Step 3:** In the new class add class level variables for the following:

* String URI to the document database
* String key to the document database
* DocumentClient instance holder
* String database name
* String collection name

Pull the URI and primary Key from settings we added to the Mobile App Service using ConfigurationSettings.AppManager calls.

Make the database and collection name variables public as users of the service will need to access them.

**Step 4:** Add an asynchronous method to check if the document collection exists using an already instantiated DocumentClient (either passed to the method as a parameter or using the class level variable)

The method on the document client to check if a collection exists is ReadDocumentCollectionAsync. If a call to this method returns a DocumentClientException then it does not exist.

The method to create a new document collection id the document client’s CreateDocumentCollectionAsync. Set the OfferThroughput to 400.

Make sure the id is set to the collection name from your class level variable.

**Step 5:** Add an asynchronous method to check if the document database already exists using an already instantiated DocumentClient (either passed to the method as a parameter or using the class level variable).

The method on the document to see if the database exists is ReadDatabaseAsync taking in the database name. If it is not found it will throw a DocumentClientException with a status code of HttpStatusCode.NotFound.

If the database is not found call the CreateDatabaseAsyncMethod.

**Step 6:** Create a public asynchronous method to return a connection to the document database. If the class level client variable is null it should create a new instance of the DocumentClient class using the URI and primary key variables.

When the connection is created the methods to check if the database exists and then the method to check if the document collection exists.

**Step 7:** Open the NotesController class and implement the Get method. It should use the service class you created to get a connection to the document database. With the connection create a document query to return all the notes as so:

IDocumentQuery<Note> noteQuery = client.CreateDocumentQuery<Note>(

UriFactory.CreateDocumentCollectionUri(NoteDocumentService.DocumentDatabaseName, NoteDocumentService.DocumentCollectionName), queryOptions)

.AsDocumentQuery();

Then iterate through the document query to create the array of notes to return. Do this by creating a collection object and add the notes to it. When you are done you can use the ToArray() method on the collection to return an array of notes. This code can be used to iterate through the notes query results:

while (noteQuery.HasMoreResults)

{

foreach (var note in await noteQuery.ExecuteNextAsync<Note>().ConfigureAwait(false))

{

returnValue.Add(note);

}

}

**Step 8:** Implement the Post method. This should get a connection to the DocumentDB using the service we created and the use the CreateDocumentAsync method to save the Note object passed in as a parameter as a new note.

This method should return the note object just added.

**Step 9**: Implement the Put method to update an existing note. This method’s implementation should be the same as the Post method but call ReplaceDocumentAsync instead of CreateDocumentAsync.

**Step 10:** Implement the method to delete a note. Similar to the prior two uses the DeleteDocumentAsync method and the document id passed in as a parameter. It should return HttpStatuscode.OK.

return Request.CreateResponse(HttpStatusCode.OK);

**Step 11:** In the NotesQueryController implement the Post method. It should be similar to the Get method of the NotesController in Step 7. However, since we are using a body to specify information we are sending to the server, we are using Post in this case instead of Get.

The query will be the same except to include a Where statement to see if the content of the note contains the QueryString.

.Where(t => t.Content.Contains(query.QueryString))

Like the method in Step 7, it should also return an array of matching notes.

**Step 12:** Compile and deploy the Service project.

**Step 13:** In the Notes.Client solution, Notes.Client project modify the MainPAge.xaml.cs file to implement the GetNotesforUserAsync method. Remove the current implementation and instead check the value of the \_Filter attribute.

If the value is not equal to String.Empty call the NotesQuery API, passing a new QueryString object set with the \_Filter object value as a POST call.

notes = await azureService.GetMobileServicesClinet()

.InvokeApiAsync<Query, Note[]>("NotesQuery", new Query {QueryString = \_Filter}, HttpMethod.Post, null);

Otherwise call the Notes API to return all the notes with a GET call.

notes = await azureService.GetMobileServicesClinet().InvokeApiAsync<Note[]>("Notes", HttpMethod.Get, null);

Make sure the Notes property is set with the return value of one of those two calls using the .ToList() method.

You may also want to catch and display the login for any AnauthorizedAccessExceptions.

**Step 14:** In the EditNote.xaml.cs file implement the SaveNote\_Clicked event handler. If the IsNew property is true send the Note to the POST method of the “Notes” API. Also call into the analytics service adding an event that a not was added.

Note = await azureService.GetMobileServicesClinet()

.InvokeApiAsync<Note, Note>("Notes", Note, HttpMethod.Post, null);

Analytics.TrackEvent("Note Added", new Dictionary<string, string> { { "UserId", UserInformation.User.UserId }, { "NoteId", Note.Id } });

Otherwise call the PUT method and write to analytics that a note was edited.

Note = await azureService.GetMobileServicesClinet()

.InvokeApiAsync<Note, Note>($"Notes/{Note.Id}", Note, HttpMethod.Put, null);

Analytics.TrackEvent("Note Edited", new Dictionary<string, string> { { "UserId",

UserInformation.User.UserId }, { "NoteId", Note.Id } });

If the operation is successful leave the code to pop the current screen off the stack.

**Step 15:** In the same class edit the DeleteNoteAsync method to handle if the user states they want to delete the note. Inside the if block add code to call the Notes API, including the Note’s Id as part of the URL. Use the DELETE method for this. Also write the fact that the note was deleted to the application’s analytics.

await azureService.GetMobileServicesClinet().InvokeApiAsync<HttpResponseMessage>($"Notes/{Note.Id}", HttpMethod.Delete, null);

Analytics.TrackEvent("Note Deleted", new Dictionary<string, string> { { "UserId", UserInformation.User.UserId }, { "NoteId", Note.Id } });

Leave the class to pop the form off the stack if the operation is successful.

**Step 16:** Run the app and make sure it works