LabW07 – User Management

Objectives:

1. Understand how to use Azure Cloud service for user management
2. Understand login using social account

Tasks:

1. Use Azure Mobile App services to support Facebook login
2. Add authentication to your Mobile Services Android app (JavaScript backend)
3. Cache authentication tokens on the client
4. Service-side authorization of users in Mobile Services

Task 1: Use Azure Mobile App services to support Facebook login

After storing data in the cloud, every user of the app will see the same content. Therefore, we must also support user login and impose data access permission to each user so different users only see and modify their own data. As login with social account is very common nowadays, in this tutorial we take login with Facebook account as an example. Login with other type of user account such as Google and Twitter are similar with this tutorial.

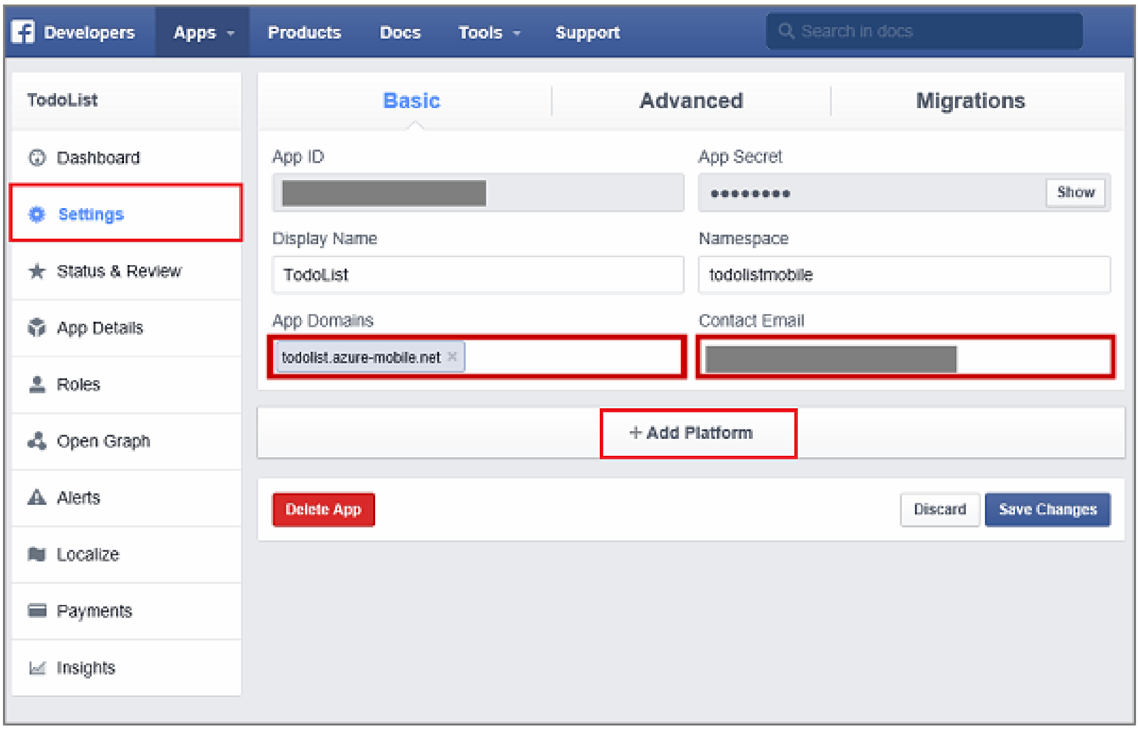
1. Register an app for Facebook authentication using Azure Mobile App service in https://developers.facebook.com/

This tutorial is based on: https://azure.microsoft.com/en-us/documentation/articles/mobile-services-how-to-register-facebook-authentication/

To complete the procedure in this topic, you must have a Facebook account that has a verified email address and a mobile phone number. To create a new Facebook account, go to facebook.com.

- Navigate to the Facebook Developers website (https://developers.facebook.com/) and sign-in with your Facebook account credentials.
- (Optional) If you have not already registered, click **My Apps** then click **Register as a Developer**, accept the policy and follow the registration steps.
- Click **My Apps > Add a New App > Website > Skip** and **Create App ID**.
- In **Display Name**, enter a unique name for your app, choose a **Category** for you app, then click **Create App ID** and complete the security check. This takes you to the developer dashboard for your new Facebook app.
- On the **App Secret** field, click **Show**, provide your password if requested, then make a note of the values of **App ID** and **App Secret**. You will uses these later to configure your application in Azure. Security Note The app secret is an important security credential. Do not share this secret with anyone or distribute it within a client application.

- In the left navigation bar, click **Settings**, type the domain of your mobile service in **App Domains**, enter an optional **Contact Email**, click **Add Platform** and select **Website**.



App ID [redacted]

App Secret ••••••••

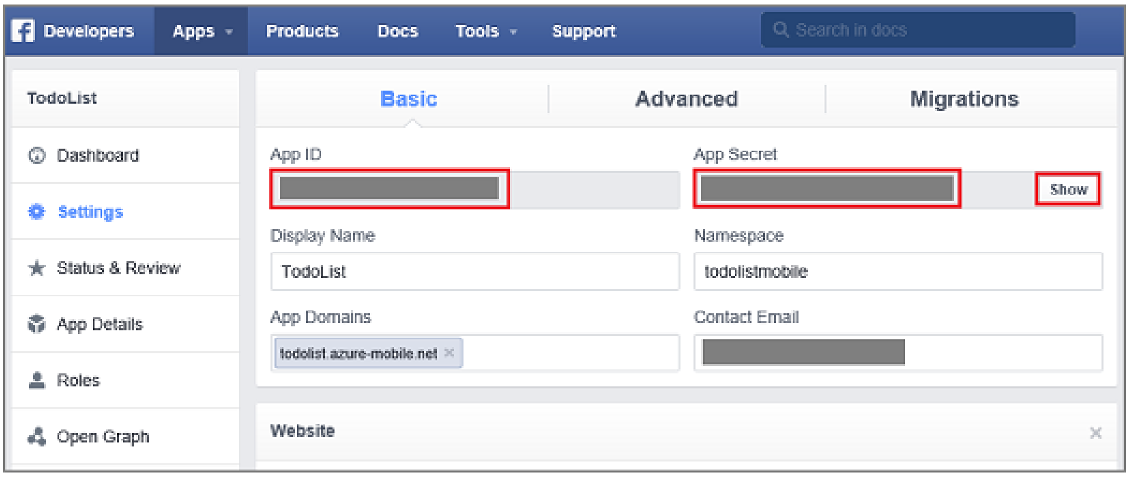
Display Name TodoList

Namespace todolistmobile

App Domains todolist.azure-mobile.net

Contact Email [redacted]

- Type the URL of your mobile service in **Site URL**, then click **Save Changes**.
- Click **Show**, provide your password if requested, then make a note of the values of **App ID** and **App Secret.**



App ID [redacted]

App Secret [redacted]

Display Name TodoList

Namespace todolistmobile

App Domains todolist.azure-mobile.net

Contact Email [redacted]

- Click the Advanced tab, type one of the following URL formats in **Valid OAuth redirect URIs**, then click **Save Changes**:
  .NET backend: https://<mobile_service>.azure-mobile.net/signin-facebook
  JavaScript backend: https://<mobile_service>.azure-mobile.net/login/facebook
  Make sure that you use the correct redirect URL path format for your type of Mobile Services backend, using the HTTPS scheme. When this is incorrect, authentication will not succeed.

- The Facebook account which was used to register the application is an administrator of the app. At this point, only administrators can sign into this application. To authenticate other Facebook accounts, click **App Review** and enable **Make todolist-complete-nodejs public** to enable general public access using Facebook authentication.

You are now ready to use a Facebook login for authentication in your app by providing the App ID and App Secret values to Mobile Services.

Task 2: Add authentication to your Mobile Services Android app (JavaScript backend)

This topic shows you how to authenticate users in Azure Mobile Services from your app. In this tutorial, you add authentication to the quickstart project using an identity provider that is supported by Mobile Services. After being successfully authenticated and authorized by Mobile Services, the user ID value is displayed. This tutorial is based on: https://azure.microsoft.com/enus/documentation/articles/mobile-services-android-get-started-users/

Before you start this tutorial, you must first complete Add Mobile Services to an existing app: https://azure.microsoft.com/zh-cn/documentation/articles/mobile-services-android-get-started-data/ .

If you completed the quickstart tutorial prior to the release of Azure Mobile Services Android SDK 2.0, you must re-do it, because the SDK is not backwards compatible. To verify the version, check the dependencies section of your project's build.gradle file.

1. In Android Studio, open the project that you created when you completed the above step.

   From the Run menu, then click Run app; verify that an unhandled exception with a status code of 401 (Unauthorized) is raised after the app starts.

   This happens because the app attempts to access Mobile Services as an unauthenticated user, but the TodoItem table now requires authentication.

2. Next, we will update the app to authenticate users before requesting resources from the mobile service.

In Project Explorer in Android Studio, open the ToDoActivity.java file and add the following import statements.

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.atomic.AtomicBoolean;

import android.content.Context;
import android.content.SharedPreferences;
import android.content.SharedPreferences.Editor;

import com.microsoft.windowsazure.mobileservices.authentication.MobileServiceAuthenticationProvider;
import com.microsoft.windowsazure.mobileservices.authentication.MobileServiceUser;
```

3. Add the following method to the ToDoActivity class:

```
private void authenticate() {
   // Login using the Google provider.

   ListenableFuture<MobileServiceUser> mLogin =
mClient.login(MobileServiceAuthenticationProvider.Facebook);

   Futures.addCallback(mLogin, new FutureCallback<MobileServiceUser>() {
      @Override
      public void onFailure(Throwable exc) {
         createAndShowDialog((Exception) exc, "Error");
      }
      @Override
      public void onSuccess(MobileServiceUser user) {
         createAndShowDialog(String.format(
               "You are now logged in - %1$2s",
               user.getUserId()), "Success");
         createTable();
      }
   });
}
```

This creates a new method to handle the authentication process. A dialog is displayed which displays the ID of the authenticated user. You cannot proceed without a positive authentication.

4. In the onCreate method, add the following line of code after the code that instantiates the MobileServiceClient object. This call starts the authentication process.

```
authenticate();
```

5. Move the remaining code after authenticate(); in the onCreate method to a new createTable method, which looks like this:

```
private void createTable() {

    // Get the table instance to use.
    mToDoTable = mClient.getTable(ToDoItem.class);

    mTextNewToDo = (EditText) findViewById(R.id.textNewToDo);

    // Create an adapter to bind the items with the view.
    mAdapter = new ToDoItemAdapter(this, R.layout.row_list_to_do);
    ListView listViewToDo = (ListView) findViewById(R.id.listViewToDo);
    listViewToDo.setAdapter(mAdapter);

    // Load the items from Azure.
    refreshItemsFromTable();
}
```

6. From the Run menu, then click Run app to start the app and sign in with your chosen identity provider. When you are successfully logged-in, the app should run without errors, and you should be able to query the backend service and make updates to data. The final code should look like week07_nocache.java in your lib file.

Task 3: Cache authentication tokens on the client

The previous example showed a standard sign-in, which requires the client to contact both the identity provider and the backend Azure service every time that the app starts. Not only is this method inefficient, you can run into usage-related issues should many customers try to start you app at the same time. A better approach is to cache the authorization token returned by the Azure service and try to use this first before using a provider-based sign-in.

You can cache the token issued by the backend Azure service regardless of whether you are using client-managed or service-managed authentication. This tutorial uses service-managed authentication.

1. Open the ToDoActivity.java file and add the following import statements:

```
import android.content.Context;
import android.content.SharedPreferences;
import android.content.SharedPreferences.Editor;
```

2. Add the following members to the ToDoActivity class.

```
public static final String SHAREDPREFFILE = "temp";
public static final String USERIDPREF = "uid";
public static final String TOKENPREF = "tkn";
```

3. In the ToDoActivity.java file, add the following definition for the cacheUserToken method. This method stores the user id and token in a preference file that is marked private. This should protect access to the cache so that other apps on the device do not have access to the token because the preference is sandboxed for the app. However, if someone gains access to the device, it is possible that they may gain access to the token cache through other means.

```
private void cacheUserToken(MobileServiceUser user)
{
    SharedPreferences prefs = getSharedPreferences(SHAREDPREFFILE, Context.MODE_PRIVATE);
    Editor editor = prefs.edit();
    editor.putString(USERIDPREF, user.getUserId());
    editor.putString(TOKENPREF, user.getAuthenticationToken());
    editor.commit();
}
```

4. You can further protect the token with encryption if token access to your data is considered highly sensitive and someone may gain access to the device. However, a completely secure solution is beyond the scope of this tutorial and dependent on your security requirements.

5. In the ToDoActivity.java file, add the the following definition for the loadUserTokenCache method.

```
private boolean loadUserTokenCache(MobileServiceClient client)
{
    SharedPreferences prefs = getSharedPreferences(SHAREDPREFFILE, Context.MODE_PRIVATE);
    String userId = prefs.getString(USERIDPREF, null);
    if (userId == null)
        return false;
    String token = prefs.getString(TOKENPREF, null);
    if (token == null)
        return false;

    MobileServiceUser user = new MobileServiceUser(userId);
    user.setAuthenticationToken(token);
    client.setCurrentUser(user);

    return true;
}
```

6. In the ToDoActivity.java file, replace the authenticate method with the following method which uses a token cache.

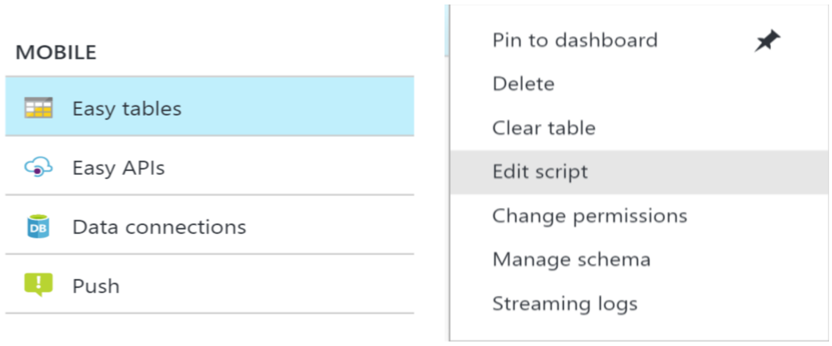
```
private void authenticate() {
    // We first try to load a token cache if one exists.
    if (loadUserTokenCache(mClient))
    {
        createTable();
    }
    // If we failed to load a token cache, login and create a token cache
    else
    {
        // Login using the Google provider.
        ListenableFuture<MobileServiceUser> mLogin =
mClient.login(MobileServiceAuthenticationProvider.Facebook);

        Futures.addCallback(mLogin, new FutureCallback<MobileServiceUser>() {
            @Override
            public void onFailure(Throwable exc) {
                createAndShowDialog("You must log in. Login Required", "Error");
            }
            @Override
            public void onSuccess(MobileServiceUser user) {
                createAndShowDialog(String.format(
                        "You are now logged in - %1$2s",
                        user.getUserId()), "Success");
                cacheUserToken(mClient.getCurrentUser());
                createTable();
            }
        });
    }
}
```

7. Build the app and test authentication using a valid account. Run it at least twice. During the first run, you should receive a prompt to login and create the token cache. After that, each run will attempt to load the token cache for authentication and you should not be required to login. The finial code should look like week07.java in your lib file.

Task 4: Service-side authorization of users in Mobile Services

This topic shows you how to use server-side scripts to authorize users. In this tutorial, you register scripts with Azure Mobile Services, filter queries based on user IDs, and give users access to only their own data. Filtering a user's query results by the user ID is the most basic form of authorization. Depending on your specific scenario, you might also want to create Users or Roles tables to track more detailed user authorization information, such as which endpoints a given user is permitted to access.



1. Log on to the Azure portal, click Mobile Services, and then click on your mobile service. Click the Easy tables tab, and select your table and then click Edit script.
2. In the popped up new view, replace the existing script with the following:

```
/*
** Sample Table Definition - this supports the Azure Mobile Apps
** TodoItem product with authentication and offline sync
*/
var azureMobileApps = require('azure-mobile-apps');
// Create a new table definition
var table = azureMobileApps.table();

// Defines the list of columns
table.columns = {
    "userId": "string",
    "text": "string",
    "complete": "boolean"
};
// Turns off dynamic schema
table.dynamicSchema = false;

// Must be authenticated for this to work
table.access = 'authenticated';

// Limit the viewable records to those that the user created.  This
// is used in the individual read, update and delete operations to
// ensure that one user cannot touch another users records
function limitToAuthenticatedUser(context) {
    context.query.where({ userId: context.user.id });
    return context.execute();
}

// Attach the limitation to each of the affected operations
table.read(limitToAuthenticatedUser);
table.update(limitToAuthenticatedUser);
table.delete(limitToAuthenticatedUser);

// When adding a new record, overwrite the userId with the
// id of the user so that the read/update/delete limitations
// will work properly
table.insert(function (context) {
    context.item.userId = context.user.id;
    return context.execute();
});

module.exports = table;
```

Run your app. If you finish the above tasks properly, you should be able to edit your todolist and save it to the cloud.

Notice that when you now run your client-side app, although there are items already in the TodoItem table from previous tutorials, no items are returned. This happens because previous items were inserted without the user ID column and now have null values. Verify newly added items have an associated userId value in the TodoItem table.

If you have additional login accounts, verify that users can only see their own data by closing and deleting the app and running it again. When the login credentials dialog is displayed, enter a different login and verify that items entered under the previous login are not displayed.