AZR120

Authentication with Azure App Services

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft or Xamarin may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft or Xamarin, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

Xamarin, MonoTouch, MonoDroid, Xamarin.iOS, Xamarin.Android, Xamarin Studio, and Visual Studio are either registered trademarks or trademarks of Microsoft in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

Objectives

1. Introduce authentication options
2. Require authentication in an Azure App Service
3. Log into your mobile service
4. Add support for multiple users

Introduce authentication options

Xamarin University

Tasks

1. Examine roles in authentication
2. Compare identity providers
3. Explore the server flow
4. Explore the client flow

Reasons to support authentication

- Many applications benefit from knowing *who* the client is

Provide access to user-specific data

Control access to a backend service

Ensure a piece of information is genuine

Azure authentication

- Azure uses OAuth to identify web and mobile users; OAuth defines how three participants route authentication requests and verify identity

Client
(Application)

Resource
(Azure)

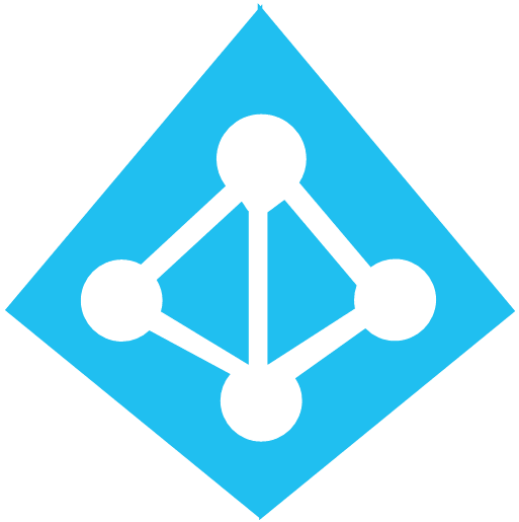
Identity Provider
(IdP)

Identity Providers [3rd party]

- Azure supports authentication through several 3rd party social providers

Identity Providers [Azure]

- Azure also supports using Azure Active Directory



Can use Azure Active Directory to do enterprise authentication

Or Azure Active Directory B2C which provides a custom user/password store in Azure

Identity Providers [Custom]

- Finally, can integrate custom identity providers which conform to the OAuth specification

Auth0

Identity Server

Check out this post for details on how to implement custom authentication with your Azure App service: http://bit.ly/2es6cR5

Which one should I choose?

- Which identity provider you support is dependent upon your app and audience, however this flow is a good one to think about

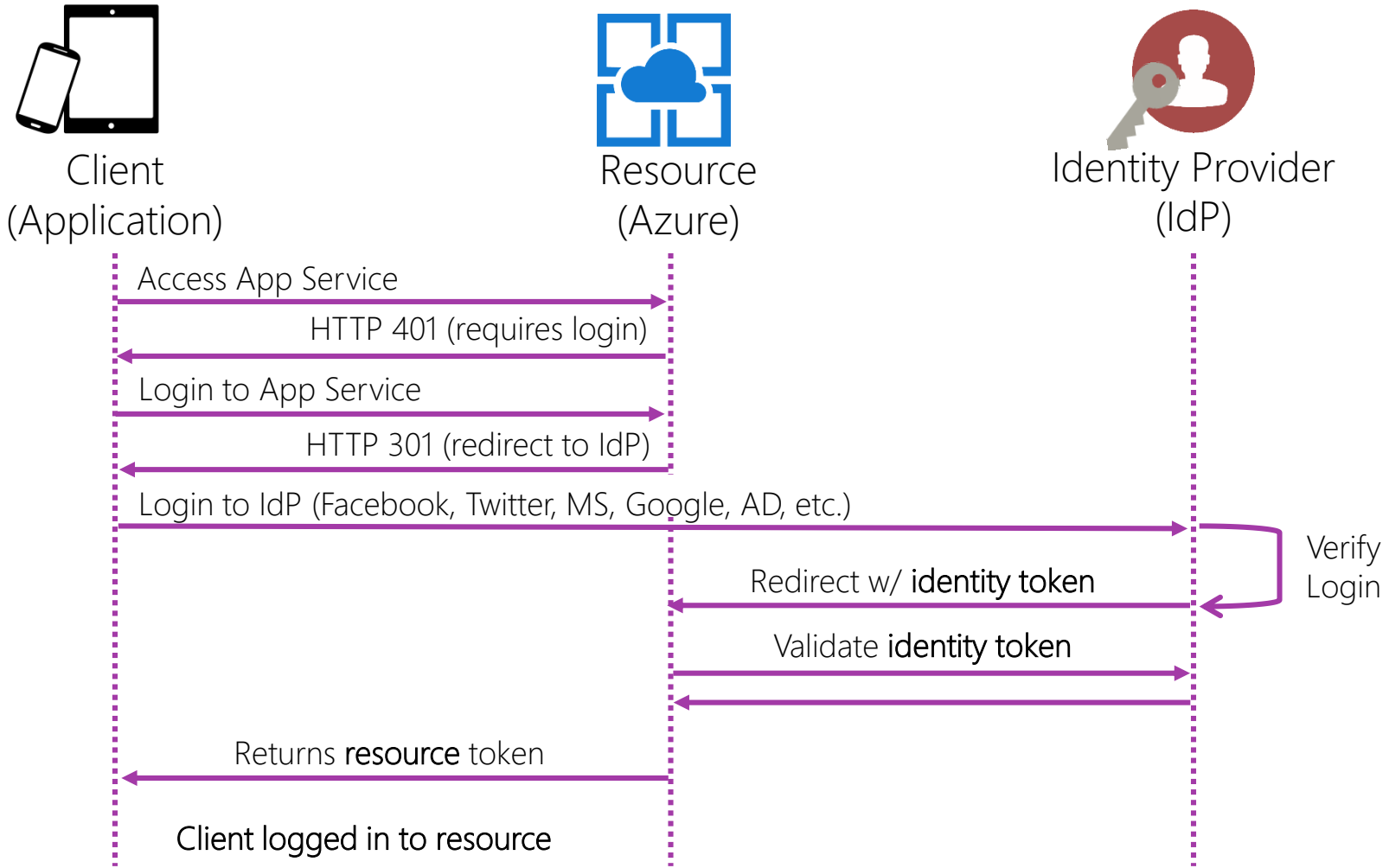
If..	Use this provider
You need enterprise authentication	Use **Azure Active Directory**
You want to use a social provider	Use that **specific provider**
You want to use *multiple* social providers	Use an aggregate provider front-end like **Auth0** or **Identity Server**
You want to use names/passwords	Use **Azure Active Directory B2C**

OAuth flows [server]

- OAuth supports two authentication flows

Server flow

- ✓ Client uses WebView to display login page from Azure
- ✓ Azure redirects to login page for specific IdP (Google, MS, Twitter, etc.)
- ✓ User logs in using IdP credentials
- ✓ IdP directs back to the client with token
- ✓ Resource validates the token and returns a new resource token to client

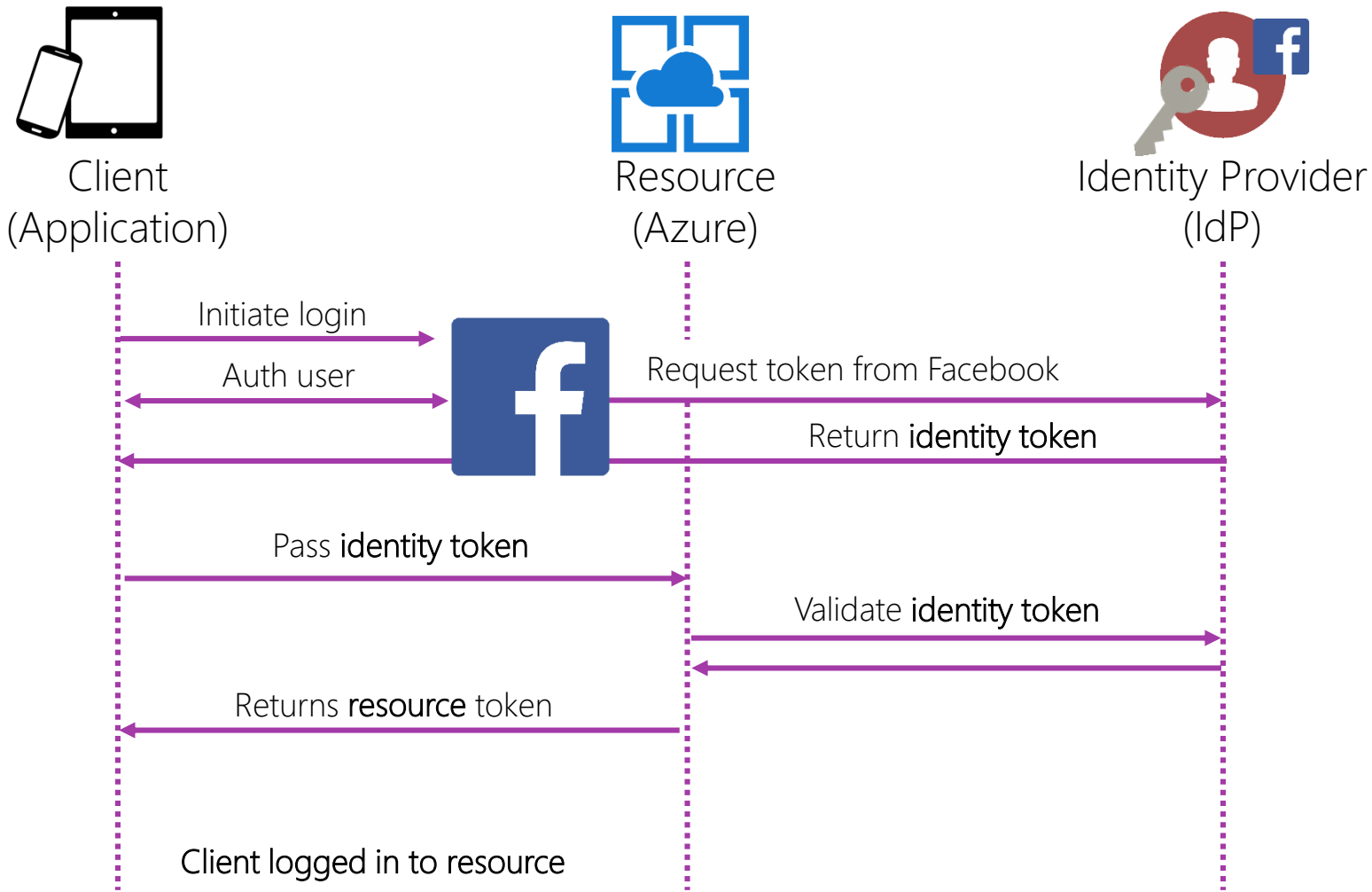

Client
(Application)
Resource
(Azure)
Identity Provider
(IdP)
Access App Service
HTTP 401 (requires login)
Login to App Service
HTTP 301 (redirect to IdP)
Login to IdP (Facebook, Twitter, MS, Google, AD, etc.)
Verify
Login
Redirect w/ identity token
Validate identity token
Returns resource token
Client logged in to resource

OAuth flows [client]

- OAuth supports two authentication flows

- ✓ Client uses the SDK to talk to the IdP
- ✓ IdP authenticates the user and returns authentication token
- ✓ Client presents auth token to the resource
- ✓ Resource validates the token with IdP
- ✓ Resource generates a resource token and returns it to the client

Client flow


Client
(Application)
Resource
(Azure)
Identity Provider
(IdP)
Initiate login
Auth user
Request token from Facebook
Return identity token
Pass identity token
Validate identity token
Returns resource token
Client logged in to resource

Resource Tokens

- Azure returns a **resource token** which is actually a JSON Web Token (JWT) that contains a set of *claims* about the user as well as the identity provider

```
{
   "stable_sid": "sid:59abaa14a1dd18e249395286365daa42",
   "sub": "sid:78ea3dc7f6cce5f8d732694bc7bcde96",
   "idp": "twitter",
   "ver": "3",
   "iss": "https://mysecurepersonaldiary.azurewebsites.net/",
   "aud": "https://mysecurepersonaldiary.azurewebsites.net/",
   "exp": 1478118914,
   "nbf": 1475526914
}
```


Note: this is the unencoded version of the token – normally the token will look like a jumbled set of characters – not clear text as shown here

Authenticating requests

- Resource token must be passed to Azure on *every* request that **requires authentication** – this is done through a special header value

Header	Value
ZUMO-API-VERSION	2.0.0
X-ZUMO-AUTH	{resource token}

Flash Quiz

Xamarin University

Flash Quiz

① If you want to manage your own password store, then you should choose _____ as your Identity Provider for Azure

a) Azure Active Directory

b) Google+

c) Auth0

d) Azure Active Directory B2C

Flash Quiz

① If you want to manage your own password store, then you should choose _____ as your Identity Provider for Azure

a) Azure Active Directory

b) Google+

c) Auth0

d) Azure Active Directory B2C

Flash Quiz

② If I want to allow my users to authenticate with *either* Twitter or Facebook, I should use _____ as my Identity Provider

a) Azure Active Directory
b) Twitter
c) Auth0
d) Azure Active Directory B2C

Flash Quiz

② If I want to allow my users to authenticate with *either* Twitter or Facebook, I should use _____ as my Identity Provider

a) Azure Active Directory

b) Twitter

c) Auth0

d) Azure Active Directory B2C

Summary

1. Examine roles in authentication
2. Compare identity providers
3. Explore the server flow
4. Explore the client flow

Require authentication in an Azure App Service

Tasks

- Turn on authentication
- Decide on the Identity Provider(s)
- Configure your Identity Provider(s)
- Control which endpoints and operations require authentication
- Use enterprise authentication

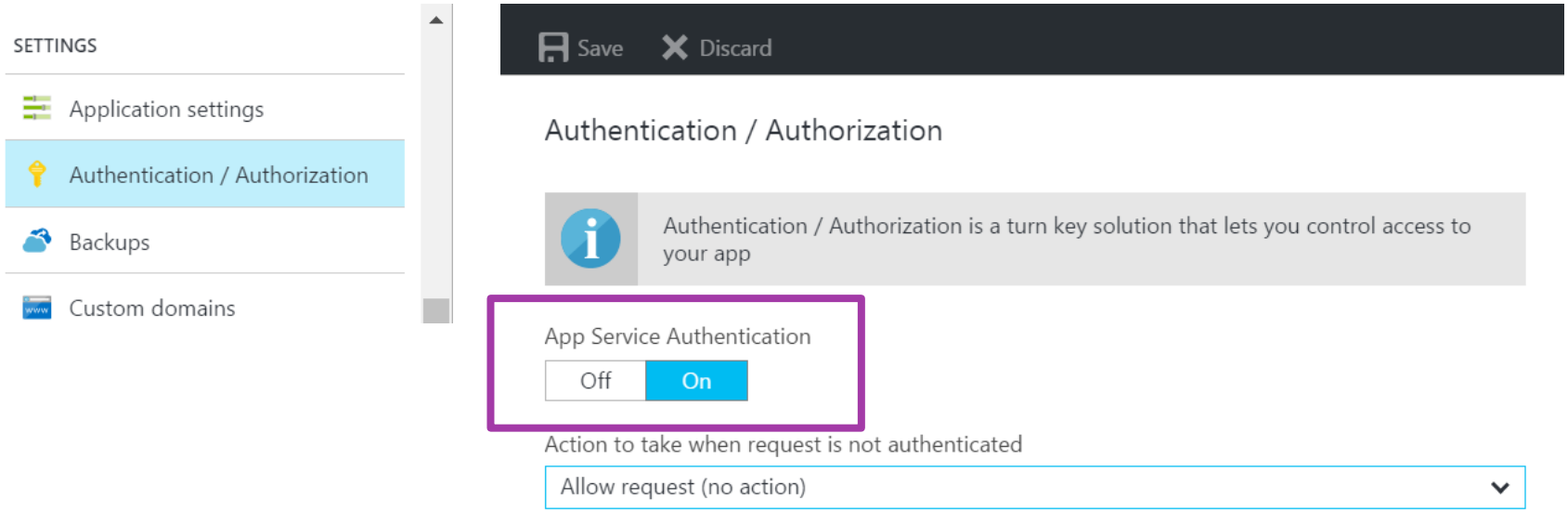
Authentication in Azure

- By default all tables and APIs on an Azure App Service allow anonymous callers; authentication is an *opt-in* feature

- You can enforce authentication for all endpoints, all operations on a specific endpoint, or for specific HTTP operation(s) on an endpoint

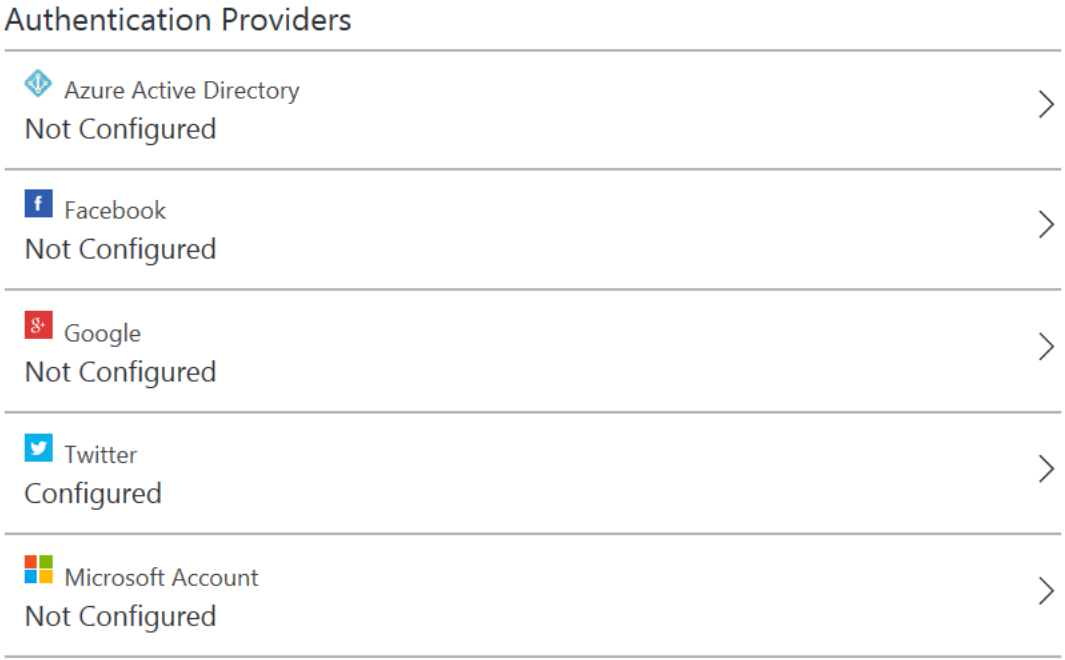
Step 1: turn on authentication

- Must tell the Azure service that it should *support* authentication



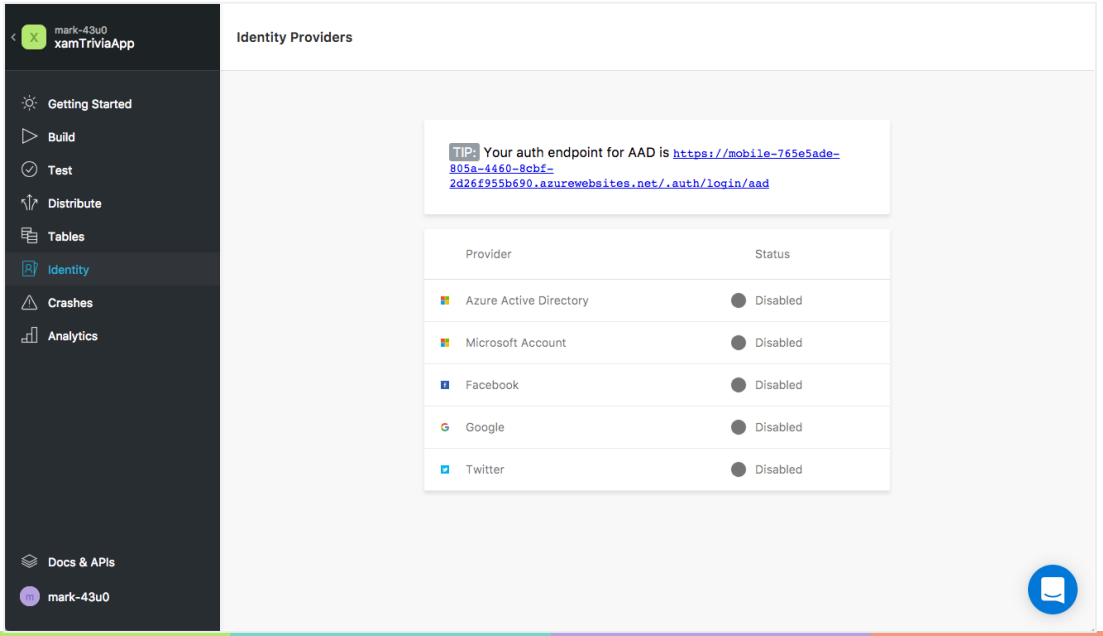
Step 2: decide which IdP(s) to use

- Next, you need to configure your Azure app to have a relationship with each Identity Provider you want to authenticate against



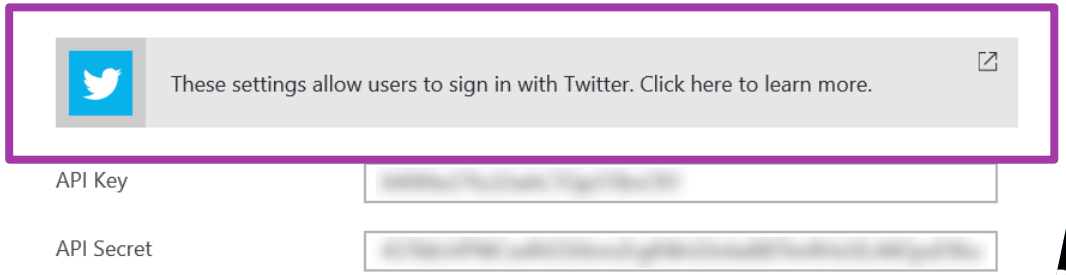
Visual Studio Mobile Center

- Mobile Center also supports setting up authentication using the supported identity providers



Configuring an IdP

- Each provider has different steps, but all involves creating an App URL association between Azure and the IdP



- Click the banner at the top of each blade to get setup instructions

OAuth endpoints

- Azure exposes unique OAuth redirect (callback) URLs for each provider – this URL must start with **`https://`**

Social provider	Endpoint
Facebook	`<site>/.auth/login/facebook/callback`
Twitter	`<site>/.auth/login/twitter/callback`
Google	`<site>/.auth/login/google/callback`
Microsoft	`<site>/.auth/login/microsoftaccount/callback`
Azure AD	`<site>/.auth/login/aad/callback`

OAuth endpoints [Twitter]

Application Details

Name *

Secure Task List

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

Tasky Pro

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

https://tasky.azurewebsites.net

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

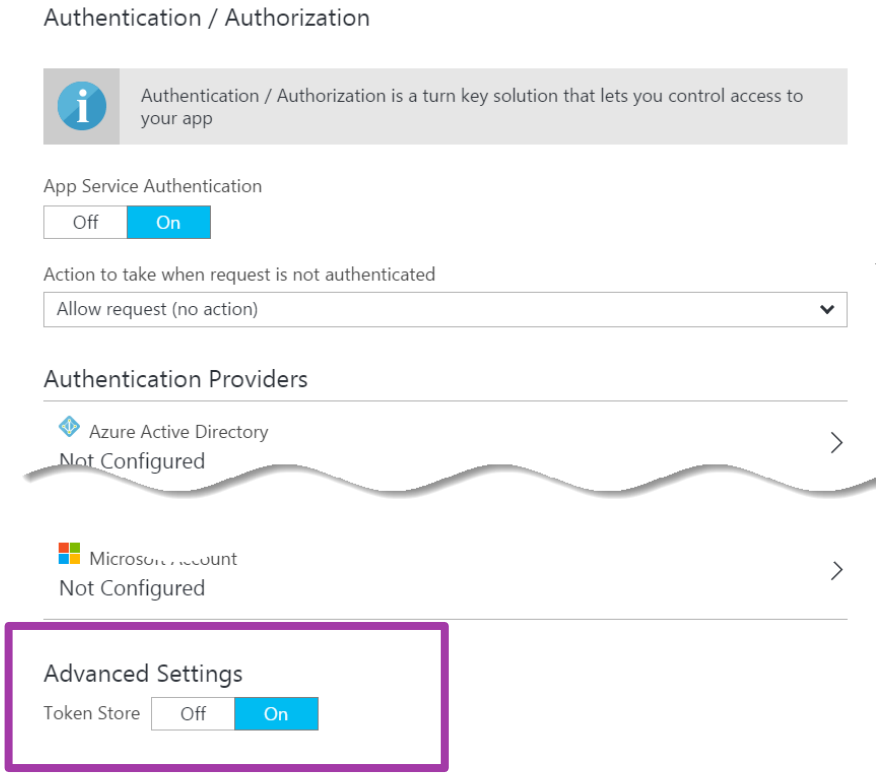
Callback URL

https://tasky.azurewebsites.net/.auth/login/twitter/callback

Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Step 3: enable the token store

- Must enable the token store option for mobile apps; this allows Azure to cache off the underlying token it will negotiate with the identity provider
- This should be enabled by default, but if not, make sure it's turned on



Testing your authentication

- Can test Identity Provider setup through a browser or REST client by hitting a specific URL

Social provider	Endpoint
Facebook	`<site>/.auth/login/facebook`
Twitter	`<site>/.auth/login/twitter`
Google	`<site>/.auth/login/google`
Microsoft	`<site>/.auth/login/microsoftaccount`

Browser should be redirected to the proper identity provider's login page

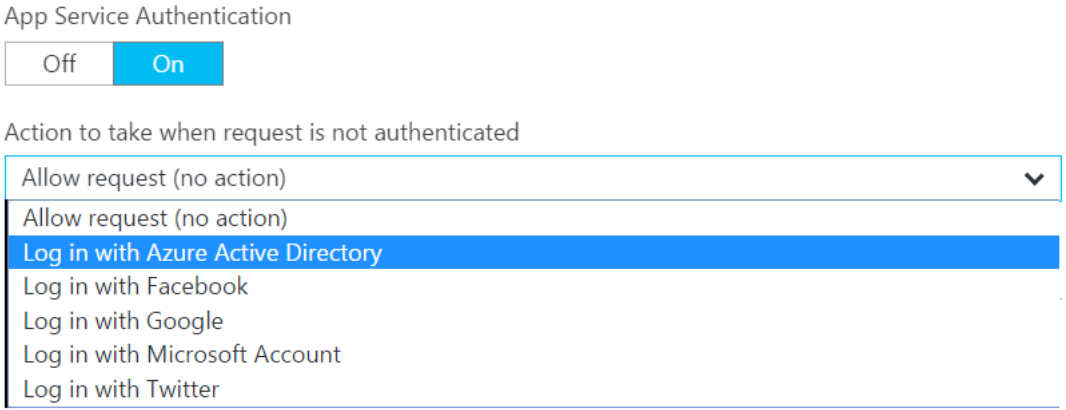
Demonstration

Configure an Identity Provider with Azure

Xamarin University

Step 4: decide when to authenticate

- Can force *all* endpoints + operations to require authentication by selecting option in dropdown



Note: Most public services should set this to **"Allow request (no action)"** to allow for flexibility in the choice of the identity provider

Enable authentication

- Rather than enforcing authentication on *all* endpoints, can turn it on individually or even for specific operations in the server code

Notice that it's the *server* that makes this decision – it is always the server that decides which endpoints and operations need authentication

Authentication in ASP.NET service

- **`ApiController`** and **`TableController<T>`** support authentication by applying an **`[Authorize]`** attribute

Can require all actions be authenticated

```
[Authorize]
public class DiaryController : TableController<DiaryEntry>
```

… Or just require **specific operations** be authenticated

```
[Authorize]
public Task<DiaryEntry> PostDiaryEntry(DiaryEntry item) ...
```

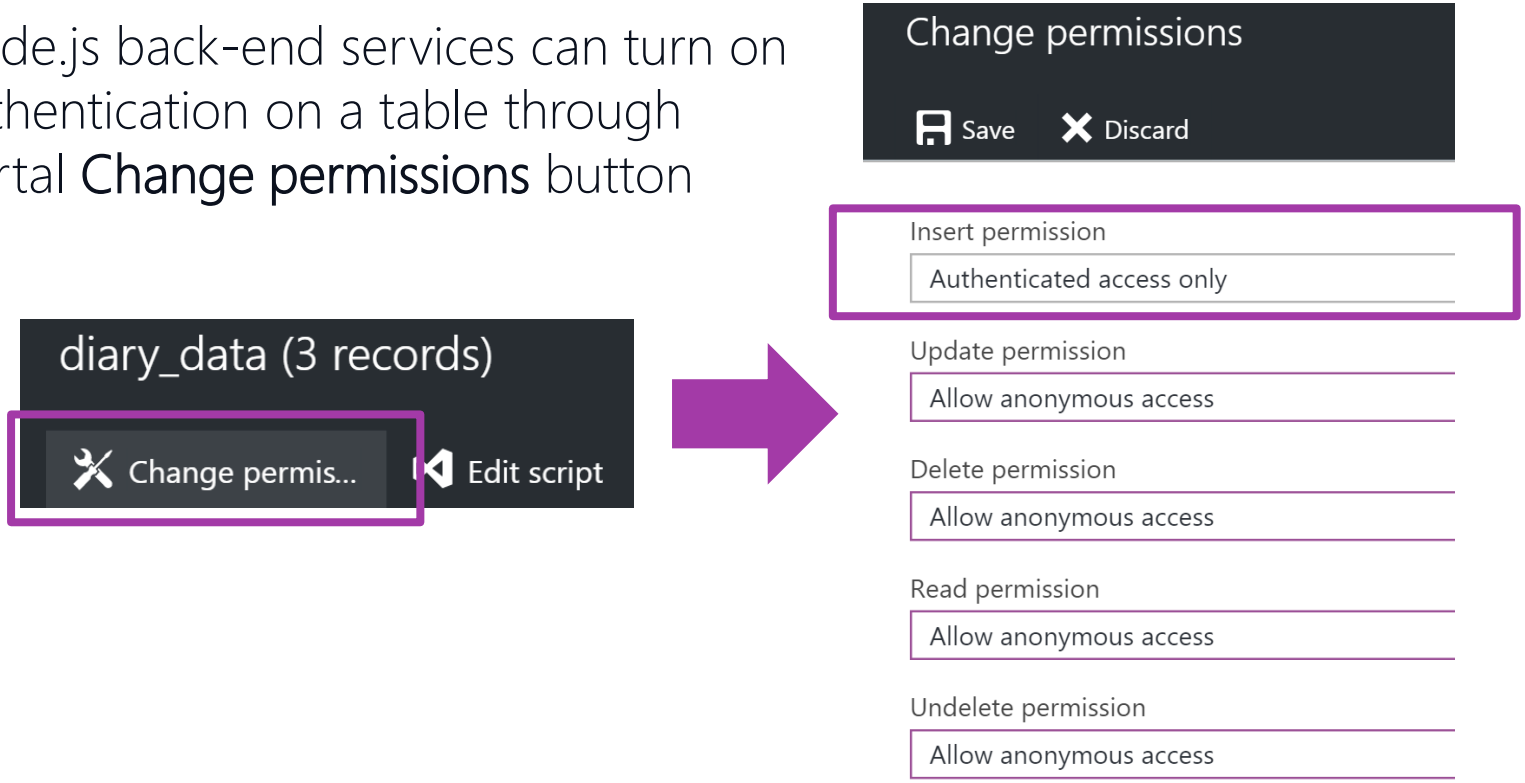
Authentication in ASP.NET service

- Can turn *off* authentication for an endpoint or action with the `[AllowAnonymous]` attribute

```
[Authorize]
public class DiaryController : TableController<DiaryEntry>
{
    ...
    [AllowAnonymous]
    public Task<DiaryEntry> PostDiaryEntry(DiaryEntry item)
    { ... }
}
```

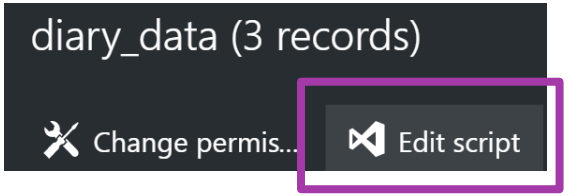
Authentication in node.js service

- Node.js back-end services can turn on authentication on a table through portal **Change permissions** button



Authentication in node.js service

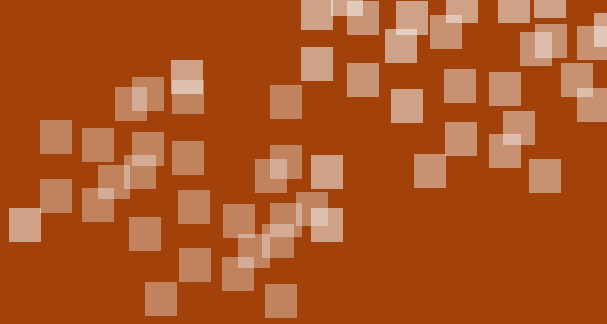
- Can also set the **access** property on the table in the **node.js** configuration for the table to force it to be authenticated; can do it globally or per operation (**table.read.access**, **table.update.access**, etc.)



diary_data.js tables

```
1
2 var table = module.exports = require('azure-mobile-apps
3 // Users must be authenticated to access this table.
4 table.access = 'authenticated';
5
```

access property can be **"anonymous"** (default), **"authenticated"** to force each request to have an authorization token, or **"disabled"** to disable access to the table



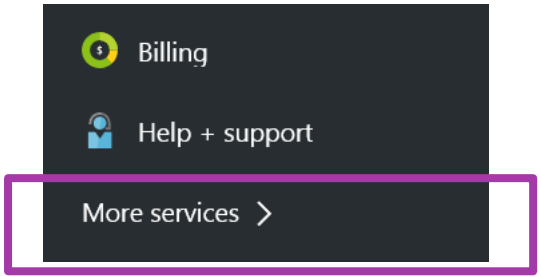
Demonstration

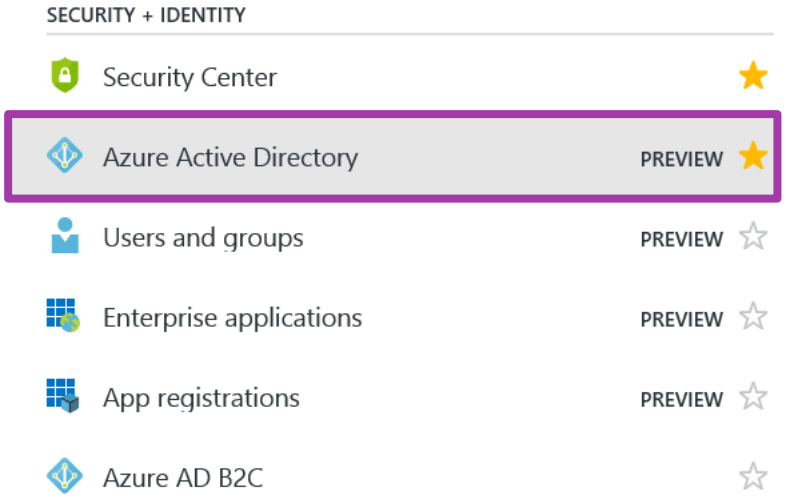
Configure a table to require authentication

Xamarin University

Enterprise authentication

- Enterprise authentication is handled by Azure Active Directory which is included as part of your Azure subscription or Office 365

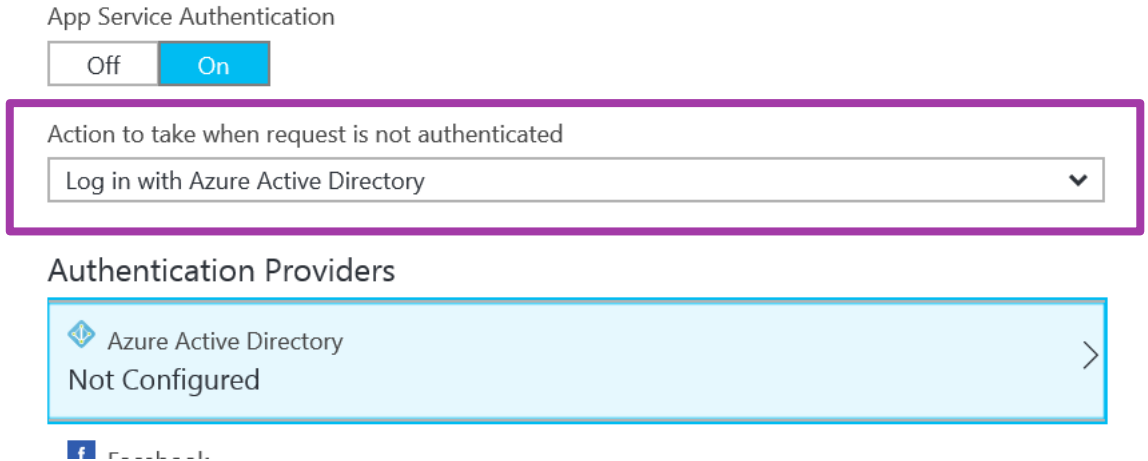




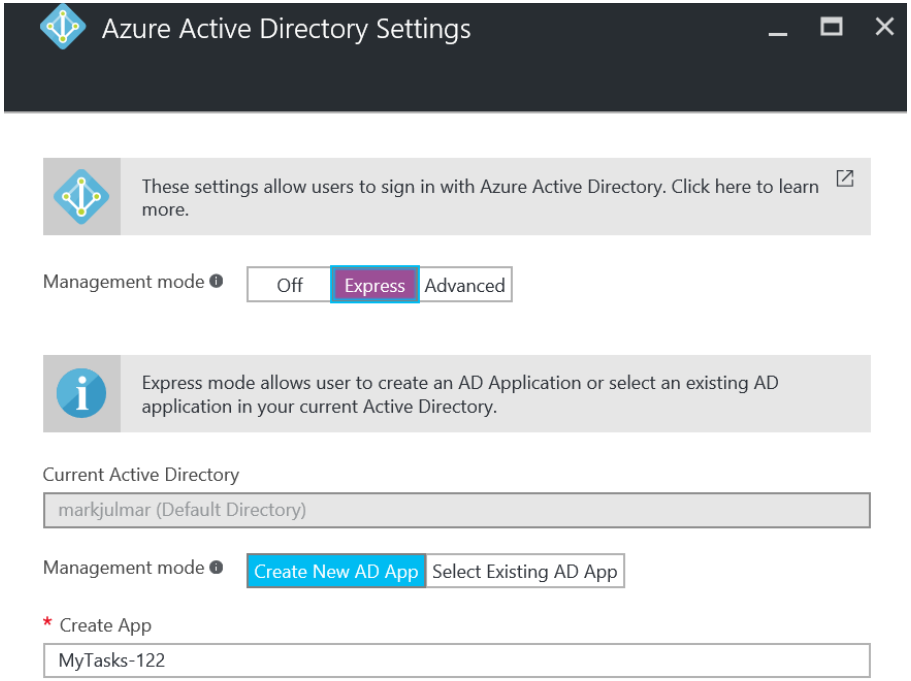
AD configuration is available in the services sidebar on the new management portal as a preview, or can use the classic portal (manage.windowsazure.com)

Configuring enterprise authentication

- Unlike social identity providers, enterprise based authentication often mandates that *all* endpoints are secured; this is not a requirement but is often the way the app is configured

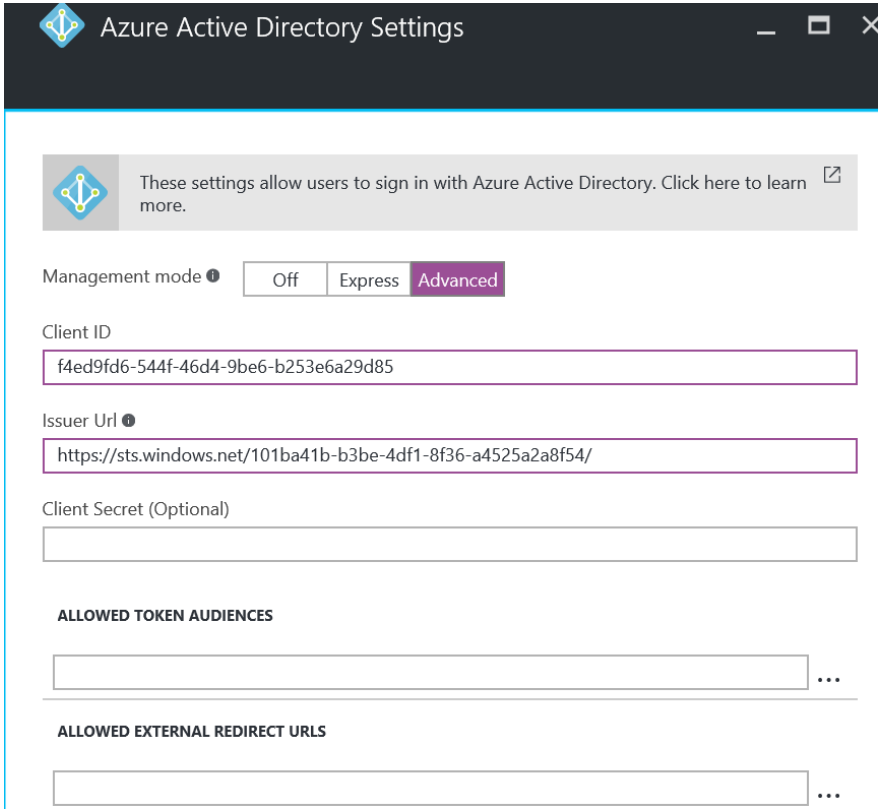


Configuring enterprise authentication



- Can use **Express** mode to easily configure to your orgs AD – this fills in all the details for you

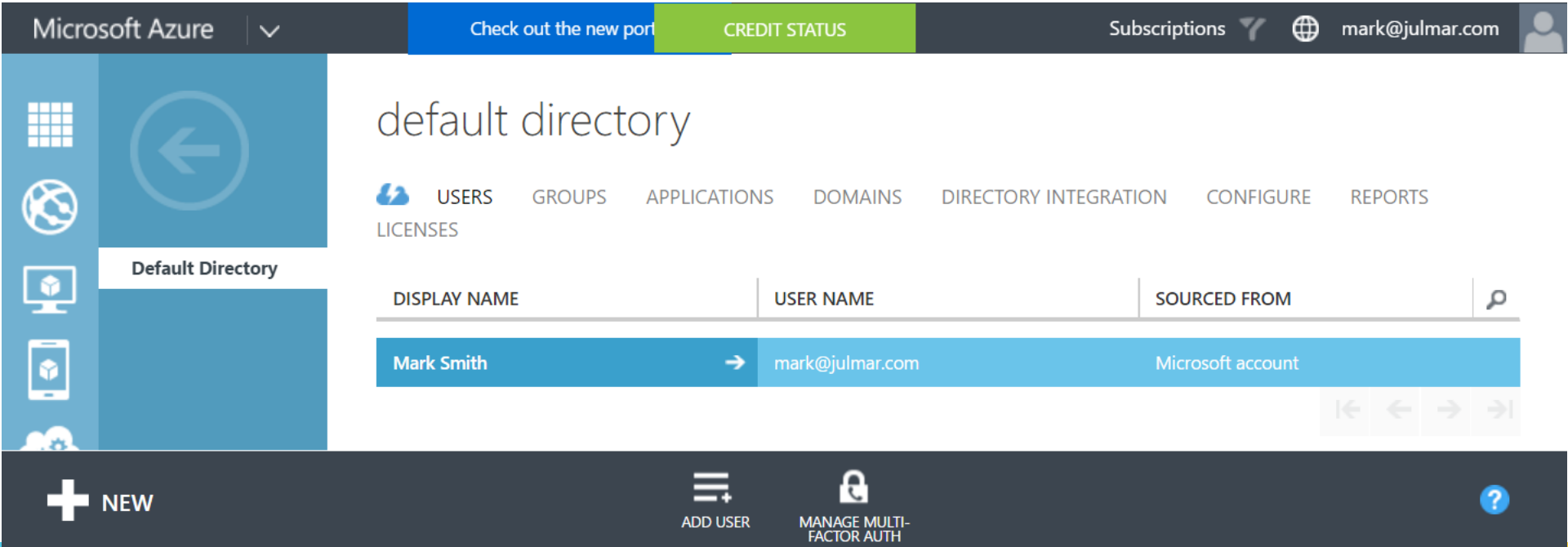
Configuring enterprise authentication

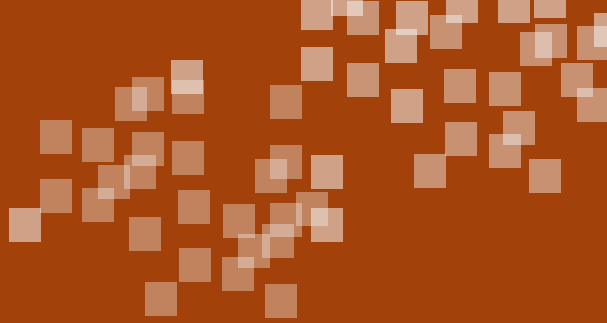


- Use **Advanced** to configure the app to manually configure the app to use a different AD tenant than the one you sign into (where you have more than one directory)
- See http://bit.ly/2f1x3oa for details on how to configure this

Creating users in Azure AD

- Currently, only the classic portal (manage.windowsazure.com) supports *modifying* the directory data; use this portal to add users





Demonstration

Use Enterprise authentication

Xamarin University

Summary

- Turn on authentication
- Decide on the Identity Provider(s)
- Configure your Identity Provider(s)
- Control which endpoints and operations require authentication
- Use enterprise authentication

Log into your mobile service

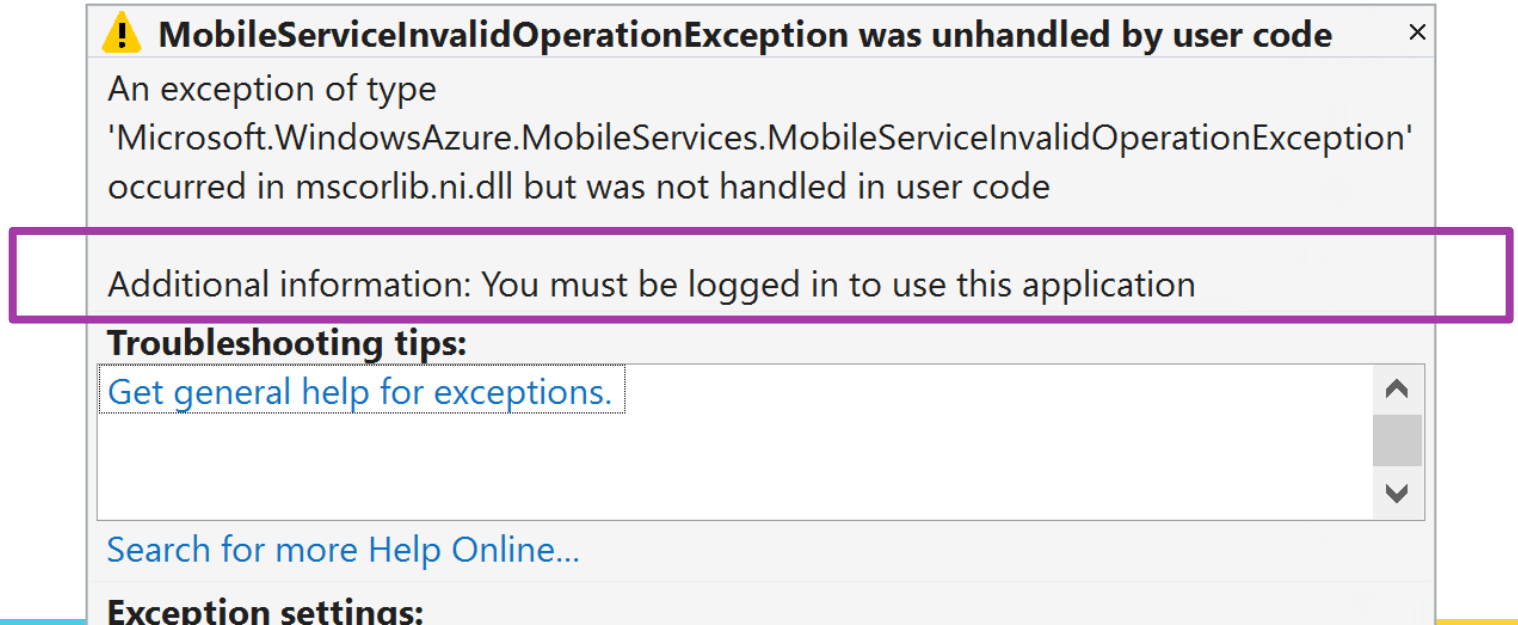
Xamarin University

Tasks

- Requesting authentication
- Segregating out platform-specific code
- Identifying the user
- Dealing with token expiration
- Refreshing tokens

Unauthorized access

- Error is returned when endpoint that requires authentication is hit without the proper authentication header



Selecting an OAuth flow

- Client can choose which OAuth flow it wants to utilize

Client flow

Server flow

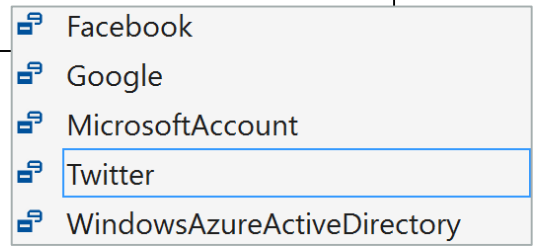
Note: Google has announced that they are disabling support for the server flow OAuth model (http://bit.ly/2hJvNXo); if you plan to support Google as an IdP, then it will need to use the client flow

Requesting authentication

- **`MobileServiceClient`** has support to provide a Login UI for several 3rd party OAuth providers through the **`LoginAsync`** method

```
MobileServiceClient MobileService = ...;

MobileServiceUser User = await MobileService.LoginAsync(...
            MobileServiceAuthenticationProvider.
```



Client decides which identity provider to use – must be one that's usable with the service or an error is returned

Requesting authentication

- **`LoginAsync`** method is *platform-specific* and each platform has different parameters; can use partial classes or **`#if`** conditionals in Shared Projects

```
MobileServiceClient client = ...;

#if __ANDROID__
await client.LoginAsync(Forms.Context,
       MobileServiceAuthenticationProvider.Twitter);
#elif __IOS__
await client.LoginAsync(UIApplication.SharedApplication
           .KeyWindow.RootViewController,
       MobileServiceAuthenticationProvider.Twitter);
#endif
```

Abstracting login out to a service

- When using PCLs for shared code, **`LoginAsync`** method will not be present and must be abstracted out to a service which is implemented by the native app code

PCL

```
public interface ILoginProvider
{
    Task LoginAsync(MobileServiceClient client);
}
```

```
class LoginProviderDroid : ILoginProvider
```

```
class LoginProvideriOS   : ILoginProvider
```

```
class LoginProviderWin   : ILoginProvider
```

```
using System.Threading.Tasks;
using Microsoft.WindowsAzure.MobileServices;

namespace Services
{
   class LoginProviderDroid : ILoginProvider
   {
      Task LoginAsync(MobileServiceClient client,
                      MobileServiceAuthenticationProvider provider)
      {
         return client.LoginAsync(Xamarin.Forms.Forms.Context, provider);
      }
   }
}
```

```
using System.Threading.Tasks;
using Microsoft.WindowsAzure.MobileServices;

namespace Services
{
   class LoginProvideriOS : ILoginProvider
   {
      Task LoginAsync(MobileServiceClient client,
                      MobileServiceAuthenticationProvider provider)
      {
         return client.LoginAsync(
                   UIApplication.SharedApplication
                        .KeyWindow.RootViewController, provider);
      }
   }
}
```

```
using System.Threading.Tasks;
using Microsoft.WindowsAzure.MobileServices;

namespace Services
{
   class LoginProviderWin : ILoginProvider
   {
      Task LoginAsync(MobileServiceClient client,
                      MobileServiceAuthenticationProvider provider)
      {
         return client.LoginAsync(provider);
      }
   }
}
```

Locating the ILoginProvider

- Can use dependency injection/location to get access to the platform-specific login provider in shared PCL code

```
MobileServiceClient client = ...;
...
public Task LoginAsync()
{
    var loginProvider = DependencyService.Get<ILoginProvider>();
    return loginProvider.LoginAsync(client);
}
```

Here we are using Xamarin.Forms **DependencyService**, but any IoC approach would work to retrieve the dependency in your shared code

Using the client flow

- Can integrate more tightly to an identity provider through the client flow; this requires that you utilize a specific IdP SDK *and* have that social application installed on the device

```
public Task<string> LoginFacebookAsync() { ... } // Provider specific
```

```
public async Task LoginAsync()
{
    var accessToken = await LoginFacebookAsync();
    var tokenData = new JObject();
    tokenData["access_token"] = accessToken;
    return client.LoginAsync(...,
            MobileServiceAuthenticationProvider.Facebook, tokenData);
}
```

Where do I get the client SDKs?

- Client SDKs are all distributed as NuGet packages which must be added to your projects

SDK	NuGet package identifier
Azure Active Directory	Microsoft.IdentityModel.Clients.ActiveDirectory
Facebook	Xamarin.Facebook.iOS and Xamarin.Facebook.Android
Google	Google.Apis.Core

Twitter does not have a specific library for .NET / Xamarin, but there are several 3rd party versions which will work – check out https://dev.twitter.com/overview/api/twitter-libraries

Exercise

Add support to login to an Azure App Service

Xamarin University

Identifying the user

- **`LoginAsync`** returns a **`MobileServiceUser`** to represent the authenticated user

```
Task<MobileServiceUser> LoginAsync(...,
    MobileServiceAuthenticationProvider provider);
```

```
try {
   MobileServiceUser user = await client.LoginAsync(...,
       MobileServiceAuthenticationProvider.Twitter);
   ...
}
catch (Exception ex) { ... failed }
```

Exploring the user object

- User is represented by a unique **user id** from the identity provider and a **resource token** from the Azure App service

```
public class MobileServiceUser
{
    public virtual string UserId { get; set; }
    public virtual string MobileServiceAuthenticationToken { get; set; }
}
```

Notice that these properties are *settable* – this is how we can authenticate in the future *without* user interaction

Getting (and setting) the logged on user

- **`MobileServiceClient`** has a **`CurrentUser`** property which exposes the authenticated user object; it is set after a successful login

```
MobileServiceClient client = ...;

if (client.CurrentUser != null) {
    // Authenticated
}
else {
   await client.LoginAsync(...);
}
```

Restoring the user state

- Can restore the logged on user by setting the **`CurrentUser`** property before making an authenticated API call

```
void RestoreLogin(string id, string token) {
    var user = new MobileServiceUser(id) {
        MobileServiceAuthenticationToken = token
    };

    client.CurrentUser = user;
}
```

Persisting the user info

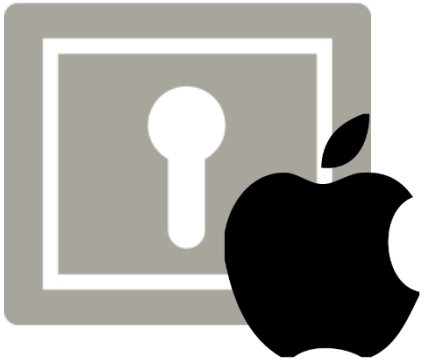
- Must *securely* store the id and token information locally
- These values allow access to the service and should be treated as confidential information
- Several possible approaches can be used – check out **ENT170** for full details

Using native APIs

- Each platform has built-in APIs to protect secrets (commonly used for passwords); can use these APIs to protect your user data

PasswordVault

KeyStore



KeyChain

Cross-platform wrappers

- Several wrappers exist which provide cross-platform support to securely store key/value pairs

Xamarin.Auth
(covered in ENT170)

sameerIOTApps.Plugin.SecureStorage

Cross-platform wrapper

- **Xamarin.Auth** is a cross-platform, open-source NuGet component which can be used to store strings securely using an **`AccountStore`**

```
void SavedCurrentUser()
{
    MobileServiceUser user = client.CurrentUser; //Logged on user
    var account = new Account() { Username = user.UserId };
    account.Properties.Add("token",
         user.MobileServiceAuthenticationToken);
    AccountStore store = AccountStore.Create();
    store.Save(account, "MyAppName");
}
```

Cross-platform wrapper

- Can retrieve saved credentials on next launch by retrieving **`Account`**

```
void RestoreCurrentUser()
{
    AccountStore store = AccountStore.Create();
    var accounts = store.FindAccountsForService("MyAppName");
    var account = accounts?.FirstOrDefault();
    if (account != null) {
        string token;
        if (account.Properties.TryGetValue("token", out token)) {
            client.CurrentUser = new MobileServiceUser(account.Userid) {
                MobileServiceAuthenticationToken = token
            };
        }
    }
}
```

Exercise

Restore the user credentials

Xamarin University

Expiring tokens

- JSON Web Tokens can (and almost always do) have an *expiration date* associated with them
- Lifetime is often fairly short (1 hour) and determined by the Identity Provider
- Token can even expire while app is running – Azure responds with **401 – Unauthorized** when this happens

Detecting expired tokens

- Expiration date is coded into the JWT as # of seconds since 1/1/1970

```
{
   "stable_sid": "sid:59abaa14a1dd18e249395286365daa42",
   "sub": "sid:78ea3dc7f6cce5f8d732694bc7bcde96",
   "idp": "twitter",
   "ver": "3",
   "iss": "https://mysecurepersonaldiary.azurewebsites.net/",
   "aud": "https://mysecurepersonaldiary.azurewebsites.net/",
   "exp": 1478118914,
   "nbf": 1475526914
}
```


Note: Remember that the token is actually BASE64 encoded and requires a little JSON parsing before you can extract the expiration from it

Putting it all together

- Can check token expiration as part of login step

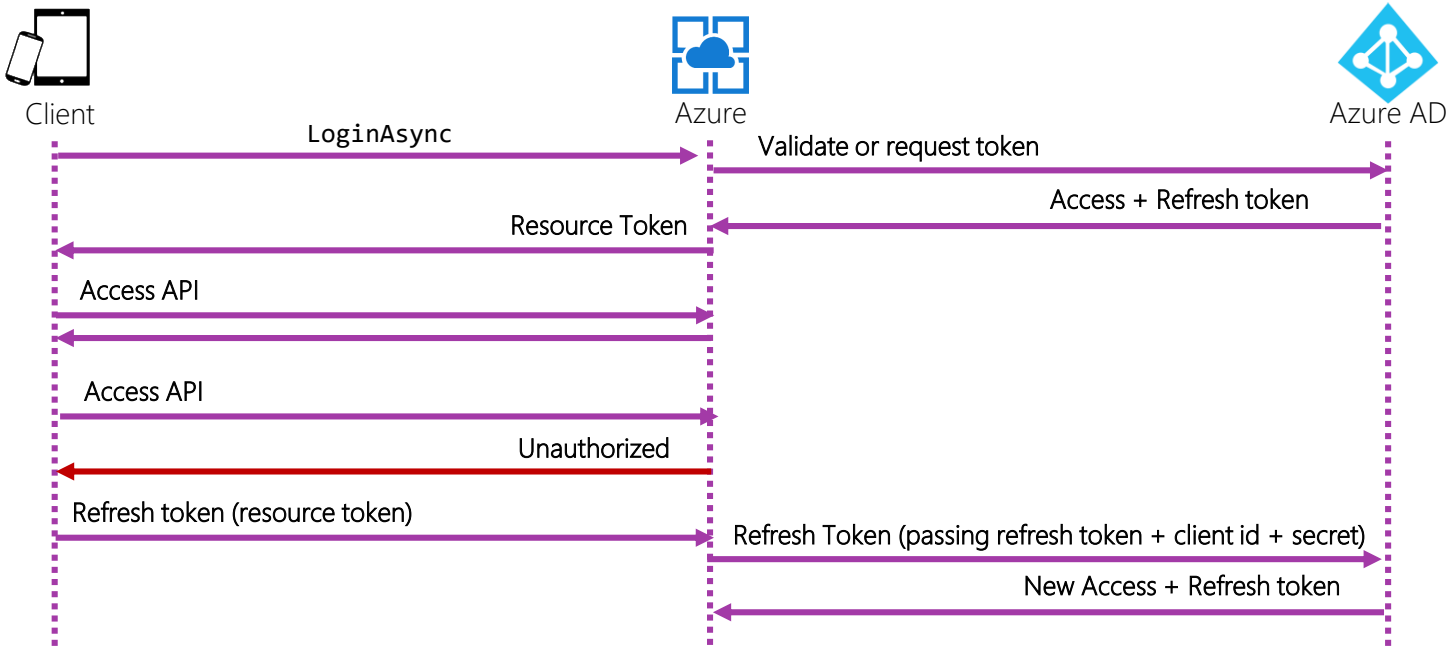
```
async Task LoginAsync()
{
    RestoreCurrentUser();
    if (client.CurrentUser != null &&
        !IsTokenExpired(client.CurrentUser
                        .MobileServiceAuthenticationToken))
        return;

    await client.LoginAsync(...);
    SaveCurrentUser();
}

bool IsTokenExpired(string token) { ... } // method to check token
```

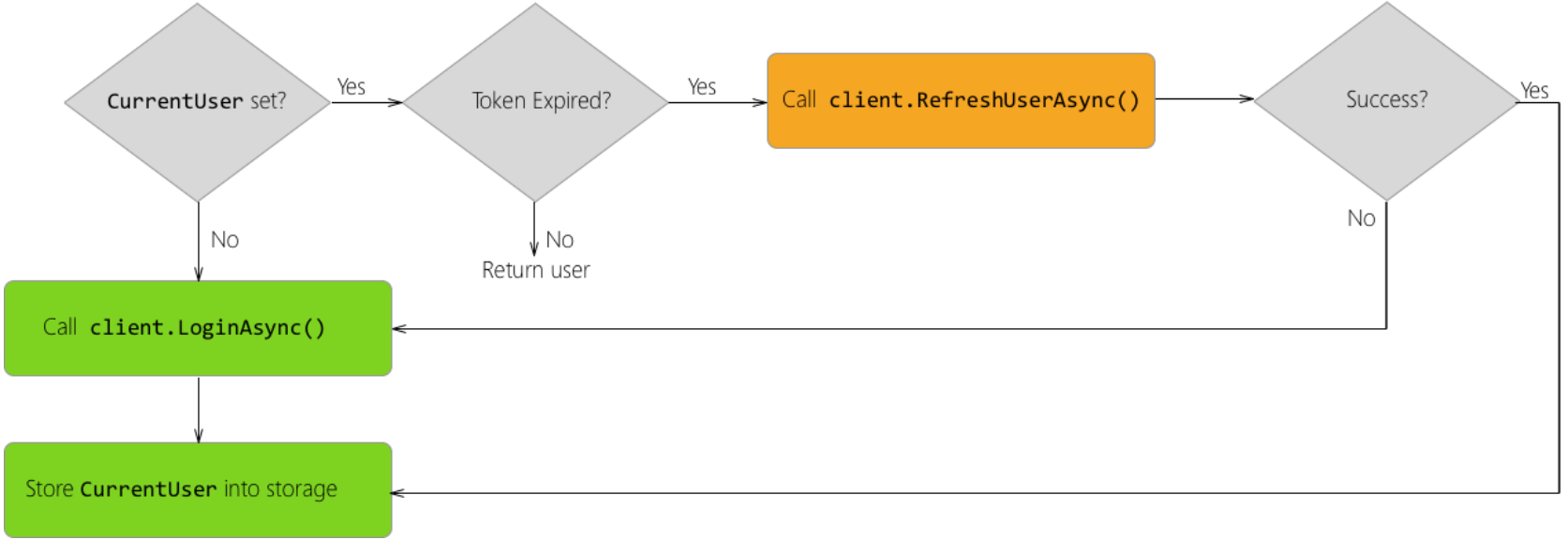
Refreshing tokens

- OAuth allows for longer-lived tokens, called *refresh tokens*, which can be passed back to the Identity Provider to get a new access token



What should I do with an expired token?

- When the token expires, all authenticated requests will fail; must either go back through a login (e.g. get a new token), or *refresh* the current token



Refreshing tokens

- Calling **`RefreshUserAsync`** will attempt to refresh the current user + token from the Identity Provider – however this feature is only supported by specific providers *and* only if you enable it in the provider!

Note: this method will call an **/.auth/refresh** endpoint on your Azure server; if you implement custom auth, then you should implement this endpoint to support refresh

Configuring refresh tokens [Google]

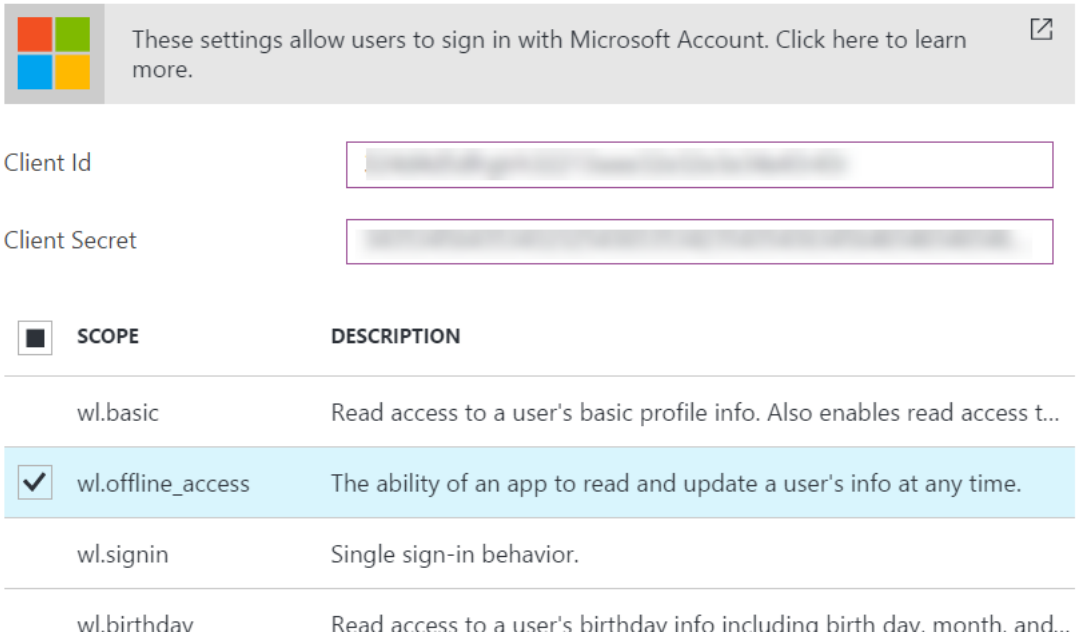
- Can request a refreshable token with Google by adding an **access type** to the server-flow login request

```
client.LoginAsync(..., MobileServiceAuthenticationProvider.Google,
     new Dictionary<string,string> {
        { "access_type", "offline" }
});
```

Dictionary values are passed as query string parameters to the login endpoint

Configuring refresh tokens [MSA]

- Must select the **wl.offline_access** scope in the Azure management portal to enable refresh tokens with MSA



Each identity provider has different configuration and setup requirements to support refresh tokens; consult the documentation for each one you want to support

Token refresh notes

- If the refresh fails (**400** or **401**) you must go back through a full login flow
- Tokens that has been expired > 72 hours cannot be refreshed and will return a **401 - Unauthorized**
- Access tokens can also be revoked; in this case you get a **403 – Forbidden** which requires a full login

Logging off [client]

- Use the **LogoutAsync** method to clear the **MobileServiceClient** info

```
await client.LogoutAsync();
Debug.Assert( client.CurrentUser == null );
```

Can also just set the current user to **null** – this is essentially what the method does today.

Don't forget to clear any locally cached token versions from secure local storage.

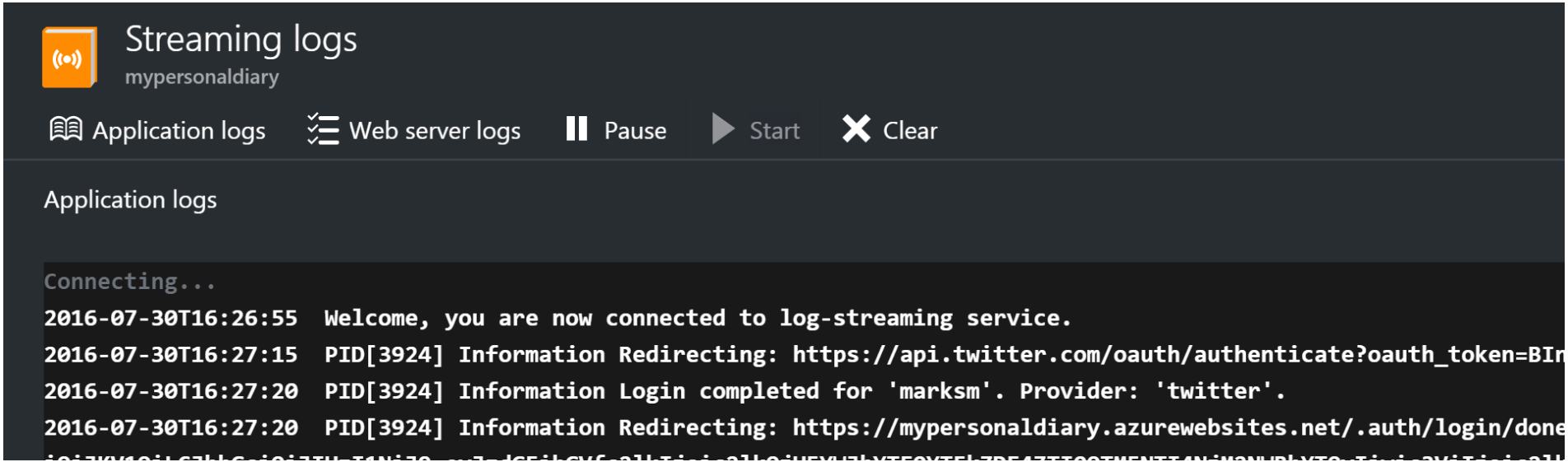
Logging off [server]

- Can call the /.auth/logout endpoint to remove the tokens from the server

```
// Remove the token on the service
var uri = new Uri($"{client.MobileAppUri}/.auth/logout");
using (var httpClient = new HttpClient()) {
   httpClient.DefaultRequestHeaders.Add("X-ZUMO-AUTH",
     client.CurrentUser.MobileServiceAuthenticationToken);
   await httpClient.GetAsync(uri);
}
// Remove it on the client
await client.LogoutAsync();
```

Troubleshooting logins

- Can use the application streaming log to monitor real-time access requests as well as the URL transitions between your service and the provider



Demonstration

Add support to refresh tokens

Xamarin University

Summary

- Requesting authentication
- Segregating out platform-specific code
- Identifying the user
- Dealing with token expiration
- Refreshing tokens

Add support for multiple users

Xamarin University

Tasks

1. Identifying the user on the server
2. Associating data to a specific user
3. Getting additional claim information

Authentication vs. Authorization

Authentication is the process of identifying a user and verifying they are really who they claim to be

Authorization refers to the rules that determine *what* the user can do and what they can access

Authentication vs. data visibility

- By default, Azure does not restrict any data visibility – even if the user is authenticated; in fact **no identity information** is stored in the rows by Azure

```
var data = await (from d in diaryTable
  orderby d.UpdatedAt
  select d).ToEnumerableAsync();
```

This query will return *all* records, regardless of who created them .. No identity is stored on the table by default

Uniquely identifying the user

- Can use the **`UserId`** property to identify the user – this is populated by the identity provider so as long as all users are run through the **same IdP**, it will be unique across all your users and consistent for a given user

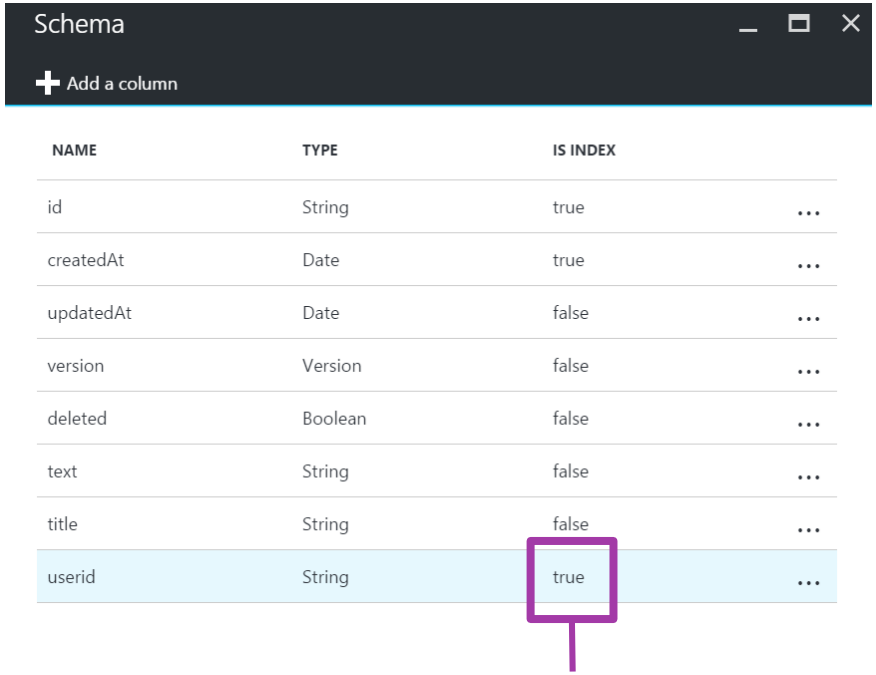
```
public class MobileServiceUser
{
    public virtual string MobileServiceAuthenticationToken { get; set; }
    public virtual string UserId { get; set; }
}
```

```
"sid:78ea3dc7f6cce5f8d732694bc7bcde96"
```

Value comes from the identity provider response, specifically the "**sub**" field

Associating rows to users

- Can add the **`UserId`** to your entity records to indicate *ownership* as it is a unique representation of this specific user
- Must add this column to your schema and then populate it as part of the insertion process



Associating data to users

- Can then **restrict your queries** on the client side to the logged on user (`client.CurrentUser.UserId`)

```
return await (from d in diaryTable
              where d.UserId == User.UserId
              orderby d.UpdatedAt
              select d).ToEnumerableAsync();
```

While this approach works, it puts the burden of security on the *client* which is not secure and easily bypassed

Moving it to the server [node.js]

- Can add **server-side logic** to your Easy tables with four interception handlers: **table.insert**, **table.read**, **table.update** and **table.delete**

```
table.read(function (context) {
    // Do something pre-execution
    return context.execute()
    // .then(function (data) {
    //     Do something post-execution
    //
    //     return data;
    // });
});
```

Function is passed a *context* which provides access to a variety of request info

Table script context

- **Context** object includes information about the item, the query, and the user, along with a bunch of other useful information and methods

user	JWT object describing the authenticated user, contains an **id** property which is an identity provider based value.
query	The DB query expressed as a queryJS object that is used to turn the OData request into a SQL query
item	The item being inserted, updated or deleted
table	Current table definition
logger	Azure mobile apps logger object
execute	Function that executes the operation; returns a promise with the results

Example: restricting data visibility

diary_data.js

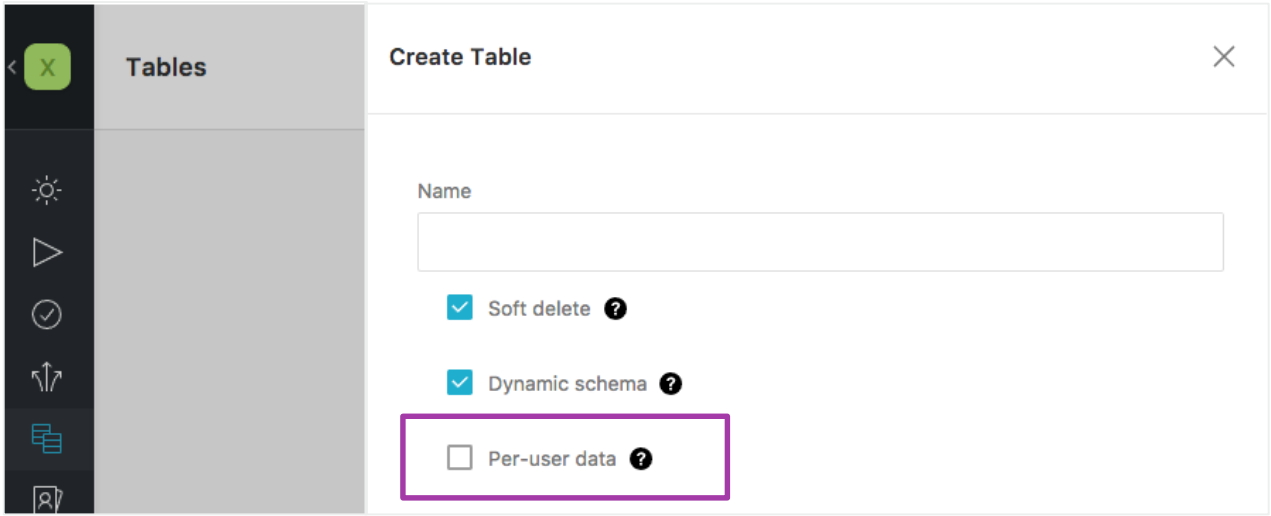
```
table.access = 'authenticated';

// Ensure that only records for the authenticated user are retrieved
table.read(function (context) {
    context.query.where({ userId: context.user.id });
    return context.execute();
});
// Make sure the user column is set to the current user
table.insert(function (context) {
     context.item.userId = context.user.id;
     return context.execute();
```


Note: This only affects the *server* queries – if you have an offline cache, it will continue to show inaccessible records until you call **PurgeAsync** on the table to get rid of them

Visual Studio Mobile Center

- Mobile Center has this support built in – just check the box when you create your Easy Table definition and it adds the **node.js** code and **userId** column for you



Moving to the server [ASP.NET]

- Use the **User** property in the service code to retrieve all the *claims* from the identity provider; can search for **`ClaimType.NameIdentifier`** to get the **`UserId`** value we see on the client

```
public string GetUserId()
{
    var user = this.User as ClaimsPrincipal;
    Claim nameClaim = user?.FindFirst(
        c => c.Type == ClaimTypes.NameIdentifier);
    return nameClaim?.Value;
}
```

Can associate this identifier to data on the server side owned by this user

Adding identity to your table [ASP.NET]

- Can add to the queries to filter the returning data on the server side

```
[Authorize]
public class DiaryController : TableController<DiaryEntry>
{
    public IQueryable<DiaryEntry> GetAll() {
      string thisUser = GetUserId();
      return Query().Where(todo => todo.UserId == thisUser);
    }

    public async Task<IHttpActionResult> Post(DiaryEntry item)
    {
       item.UserId = GetUserId();
       ...
    }
}
```

Demonstration

Add support for multiple users

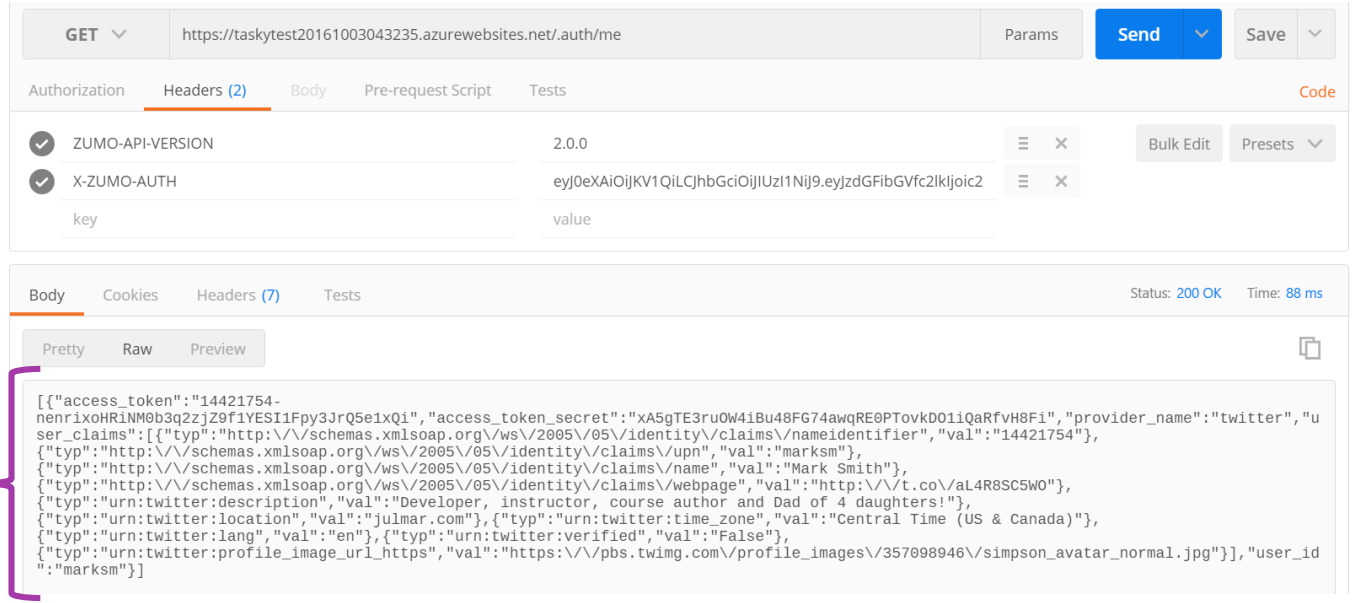
Working with other claims

- Each IdP can return a variety of claims about the user – things like their name
- If you are working with a single IdP you can rely on something more identifiable than SID
- Might need to request specific scopes or provide additional config to get to more privileged information

Getting additional claims

- Azure exposes the **/.auth/me** endpoint to retrieve all the additional claims provide by the identity provider from your mobile client

More detailed set of information was actually returned by Twitter



Example: getting the claims in your app

- Must define two data structures to hold the data – should match the JSON shape being returned from the **/.auth/me** endpoint

```
public class AzureServiceIdentity
{
    ...
    [JsonProperty(PropertyName="user_claims")]
    public List<UserClaim> UserClaims { get; set; }
}

public class UserClaim
{
    [JsonProperty(PropertyName = "typ")]
    public string Type { get; set; }
    [JsonProperty(PropertyName = "val")]
    public string Value { get; set; }
}
```

Example: getting the claims in your app

- Can then call the **auth endpoint** using the authenticated `MobileServiceClient`

```
public async Task<AzureServiceIdentity> GetClaims(MobileServiceClient client)
{
   return await client.InvokeAsync<
         List<AzureServiceIdentity>>("/.auth/me")?.FirstOrDefault();
}
```

Return the first (only) identity structure which contains the set of claims

Server-side claims [node.js]

- Can retrieve the claims from node.js with the **getIdentity** function

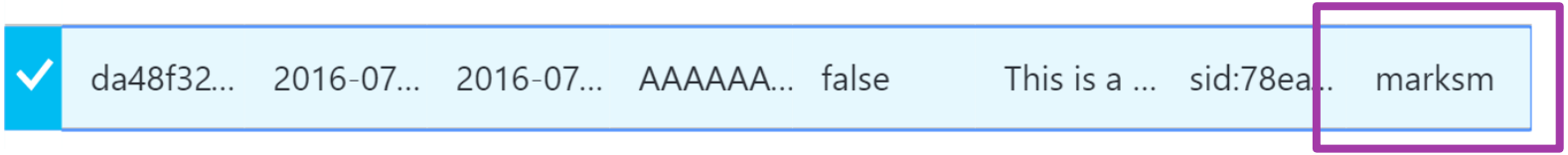
```
table.read(function (context) {
   return context.user.getIdentity().then(function (userInfo) {
     console.log('user.getIdentity = ', JSON.stringify(userInfo))
     return context.execute();
  });
});
```

Returns a JSON array of claims – here we just output them to the console

Example: using UPN from Twitter

- Once you have identified the claim you want to use, can pull it from the claims and add it to your query

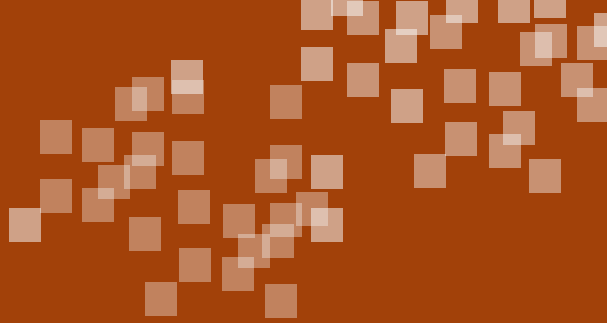
```
table.read(function (context) {
  return context.user.getIdentity().then(function (userInfo) {
     context.query.where({ userId: userInfo.twitter.claims.upn });
     return context.execute();
  });
});
```



Server-sid claims [ASP.NET]

- Extension method on **`IPrincipal`** retrieves all the **`Claim`**s for a specified provider (Facebook, Twitter, etc.)

```
[MobileAppController]
public class IdentityController : ApiController
{
   public async Task<string> GetTwitterName()
   {
      var user = this.User as ClaimsPrincipal;
      var claims = await this.User
                          .GetAppServiceIdentityAsync<TwitterCredentials>(Request);
      string twitterId = claims.UserClaims.Single(
                          c => c.ValueType == ClaimTypes.Upn).Value;
      return twitterId;
   }
}
```



Demonstration

Examine additional claims

Xamarin University

Summary

1. Identifying the user on the server
2. Associating data to a specific user
3. Getting additional claim information

Next Steps

- This class has provided a deep dive into the mobile application support in Azure
- Next we will look at adding push notifications to our mobile application that are driven from Azure

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile

Microsoft