Offline Synchronization with Azure

Offline synchronization

Azure supports **offline data synchronization** with just a few lines of code; this provides several tangible benefits

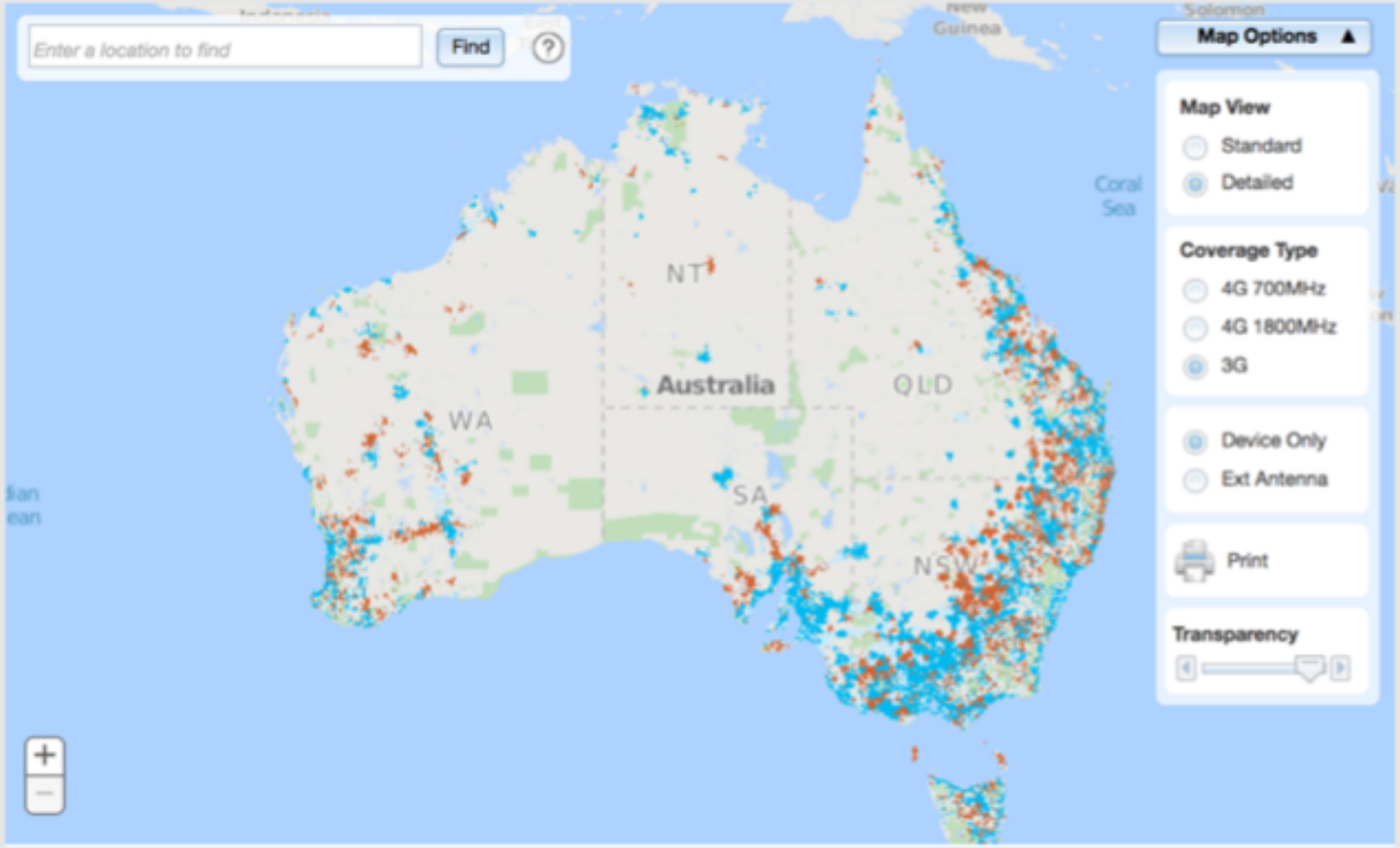
Improves app responsiveness

R/W access to data even *when network is unavailable*

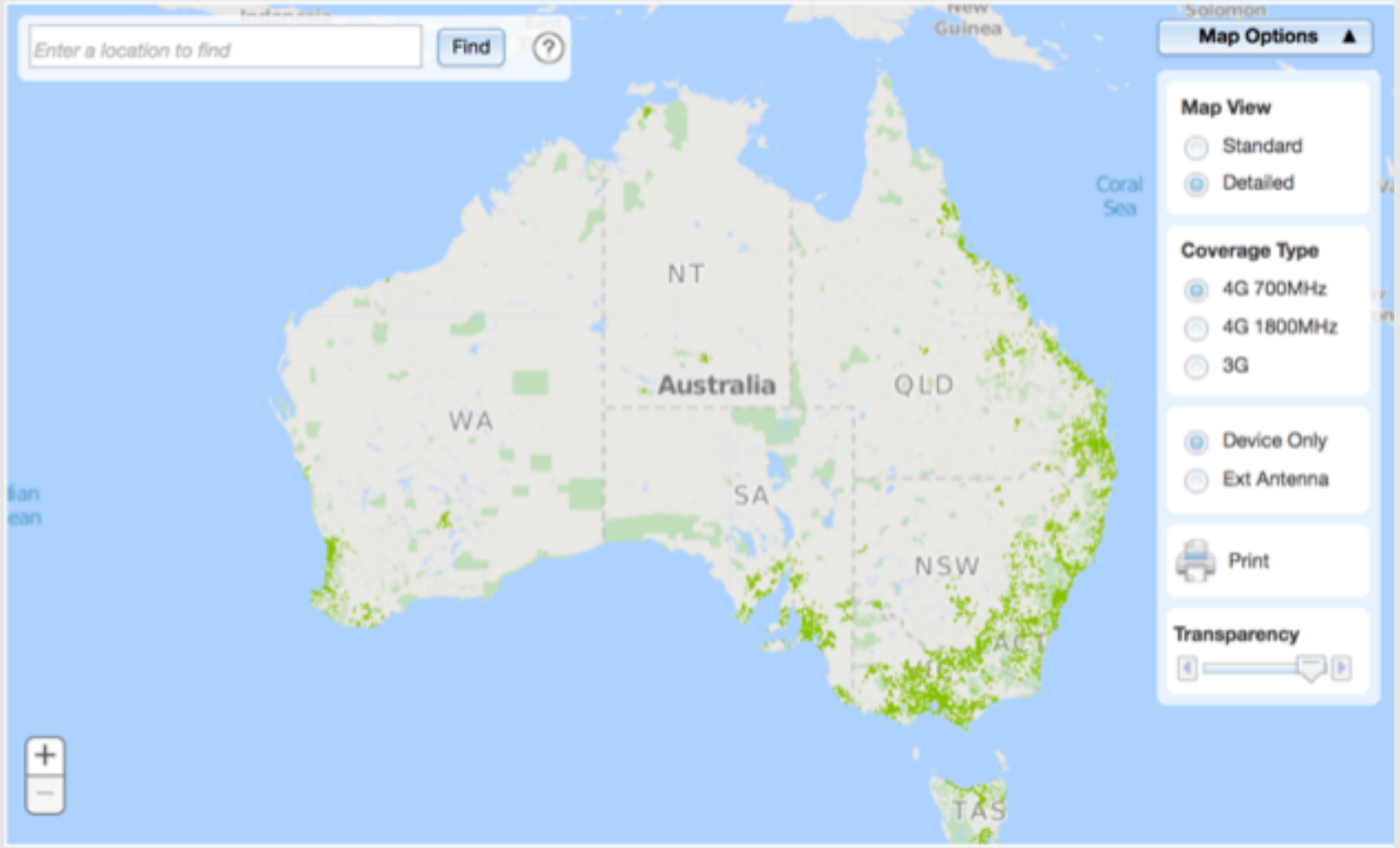
Automatic synchronization with local cache

Control when sync occurs for roaming

The reason for offline access

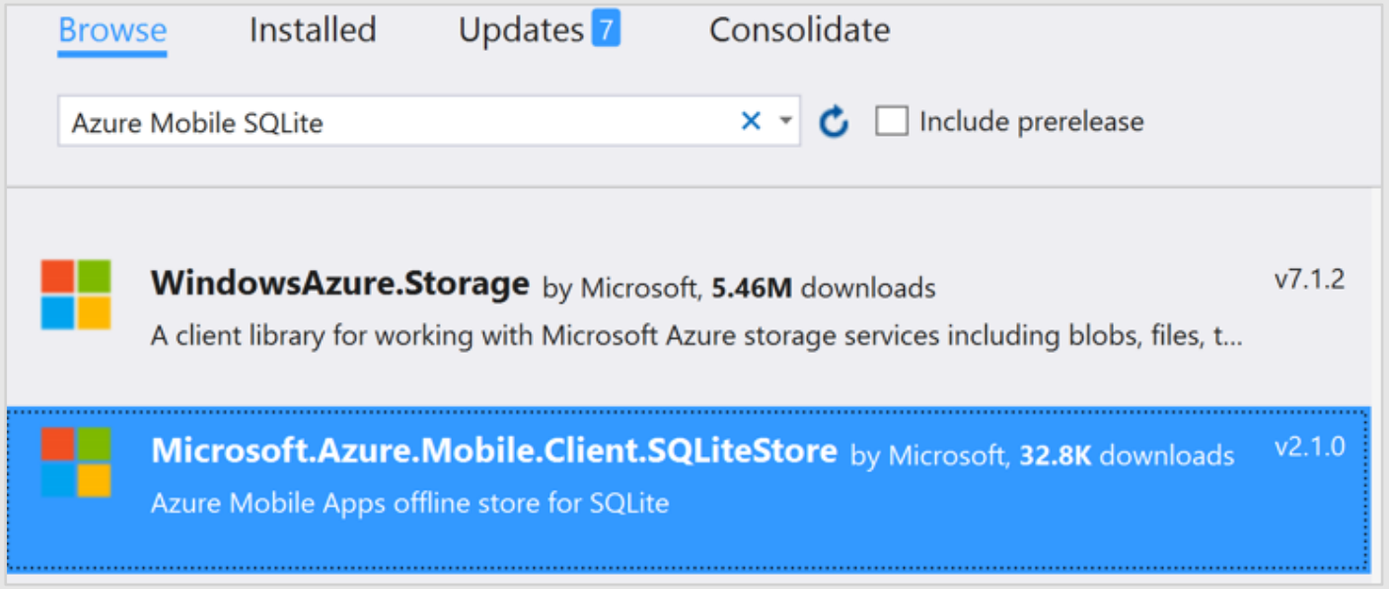


The reason for offline access



Adding support for offline sync

Add a NuGet reference to the **Azure SQLiteStore** package to support offline synchronization; this will also include SQLite support in your app



Associate the local cache

Must associate the SQLite store with the **`MobileServiceClient`** through the public **`SyncContext`** property

```
mobileService = new MobileServiceClient(AzureEndpoint);
...
var store = new MobileServiceSQLiteStore("localstore.db");
store.DefineTable<DiaryEntry>();
await mobileService.SyncContext.InitializeAsync(store,
                                    new MobileServiceSyncHandler());
```

Retrieve a sync table

Offline support is implemented by a new **`IMobileServiceSyncTable<T>`** interface; this is retrieved through the **`GetSyncTable<T>`** method

```
IMobileServiceSyncTable<DiaryEntry> diaryTable;
...
mobileService = new MobileServiceClient(AzureEndpoint);
...
diaryTable = mobileService.GetSyncTable<DiaryEntry>();
...
```

Query operators

All the same basic query operations are supported by **`IMobileServiceSyncTable`**

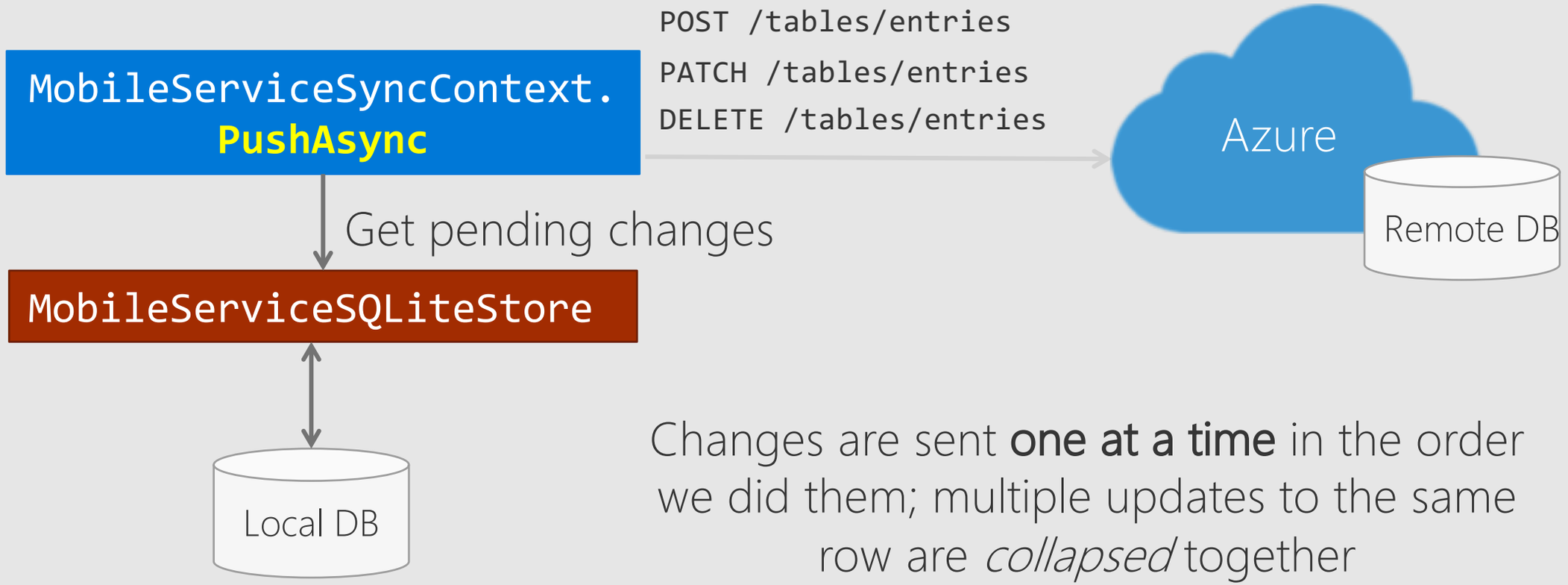
```
IMobileServiceSyncTable<DiaryEntry> diaryTable = ...;

var entry = new DiaryEntry { Text = "Some Entry" };
try {
   await diaryTable.InsertAsync(entry);
}
catch (Exception ex) {
   ... // Handle error
}
```

This works even if we aren't connected to the network!

Synchronize changes

To synchronize to the Azure remote database, your code must perform two operations; first we *push* all pending changes up to the remote DB



Changes are sent **one at a time** in the order we did them; multiple updates to the same row are *collapsed* together

Synchronizing the DB

```
private async Task SynchronizeAsync()
{
    if (!CrossConnectivity.Current.IsConnected)
        return;

    try
    {
        await MobileService.SyncContext.PushAsync();
        await diaryTable.PullAsync(null, diaryTable.CreateQuery());
    }
    catch (Exception ex)
    {
        // TODO: handle error
    }
}
```

Pull data from the server

Enable *incremental sync* by providing a client-side **query id**, or pass **`null`** to turn it off

```
await diaryTable.PullAsync("allEntries",
                diaryTable.CreateQuery());
```

OR

```
await diaryTable.PullAsync("privateEntries",
                diaryTable.Where(d => d.IsPrivate);
```

query id must be unique per-query; try to have **one query per table** to minimize storage and memory overhead in the client

Handling conflicts

Conflict handler code must walk through the set of returned errors and decide what to do for each record based on the application and data requirements

```
catch (MobileServicePushFailedException ex)
{
    if (ex.PushResult != null)
    {
        foreach (MobileServiceTableOperationError error
                    in exception.PushResult.Errors) {
            await ResolveConflictAsync(error);
        }
    }
}
```

Example: Take the client version

One possibility is to always assume the client copy is the one we want

```
async Task ResolveConflictAsync(MobileServiceTableOperationError error)
{
    var serverItem = error.Result.ToObject<Diary
    var localItem = error.Item.ToObject<DiaryEn
    if (serverItem.Text == localItem.Text) {
        // Items are the same, so ignore the conflict
        await error.CancelAndDiscardItemAsync();
    }
    else {
        // Always take the client; update the Version# and resubmit
        localItem.AzureVersion = serverItem.AzureVersion;
        await error.UpdateOperationAsync(JObject.FromObject(localItem));
    }
}
```

If the server and local row is the same then discard our change and ignore the conflict

Demo

Offline Sync

Authentication Support

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

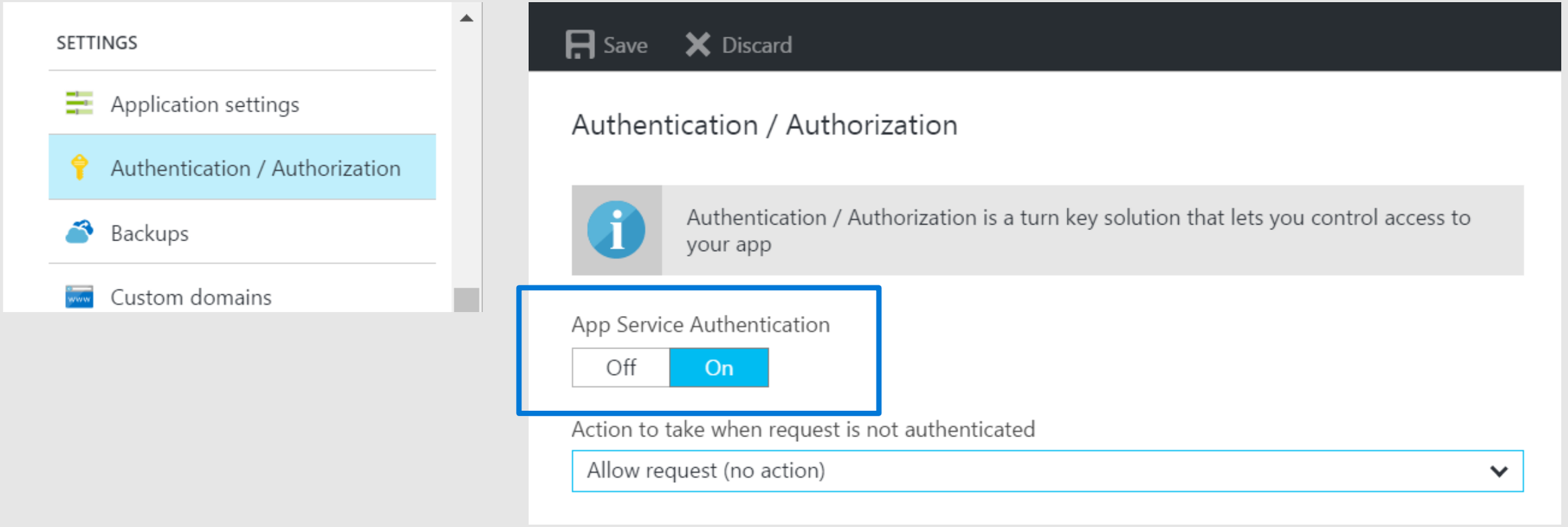
Authentication in Azure

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

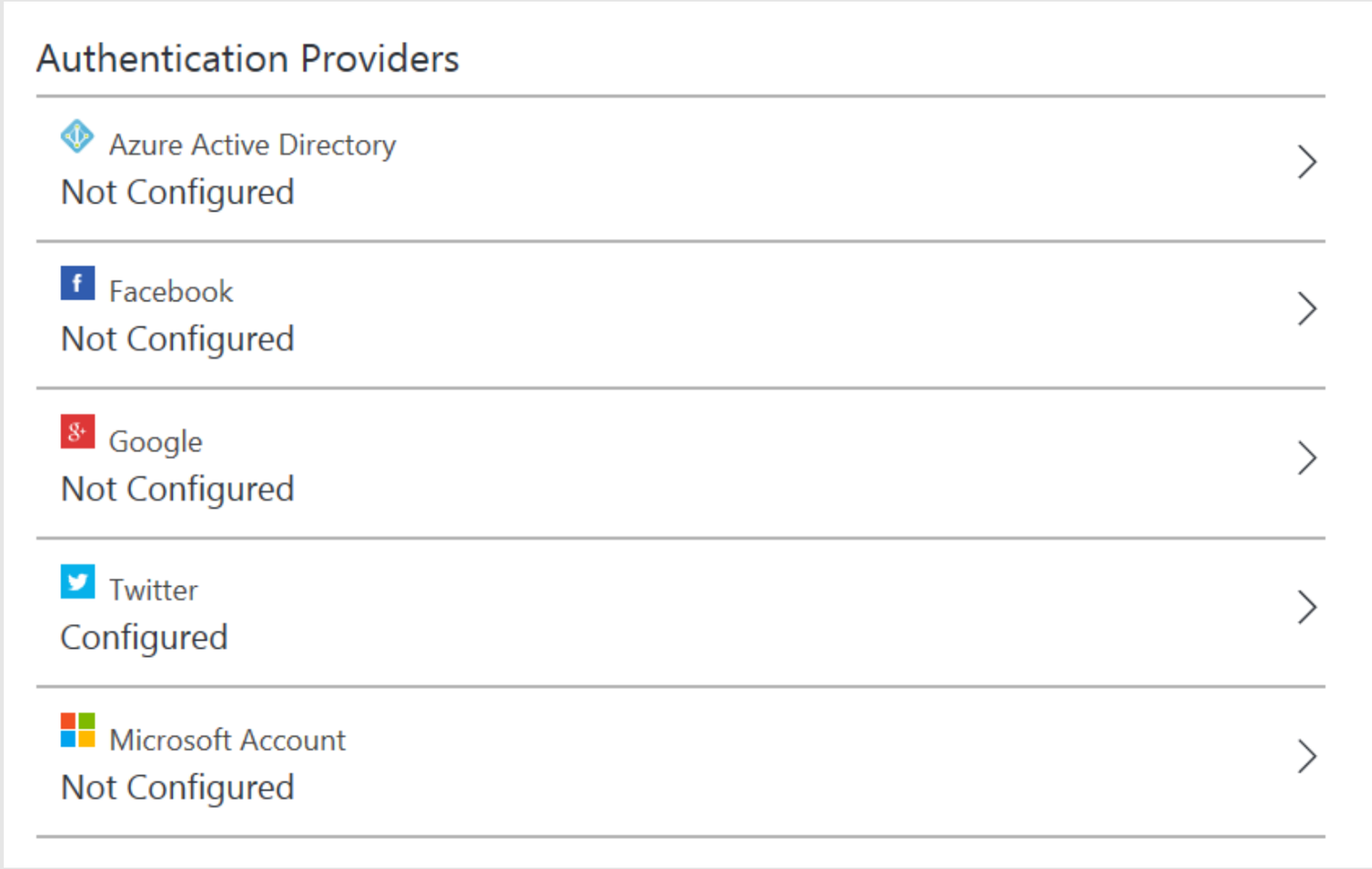
Step 1: turn on authentication

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



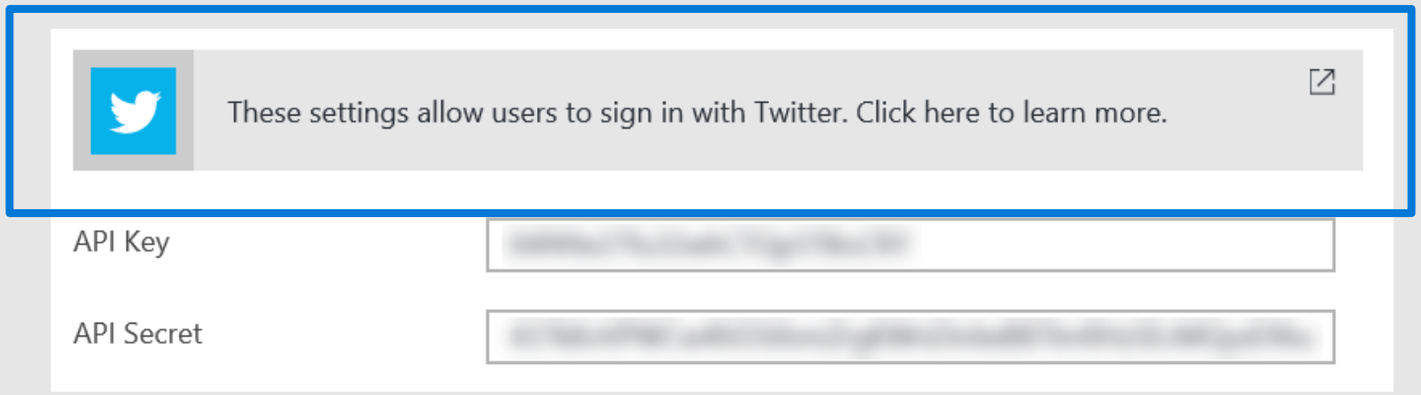
Step 2: decide which IdP(s) to use

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



Configuring your 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

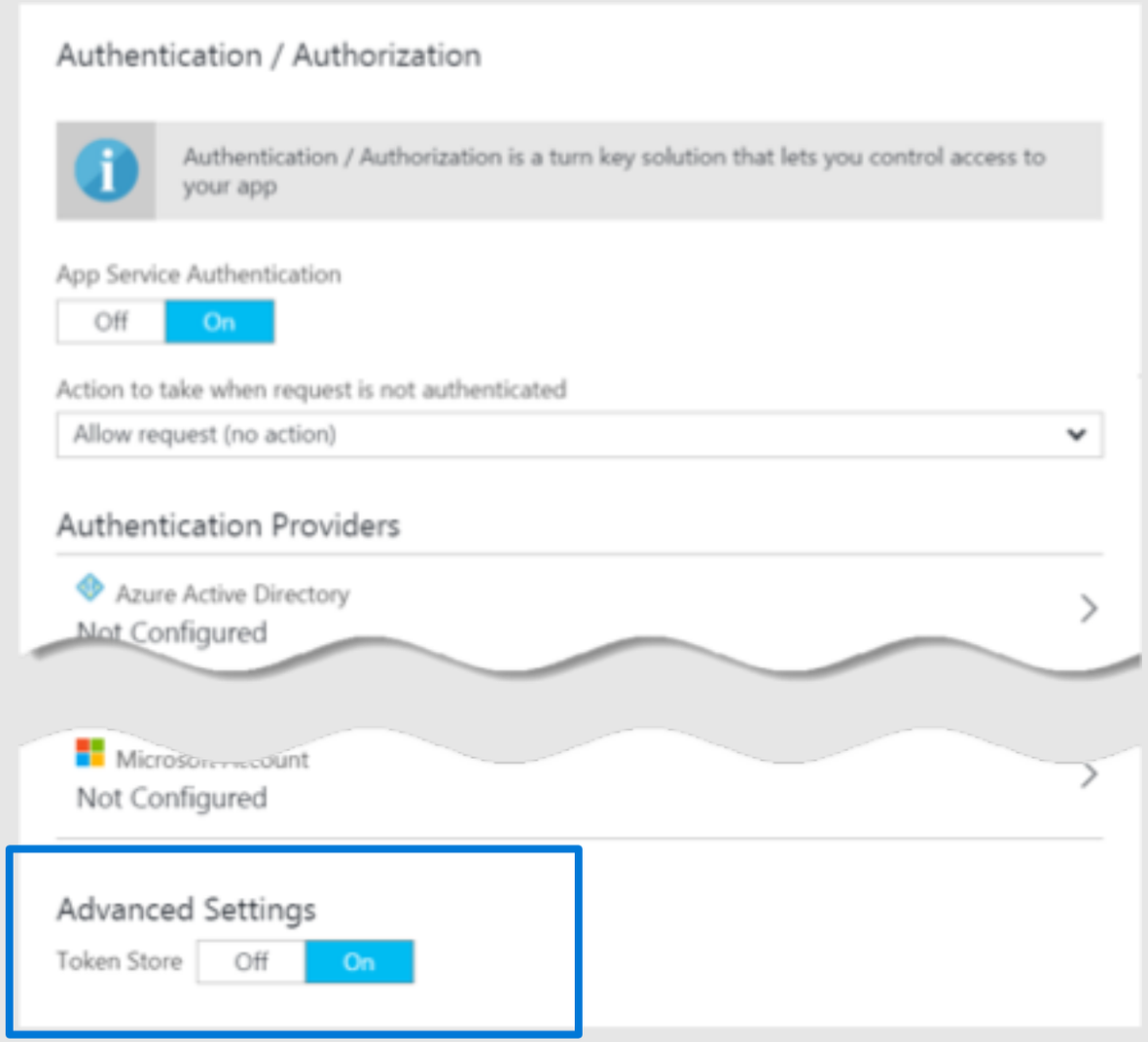
Azure exposes unique OAuth redirect (callback) URLs for each provider – this URL must start with **`https://`**

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`

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

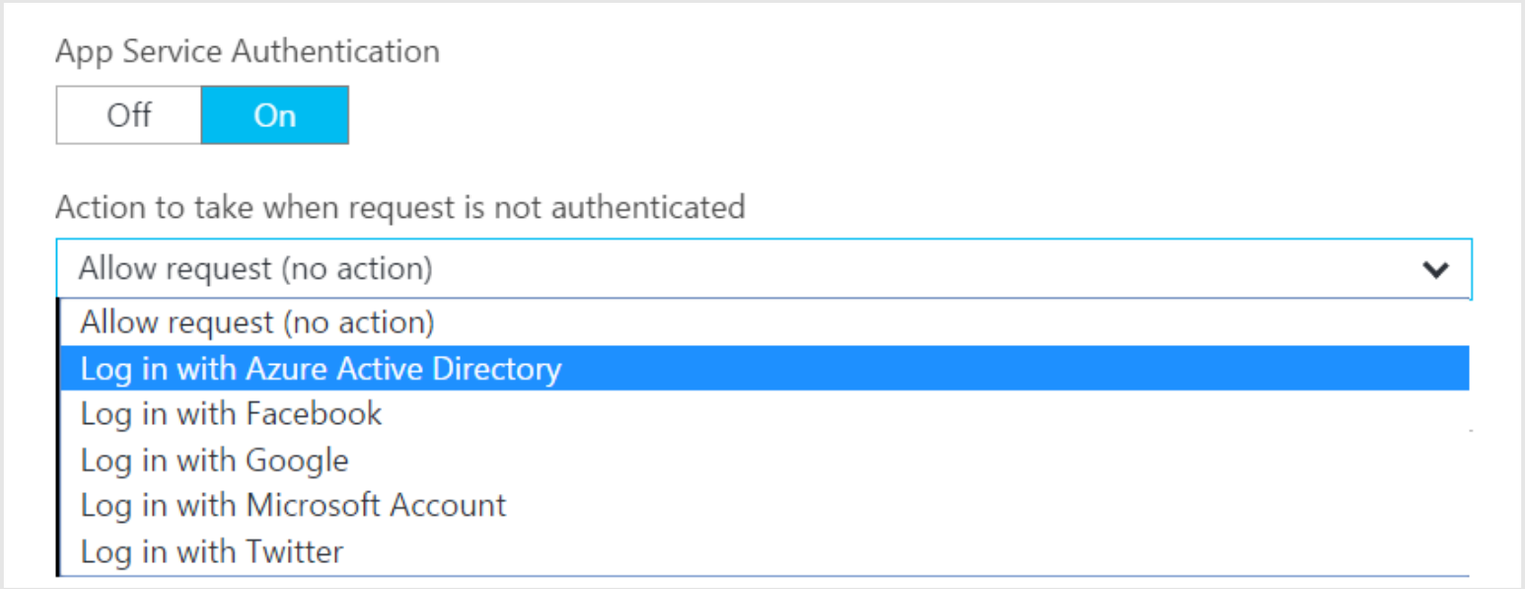
Can test Identity Provider setup through a browser or REST client by hitting a specific URL

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

Step 4: decide when to authenticate

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



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) ...
```

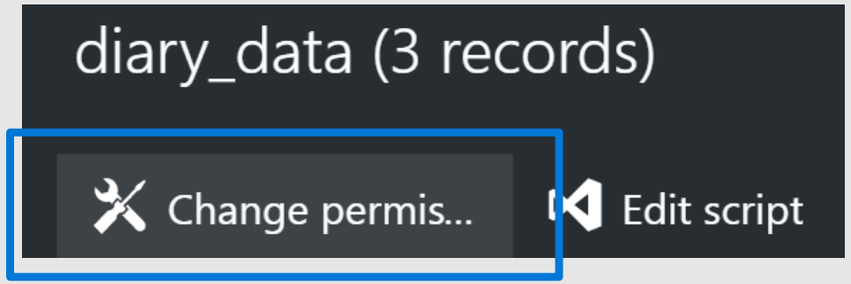
Authentication in ASP.NET service

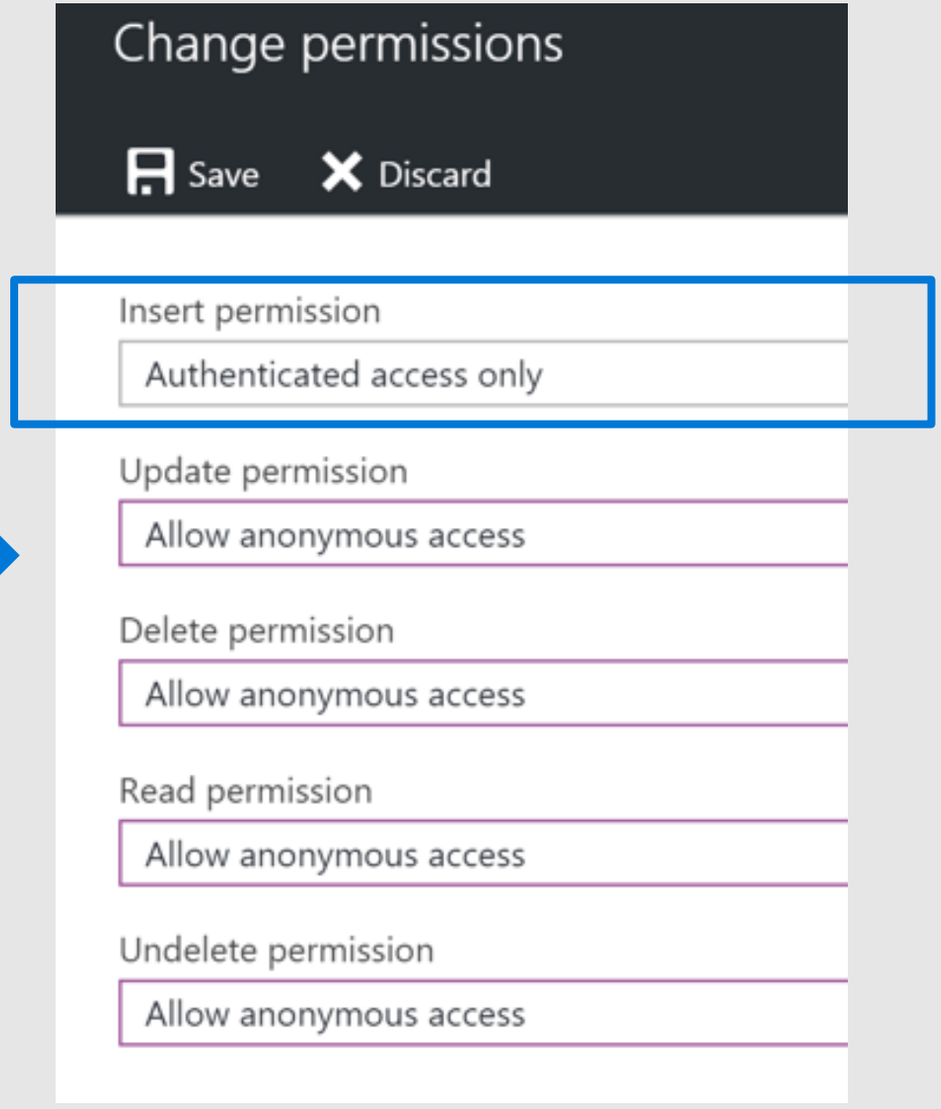
Can turn *off* authentication for an endpoint or action with the **`[AllowAnonymous]`** attribute

```
[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





Selecting an OAuth flow

Client can choose which OAuth flow it wants to utilize

Client flow

Server flow

Requesting authentication

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

```
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
```

iOS

```
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);
        }
    }
}
```

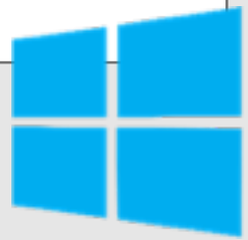
```
using System.Threading.Tasks;
using Microsoft.WindowsAzure.MobileServices;

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

iOS

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

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



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(...);
}
```

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

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

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();
```

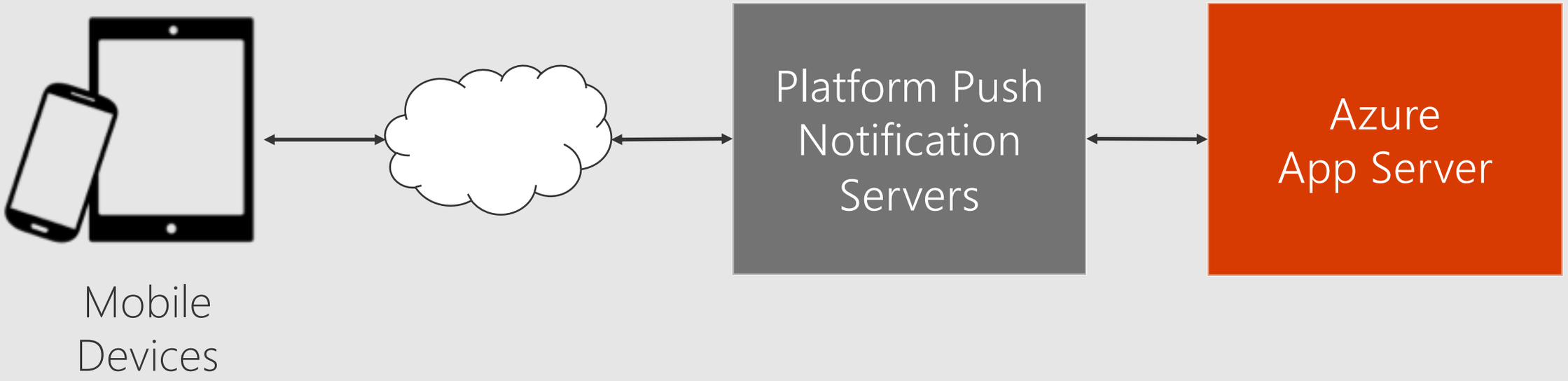
Exercise 20

Providing Authorization

Push Notifications

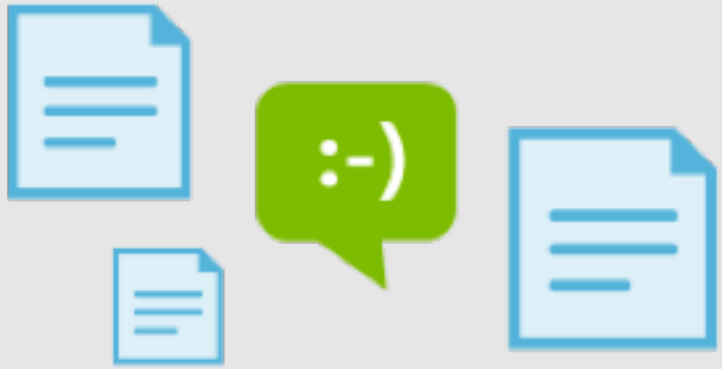
What are push notifications?

Push notifications are remote messages sent to mobile devices to inform them of some notifiable event.



Notification Hubs

Simplifies management of push notification systems from Azure

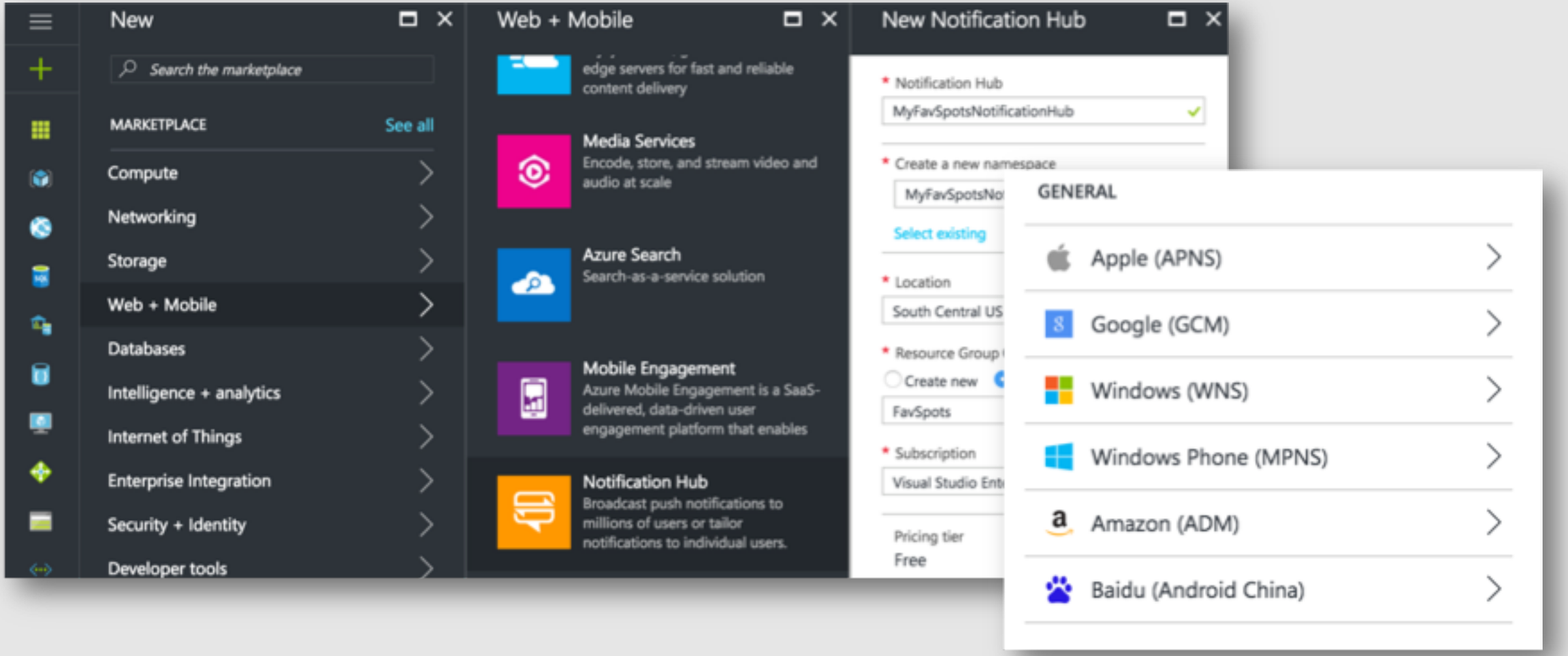


- Templates Support

Support for all popular devices (iOS/Android/Amazon/Windows)

Highly Scalable

Creating a Notification Hub



Configuring the Platforms

For each platform:

1. Create the project for each platform
2. Provide the platform details to Notification Hub blade
3. Test communication if the client devices are configured

Connecting the clients

Each client has its own process. You will need to configure the platforms. You will need to use the notification clients and connect them in.

1. Add the Notifications support NuGet library
2. Add Platform specific libraries for platforms that require them
3. Connect to the Notification Libraries and supply the appropriate templates
4. Process the notifications as they come in through the normal platform mechanisms

Sending from the Server

Test Capabilities allow you to easily test that the configuration is correct.

Supports test delivery for:

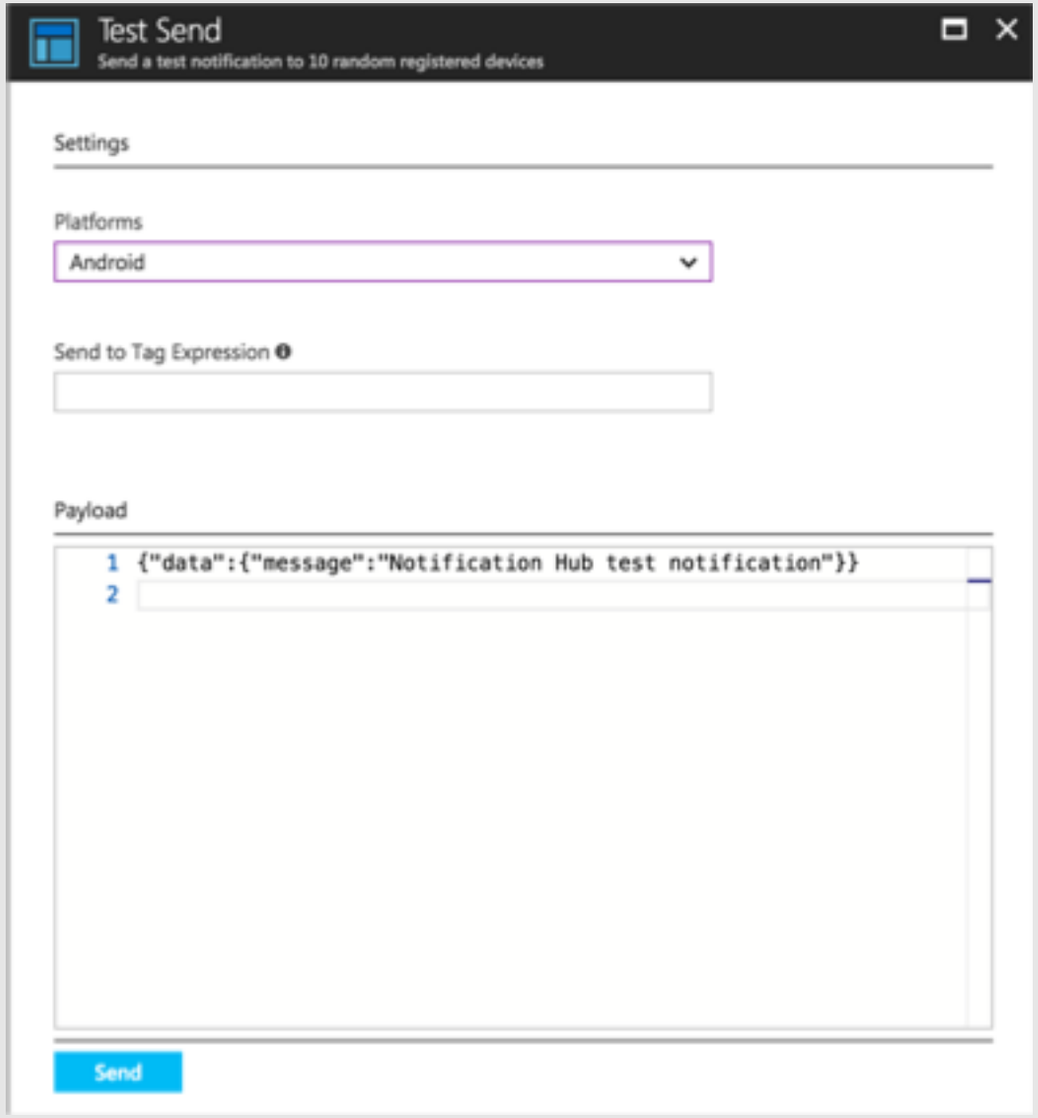
Apple

Android

Baidu

Windows Phone

Windows



Sending from the Server

From ASP.NET MVC project using the template

```
public class Notifications
{
    public static Notifications Instance = new Notifications();

    public NotificationHubClient Hub { get; set; }

    const string DefaultFullSharedAccessSignature = "...";
    const string HubName = "...";

    private Notifications()
    {
        Hub = NotificationHubClient.CreateClientFromConnectionString(
            DefaultFullSharedAccessSignature, HubName);
    }
}
```

Demo

Demonstration - Push Notifications

Summary

- Providing Offline Synchronization enables your app to be used when a network is not available

- Authentication can be performed against Social Media Identity providers and Azure Active Directory to provide a managed and trusted login experience