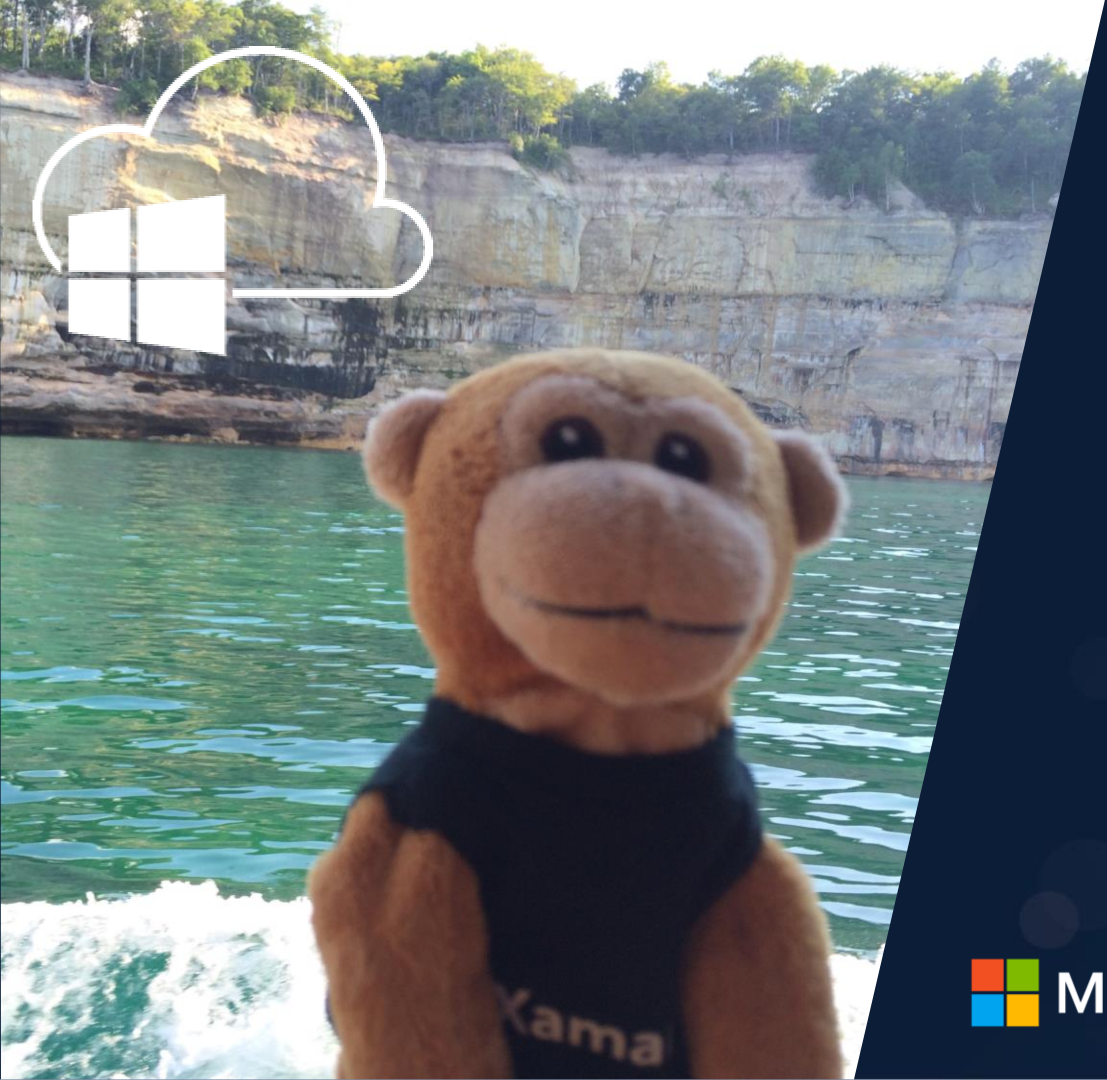


AZR115

Building an Azure Mobile App Client

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Connect your mobile app to Azure
2. Access table data from Azure
3. Add support for offline synchronization

Connect your mobile app to Azure

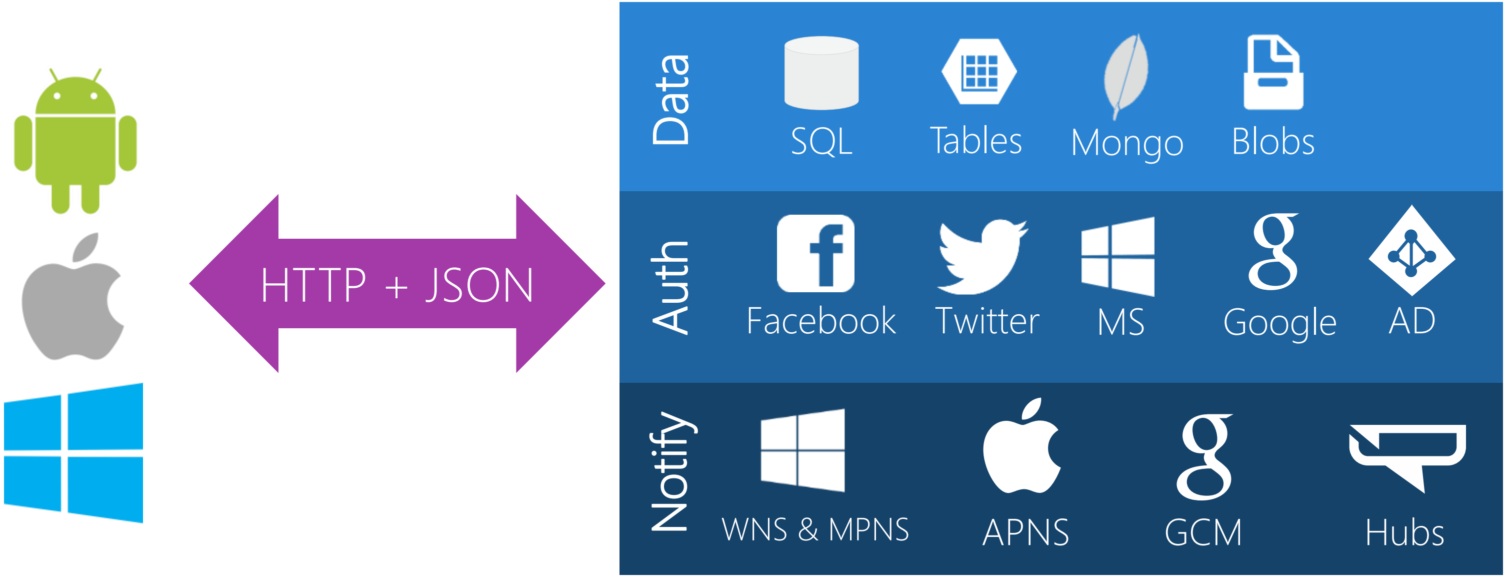
Tasks

1. Add the required NuGet packages
2. Connecting to Azure
3. Configuring the Azure client

Xamarin University

Azure App Services: Mobile App

- Mobile apps built on Azure App Services provide access to data, authentication and notifications using **standard web protocols**



Interacting with an Azure App Service

- Since the Mobile App uses standard web protocols (HTTP + JSON), .NET and Xamarin clients can use standard .NET classes such as **`HttpClient`** to access the service

```
const string AzureEndpoint = "...";

var client = new HttpClient();
client.DefaultRequestHeaders.Accept.Add(
           new MediaTypeWithQualityHeaderValue("application/json"));
...
string result = await client.GetStringAsync(AzureUrl);
... // Work with JSON string data
```

Required header value

- Must pass value **`ZUMO-API-VERSION`** on every request to indicate that the client is compatible with App Services vs. the older Mobile Services; can pass value as header or on the query string

```
var client = new HttpClient();
client.DefaultRequestHeaders.Accept.Add(
           new MediaTypeWithQualityHeaderValue("application/json"));
...
client.DefaultRequestHeaders.Add("ZUMO-API-VERSION", "2.0.0");
```

OR

```
string result = await client.GetStringAsync(AzureUrl +
                                  "?ZUMO-API-VERSION=2.0.0");
```

Parsing the response (JSON)

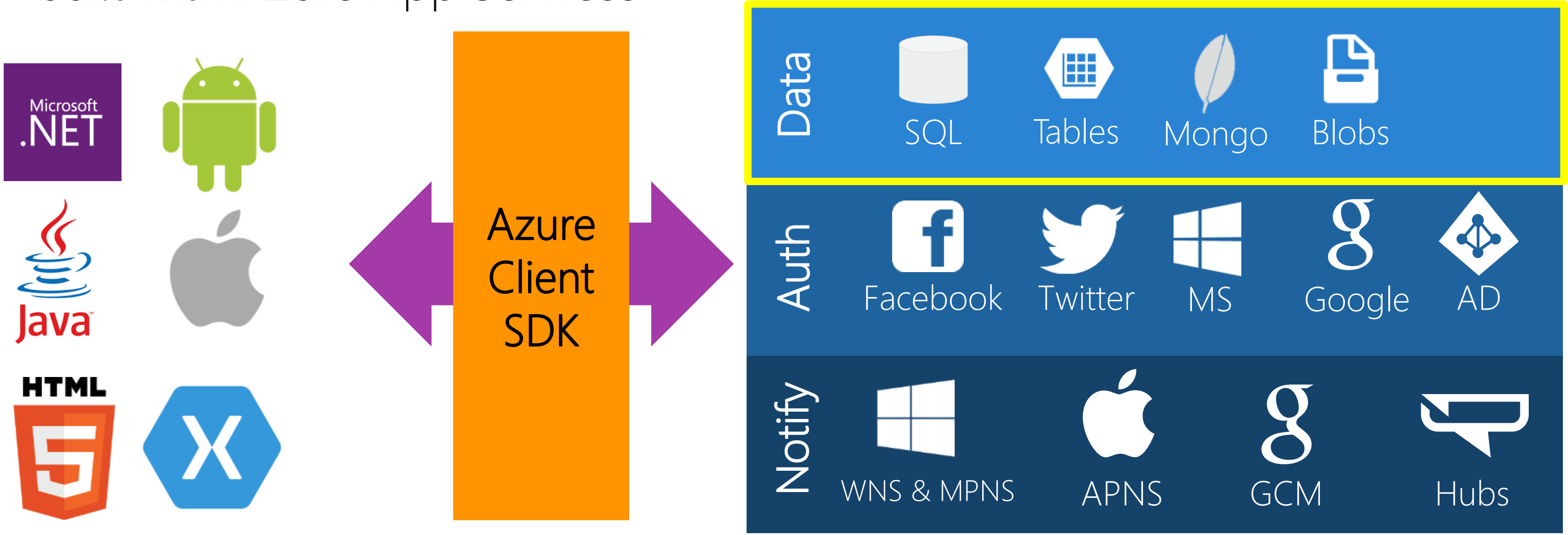
- Data is communicated using JSON – can use **standard parsers** to serialize and de-serialize information

```
string result = await client.GetStringAsync(DataEndpoint);
dynamic dataList = Newtonsoft.Json.JsonConvert
                             .DeserializeObject(result);
foreach (dynamic item in dataList) {
   Console.WriteLine("{0}", item.id);
}
```

Can parse JSON data as dynamic runtime values, JSON object must have an **id** value or this will throw a *runtime* exception

Standardized access

- Can utilize the pre-built Azure Client SDK from .NET or Xamarin to manage the HTTP/REST communication and interact with a Mobile App built with Azure App Services



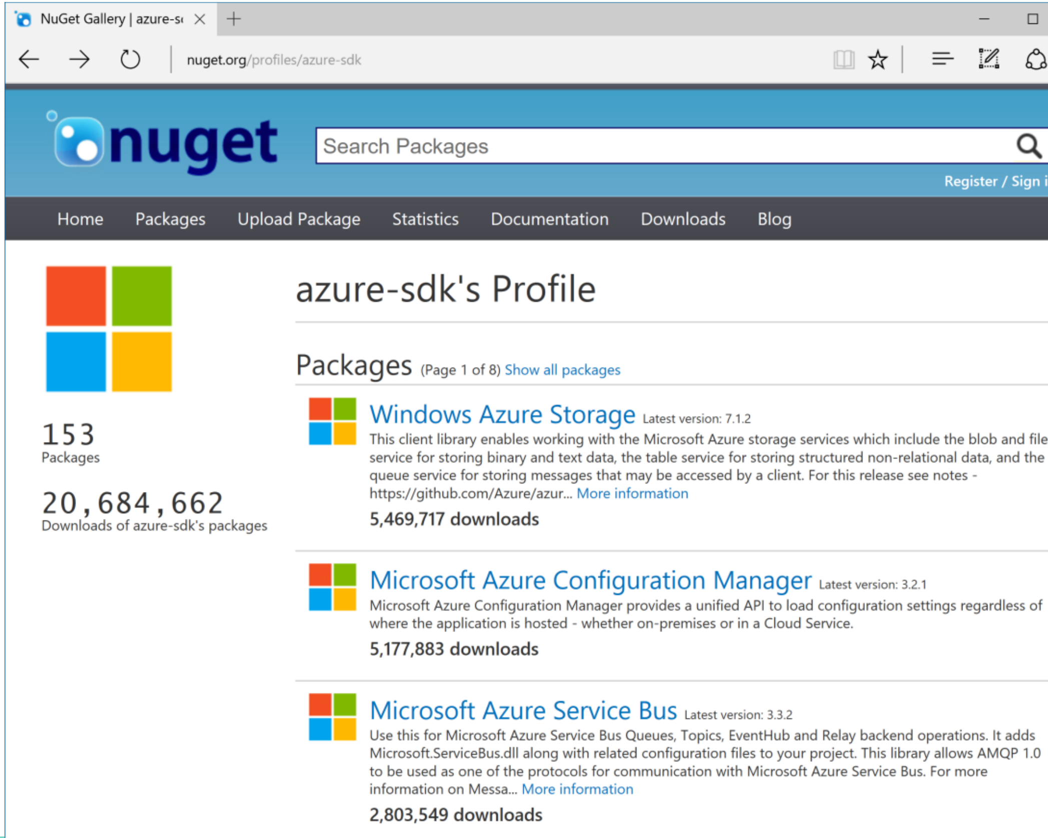
How to add the Azure client SDK

1. Add the required NuGet packages to your projects
2. Initialize the Azure client SDK in your platform projects
3. Access the mobile service using a configured `MobileServiceClient` object

NuGet packages

- .NET and Xamarin applications can use pre-built client access libraries available from NuGet to access various Azure services

- Azure SDKs are also published as open source https://github.com/Azure/



azure-sdk's Profile

153
Packages

20,684,662
Downloads of azure-sdk's packages

Packages (Page 1 of 8) Show all packages

Windows Azure Storage Latest version: 7.1.2
This client library enables working with the Microsoft Azure storage services which include the blob and file service for storing binary and text data, the table service for storing structured non-relational data, and the queue service for storing messages that may be accessed by a client. For this release see notes - https://github.com/Azure/azur... More information
5,469,717 downloads

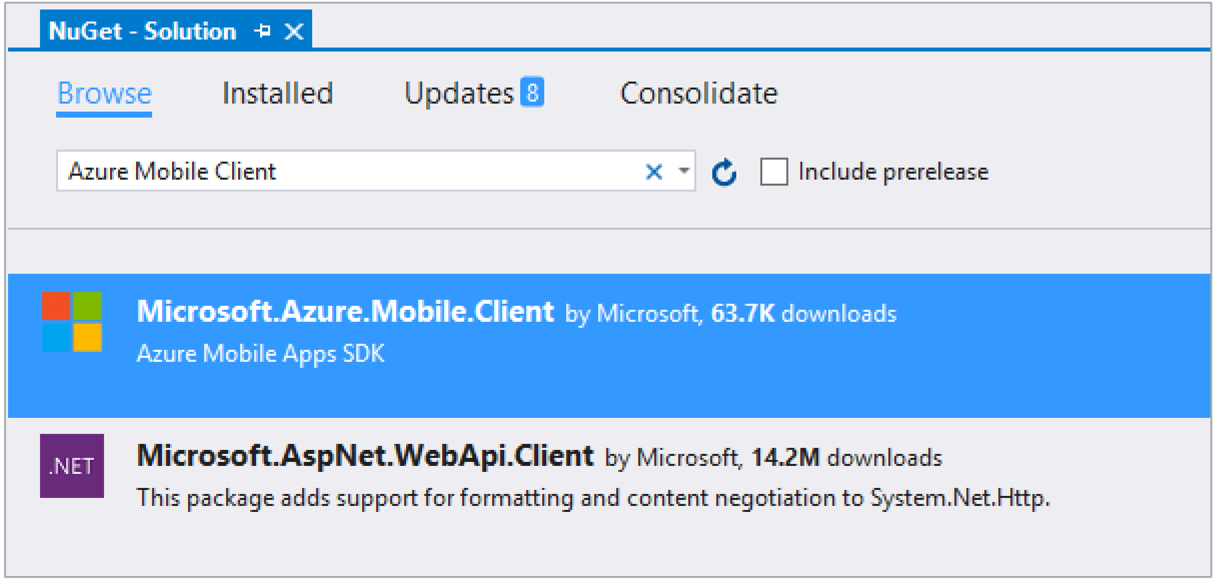
Microsoft Azure Configuration Manager Latest version: 3.2.1
Microsoft Azure Configuration Manager provides a unified API to load configuration settings regardless of where the application is hosted - whether on-premises or in a Cloud Service.
5,177,883 downloads

Microsoft Azure Service Bus Latest version: 3.3.2
Use this for Microsoft Azure Service Bus Queues, Topics, EventHub and Relay backend operations. It adds Microsoft.ServiceBus.dll along with related configuration files to your project. This library allows AMQP 1.0 to be used as one of the protocols for communication with Microsoft Azure Service Bus. For more information on Messa... More information
2,803,549 downloads

Xamarin University

Adding support for an Azure mobile app

- To add client-side support for an Azure mobile site, add a NuGet reference to the **Microsoft.Azure.Mobile.Client** package; this must be added to all the head projects *and* to any PCL using Azure classes



NuGet - Solution

Azure Mobile Client

□ Include prerelease

Microsoft.Azure.Mobile.Client by Microsoft, 63.7K downloads
Azure Mobile Apps SDK

Microsoft.AspNet.WebApi.Client by Microsoft, 14.2M downloads
This package adds support for formatting and content negotiation to System.Net.Http.

This also adds references to a few other packages such as Json.NET

Required initialization code [Android]

- iOS and Android require some initialization for the Azure client SDK, typically done as part of the app startup

```
protected override void OnCreate (Bundle bundle)
{
   base.OnCreate (bundle);
   Microsoft.WindowsAzure.MobileServices.CurrentPlatform.Init();
   ...
}
```

Can place the Android initialization wherever it makes sense – commonly done either in the global App, or as part of the main **Activity** creation

Required initialization code [iOS]

- iOS and Android require some initialization for the Azure client SDK, typically done as part of the app startup

```
public override bool FinishedLaunching(UIApplication app,
                                       NSDictionary options)
{
   Microsoft.WindowsAzure.MobileServices.CurrentPlatform.Init();
   ...
   return true;
}
```

iOS initialization is commonly placed into the App Delegate **`FinishedLaunching`** method

This code is not necessary for Windows or UWP applications

Connecting to Azure

- `MobileServiceClient` class provides the core access to Azure services; should create and cache this object off in your application

```
const string AzureEndpoint = "https://<site>.azurewebsites.net";
MobileServiceClient mobileService;
...

mobileService = new MobileServiceClient(AzureEndpoint);
```

Constructor identifies the specific Azure service to connect to

Individual Exercise

Add Azure support to our Xamarin application

Xamarin University

Summary

1. Add the required NuGet packages
2. Connecting to Azure
3. Configuring the Azure client

Access table data from Azure

Xamarin University

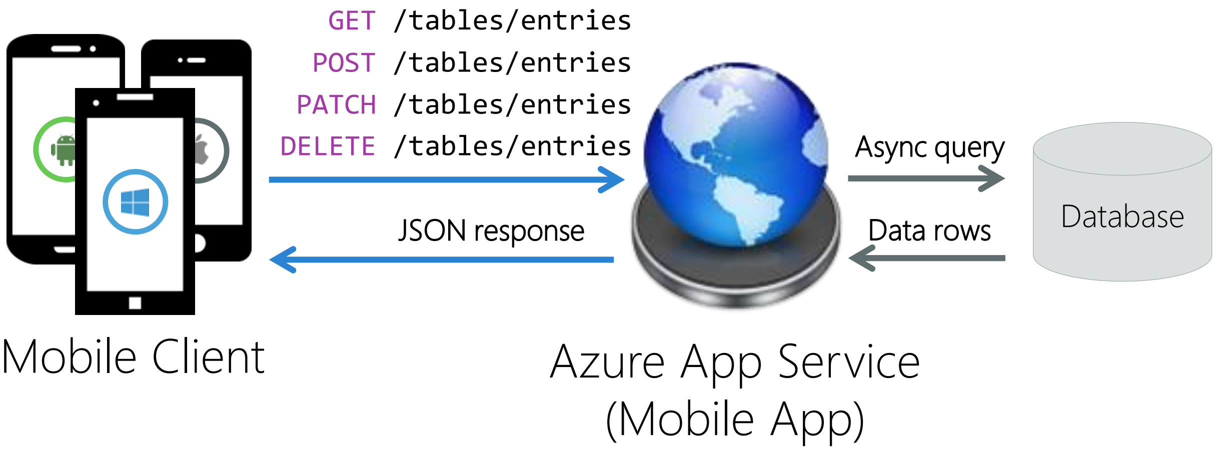
Tasks

1. Accessing an Azure DB table
2. Define the data transfer object
3. Adding a new record to the DB
4. Performing queries

Xamarin University

Accessing tables from a client

- Azure App mobile service exposes endpoints (`/tables/{tablename}`) to allow applications to perform DB queries and operations using HTTP



Accessing a table

- **`MobileServiceClient`** exposes each server-side table as a **`IMobileServiceTable`** which can be retrieved with **`GetTable`**

```
service = new MobileServiceClient("https://{site}.azurewebsites.net");
...
IMobileServiceTable table = service.GetTable("{tablename}");

var dataList = await table.ReadAsync(string.Empty);
foreach (dynamic item in dataList) {
   string id = item.id;
   ...
}
```

Same un-typed access available – under the covers this is a **`JObject`** from Json.NET

Standard table data

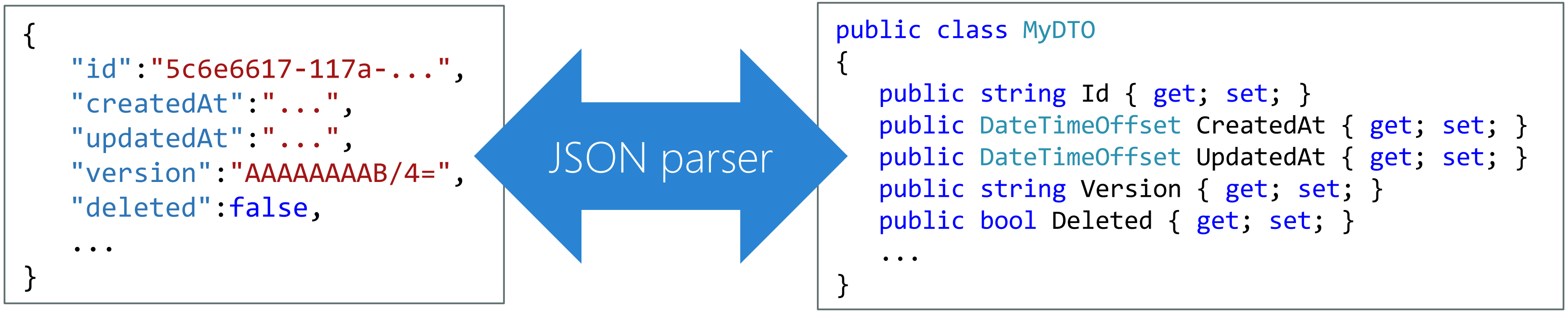
❖ Tables defined by a Mobile App always have 5 **pre-defined columns** which are passed down from the service in JSON

```
{
   "id":"5c6e6617-117a-4118-b574-487e55875324",
   "createdAt":"2016-08-10T19:14:56.733Z",
   "updatedAt":"2016-08-10T19:14:55.978Z",
   "version":"AAAAAAAAB/4=",
   "deleted":false
}
```

These fields are all **system provided values** which should not be changed by the client unless the server code is specifically written to allow it

Using strongly typed data

- Can use a parser to convert JSON table data into a **strongly typed** .NET object, referred to as a *data transfer object* (DTO)



```
{
   "id":"5c6e6617-117a-...",
   "createdAt":"...",
   "updatedAt":"...",
   "version":"AAAAAAAAB/4=",
   "deleted":false,
   ...
}
```

```
public class MyDTO
{
   public string Id { get; set; }
   public DateTimeOffset CreatedAt { get; set; }
   public DateTimeOffset UpdatedAt { get; set; }
   public string Version { get; set; }
   public bool Deleted { get; set; }
   ...
}
```

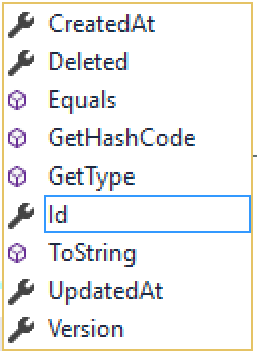
DTO must define **public properties** to hold the data represented in JSON

Using a DTO

- **MobileServiceClient** supports DTOs through generic **GetTable<T>** method which returns a **IMobileServiceTable<T>**

```
IMobileServiceTable<DiaryEntry> table = service.GetTable<DiaryEntry>();

IEnumerable<DiaryEntry> entries = await table.ReadAsync();
foreach (DiaryEntry item in entries) {
   string id = item.
   ...
}
```



Now we get Intellisense for the DTO

Required fields in your DTO

- `Id` property is **required** and must be present; this is used as the primary key for all DB operations and to manage offline synchronization

```
public class DiaryEntry
{
    public string Id { get; set; }
    ...



}
```

Should consider this a read-only property, but still must have a public setter in the DTO for JSON parser to use

Filling in property values

- Parser will use reflection match case-insensitive property names in the DTO to the JSON data

```
public class DiaryEntry
{
    public string Id { get; set; }
    public string Text { get; set; }
    ...



}
```

```
{
    "id":"5c6e6617-117a-...",
    "createdAt":"...",
    "updatedAt":"...",
    "version":"AAAAAAAAB/4=",
    "deleted":false,
    "text":"Hello, World"
}
```

Customizing the JSON shape

- Can decorate DTO with **`JsonPropertyAttribute`** to customize the JSON value the parser will use

```
public class DiaryEntry
{
    public string Id { get; set; }
    [JsonProperty("text")]
    public string Entry { get; set; }
    ...



}
```

```
{
    "id":"5c6e6617-117a-...",
    "createdAt":"...",
    "updatedAt":"...",
    "version":"AAAAAAAAB/4=",
    "deleted":false,
    "text":"Hello, Diary"
}
```

Can also use the **`DataMember`** attribute from the data contract serialization framework

Working with system properties

- Framework includes attributes which apply the correct name for most of the system-supplied values so you don't have to know the names

```
public class DiaryEntry
{
    public string Id { get; set; }
    [Version]
    public string AzureVersion { get; set; }
    [CreatedAt]
    public DateTimeOffset CreatedOn { get
    [UpdatedAt]
    public DateTimeOffset Updated { get;
}
```

```
{
    "id":"5c6e6617-117a-...",
    "createdAt":"...",
    "updatedAt":"...",
    "version":"AAAAAAAAB/4=",
    "deleted":false,
    "text":"Hello, Diary"
}
```

Ignoring DTO properties

- Tell parser to ignore DTO properties using the **`JsonIgnoreAttribute`**; this is particularly important for serialization (DTO > JSON)

```
public class DiaryEntry
{
   public string Id { get; set; }
   [JsonProperty("text")]
   public string Entry { get; set; }
   [JsonIgnore]
   public string Title { ... }
   ...

}
```

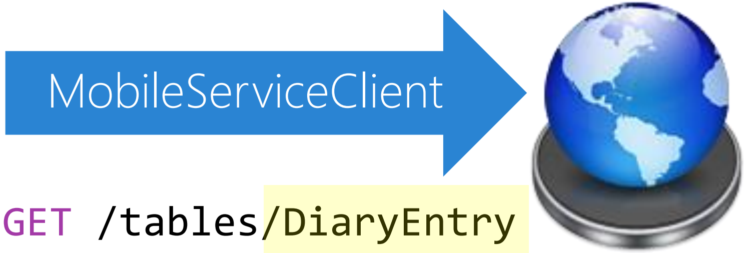
```
{
    "id":"5c6e6617-117a-...",
    "createdAt":"...",
    "updatedAt":"...",
    "version":"AAAAAAAAB/4=",
    "deleted":false,
    "text":"Hello, Diary"
}
```

Identifying the server side table

- Table endpoint is identified using the DTO name supplied to **`GetTable<T>`**

```
var table = service.GetTable<DiaryEntry>();
```

```
public class DiaryEntry
{
    ...
}
```



```
GET /tables/DiaryEntry
```

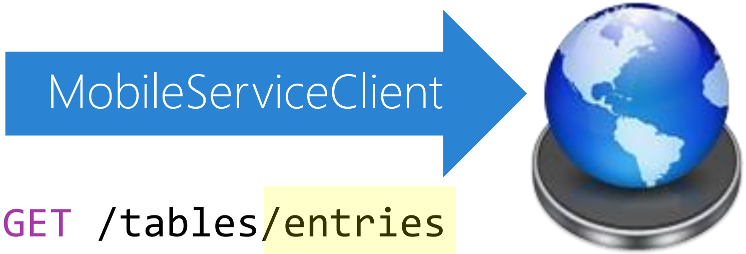
What if the server endpoint is **entries**?
Result is a **404** (Not Found) error!

Identifying the server side table

- Customize the endpoint with **`JsonObject`** or **`DataContract`** attribute

```
var table = service.GetTable<DiaryEntry>();
```

```
[JsonObject(Title = "entries")]
public class DiaryEntry
{
    ...
}
```



```
GET /tables/entries
```

Customizing the JSON serialization

- Can provide global **custom serialization settings** that apply to the JSON serializer to simplify your data entity definition

```
mobileService = new MobileServiceClient(AzureEndpoint) {
  SerializerSettings = new MobileServiceJsonSerializerSettings {
      CamelCasePropertyNames = true,
      DateFormatHandling = DateFormatHandling.IsoDateFormat,
      MissingMemberHandling = MissingMemberHandling.Ignore
  }
};
```

Individual Exercise

Customize the DTOs for the Survey service

REST operations

- `IMobileServiceTable` performs standard HTTP verbs to implement CRUD operations – Azure back-end then performs specific DB operation

Method	HTTP request	SQL Operation
`InsertAsync`	`POST /tables/{table}`	`INSERT`
`UpdateAsync`	`PATCH /tables/{table}`	`UPDATE`
`DeleteAsync`	`DELETE /tables/{table}`	`DELETE`
`ReadAsync`	`GET /tables/{table}`	`SELECT *`
`LookupAsync`	`GET /tables/{table}/{id}`	`SELECT {id}`

Adding a new record

- **InsertAsync** adds a new record to the table; it fills in the system fields in your client-side object from the server-generated columns

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

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

Async operation finishes when the REST API has added the record to the DB

Deleting and Updating data

- **`UpdateAsync`** and **`DeleteAsync`** are similar – they issue REST calls to the service identifying an existing entity record and return once the operation is complete on the server

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

try {
   await diaryTable.DeleteAsync(someEntry);
}
catch (Exception ex) {
   ... // Handle error
}
```

Retrieving data

- Mobile service table has a plethora of APIs to perform queries – the simplest ones return all records or a single record based on the `Id`

Retrieve all records

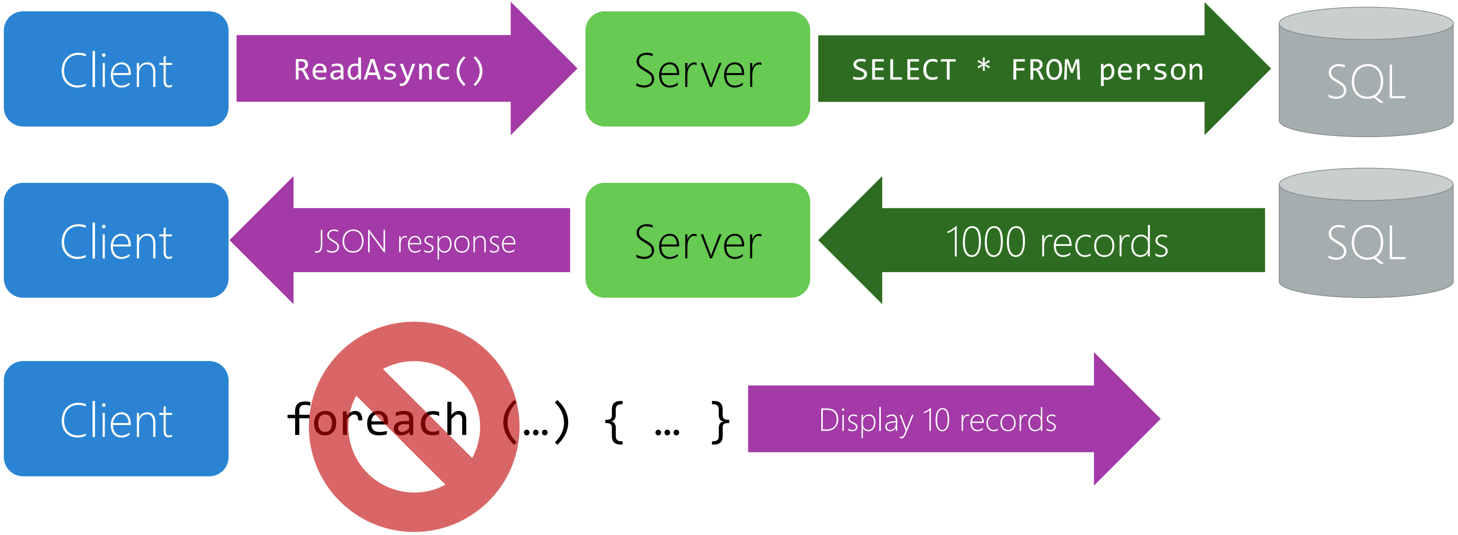
```
IEnumerable<DiaryEntry> allEntries = await diaryTable.ReadAsync();
```

Retrieve a single record by the unique identifier (id)

```
DiaryEntry entry = await diaryTable.LookupAsync(recordId);
```

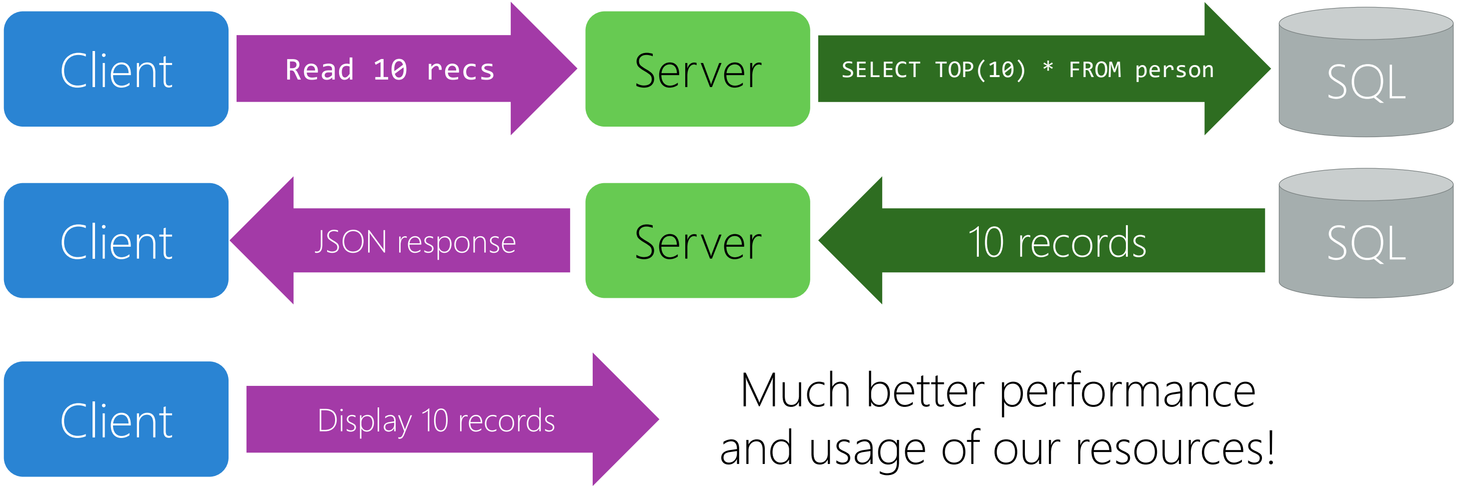
Filtering queries

- Remember this is a client/server model: pulling down all records and filtering on the client is inefficient – that's what the DB is good for!



Filtering queries

- Instead, we'd prefer to push the filtering up to the database and have it return only the records we are interested in



Much better performance and usage of our resources!

Performing queries

- Service supports basic filtering to be performed server-side; this is modeled on the client side as a *fluent LINQ API* exposed by the `IMobileServiceTableQuery` interface

```
IMobileServiceTableQuery<U> CreateQuery(...);
IMobileServiceTableQuery<U> Select<U>(...);
IMobileServiceTableQuery<T> Where(...);
IMobileServiceTableQuery<T> OrderBy<TKey>(...);
IMobileServiceTableQuery<T> OrderByDescending<TKey>(...);
IMobileServiceTableQuery<T> ThenBy<TKey>(...);
IMobileServiceTableQuery<T> ThenByDescending<TKey>(...);
IMobileServiceTableQuery<T> Skip(int count);
IMobileServiceTableQuery<T> Take(int count);
```

Make sure to execute query

- `IMobileServiceTableQuery` does not send request to server until you execute the query through a **collection method**

Method	What it does
`ToEnumerableAsync`	Returns an `IEnumerable<T>` (same as `ReadAsync`)
`ToListAsync`	Returns a `List<T>` with all retrieved data
`ToCollectionAsync`	Returns a collection with the data, supports an optional "page size" to retrieve data in chunks
`ToIncrementalLoadingCollection`	Returns a collection that pulls down data as it is accessed. Windows only

Filtering your queries

- Can use the **`Where`** method to add a **filter clause** to your query – this is evaluated on the server-side and reduces the amount of data transmitted back to the client

```
var onlySecretEntries = await diaryTable
      .Where(e => e.Text.ToLower().Contains("secret"))
      .ToEnumerableAsync();
```

Remember to call one of the collection methods to execute the request on the server – until you do this, it's just a query

```
GET /tables/diary_data?$filter=substringof('secret',tolower(Text))
```

Projecting your queries

- Can use **`Select`** to create projections of the query, the returned data will be restricted to the specified elements; any specified transformations are then performed on the retrieved data by the client

```
var JustTheFactsMaam = await diaryTable
        .Where(e => e.Text.Length > 0)
        .Select(e => e.Text.ToUpper())
        .ToListAsync();
```

Notice that the upper case request is not expressed in the OData request – that action isn't supported by the query language and is done on the client

```
GET /tables/diary_data?$filter=(length(Text)%20gt%200)&$select=Text
```

Stringing along queries

- API is fluent and allows you to string different expression options together to form a single query which can then be passed to the server

```
var all = diaryTable.CreateQuery();
var skip5 = all.Where(e => e.Text.Length > 0)
               .Skip(5)
               .OrderBy(e => e.UpdatedAt)
               .ThenBy(e => e.Text)
var firstTwo = skip5.Take(2);
var data = await firstTwo.ToCollectionAsync();
```

```
GET /tables/diary_data?$filter=(length(Text)%20gt%200)&
$orderby=updatedAt,Text&$skip=5&$top=2
```

LINQ

- Can use language integrated query (LINQ) to construct queries – compiler will then call all the methods

```
var JustTheFactsMaam = await diaryTable
       .Where(e => e.Text.Length > 0)
       .Select(e => e.Text.ToUpper())
       .ToListAsync();
```

```
var JustTheFactsMaam = await
      (from e in diaryTable
       where e.Text.Length > 0
       select e.Text.ToUpper()).ToListAsync();
```

Individual Exercise

Fill in the logic to query and update our survey records

Dealing with DELETE

- DELETE is a **destructive operation** which must be propagated to every client; tables can be configured to use a *soft delete* model where a **column in the database** is used to indicate that the record has been deleted

```
{
    "id":"5c6e6617-117a-4118-b574-487e55875324",
    "createdAt":"2016-08-10T19:14:56.733Z",
    "updatedAt":"2016-08-10T19:14:55.978Z",
    "version":"AAAAAAAAB/4=",
    "deleted":false
}
```

Reading deleted records

- Can retrieve deleted records by using the **`IncludeDeleted`** fluent method – this can be added to any query

```csharp
public Task<IEnumerable<DiaryEntry>> GetAll(bool includeDeleted = false)
{
    return (includeDeleted)
        ? diaryTable.IncludeDeleted().ToEnumerableAsync()
        : diaryTable.ToEnumerableAsync();
}
```

Undeleting records

- Can undelete a record when soft deletes are enabled; this will change the **deleted** flag to **false** on the record

```
public async Task RestoreAllRecordsAsync()
{
   var allItems = await diaryTable.IncludeDeleted()
                                  .ToListAsync();
   foreach (var item in allItems) {
      await diaryTable.UndeleteAsync(item);
   }
}
```

Can call this method on non-deleted records, add the deleted flag into your entity, or compare the **`IncludeDeleted`** list against the data returned without this flag

Xamarin University

Adding optional parameters

- Can pass optional URI parameters to any of the operations when using a custom web service endpoint or to invoke other OData filters

```
var entry = new DiaryEntry { Text = "Some Entry" };
var uri_params = new Dictionary<string,string> {
   { "require_audit", "true" },
};

try {
   await diaryTable.InsertAsync(entry, uri_params);
}
...
```

```
POST /tables/diary_entry?require_audit=true
```

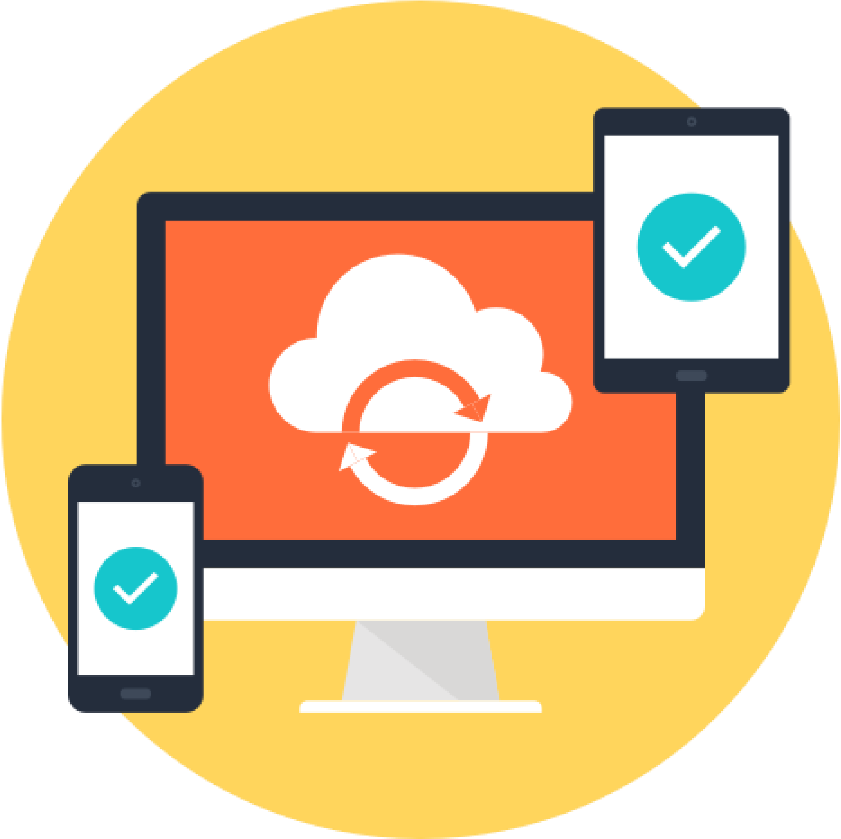
Summary

1. Accessing an Azure DB table
2. Define the data transfer object
3. Adding a new record to the DB
4. Performing queries

Add support for offline synchronization

Tasks

1. Explore the benefits of offline synchronization
2. Include support for SQLite
3. Setup the local cache
4. Synchronize to the online database



Online vs. Offline access

- Mobile devices often find themselves without network access
- Apps can choose to either stop working or provide some kind of offline cache which is synchronized when connectivity is restored
- Data synchronization is a complicated problem that requires design thought (see ENT410 for details)

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

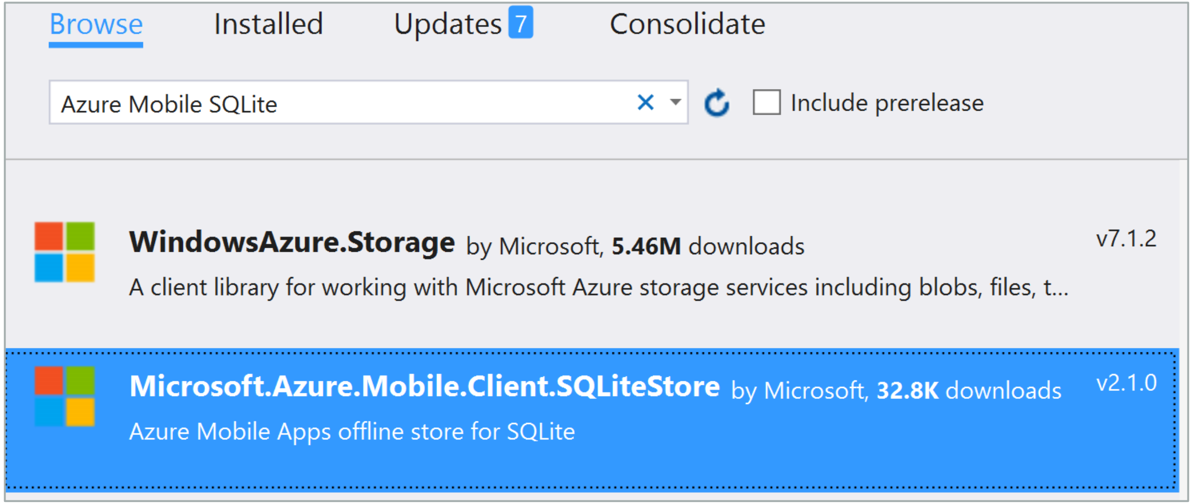
Do I need offline support?

- Adding support for offline synch isn't always necessary or even desired – it has security, storage and potentially network ramifications

- Can store rarely-updated or read-only tables on your own vs. using the Azure offline capability to minimize the overhead or take more control over the cache

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

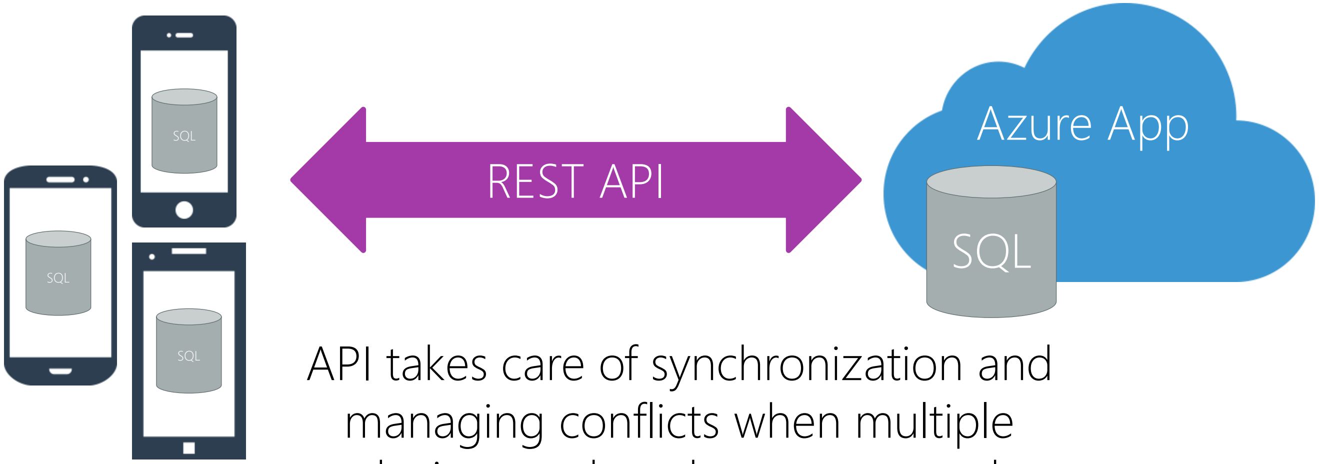


WindowsAzure.Storage by Microsoft, **5.46M** downloads v7.1.2
A client library for working with Microsoft Azure storage services including blobs, files, t...

Microsoft.Azure.Mobile.Client.SQLiteStore by Microsoft, **32.8K** downloads v2.1.0
Azure Mobile Apps offline store for SQLite

Storing data locally

- To support offline data caching, Azure client utilizes a local database which is a local copy of the cloud database

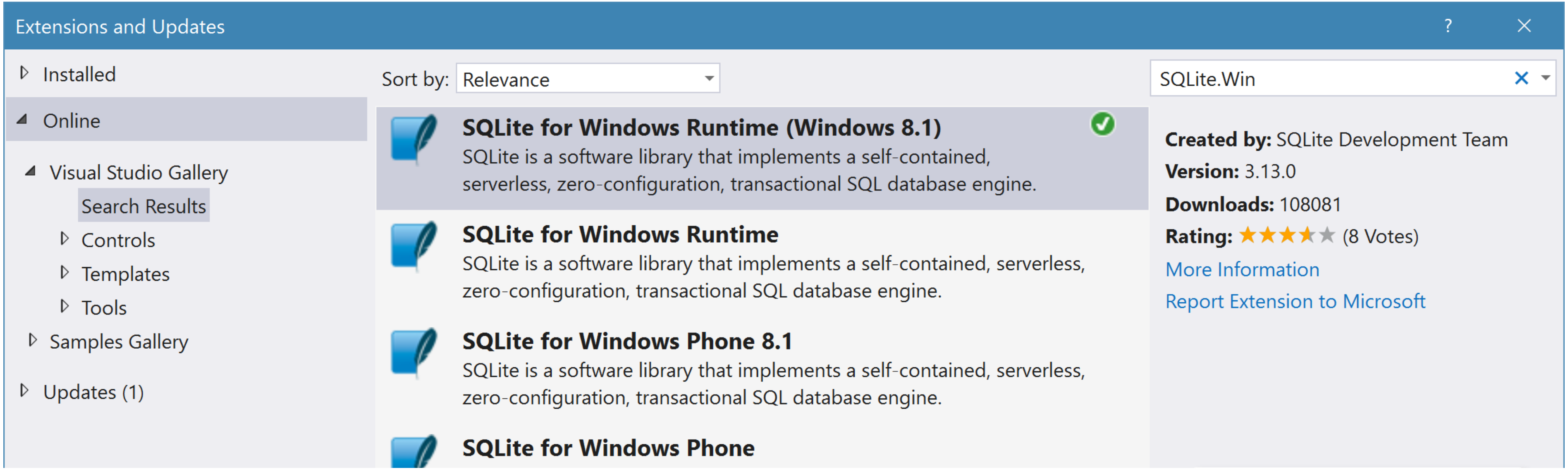


API takes care of synchronization and managing conflicts when multiple devices update the same record

By default, .NET/Xamarin apps use SQLite as the local database store, but this is a configurable feature of the Azure client SDK

Supporting SQLite on Windows

- VS does not come with SQLite pre-installed for Windows apps, but you can add the SDK through the **Tools > Extensions & Updates** dialog; this only needs to be done once as it installs into a global location


Extensions and Updates

Sort by: Relevance

SQLite.Win

SQLite for Windows Runtime (Windows 8.1)
SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine.

SQLite for Windows Runtime
SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine.

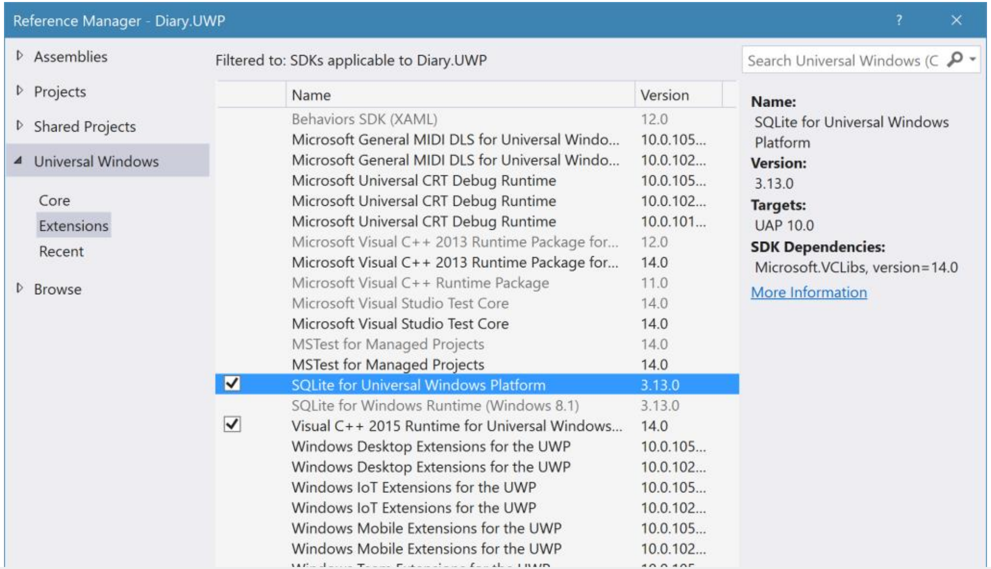
SQLite for Windows Phone 8.1
SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine.

SQLite for Windows Phone

Created by: SQLite Development Team
Version: 3.13.0
Downloads: 108081
Rating: ★★★★☆ (8 Votes)

SQLite for Windows / UWP

- Then you can add a reference to the binary from the extensions section



Reference Manager - Diary.UWP

Filtered to: SDKs applicable to Diary.UWP

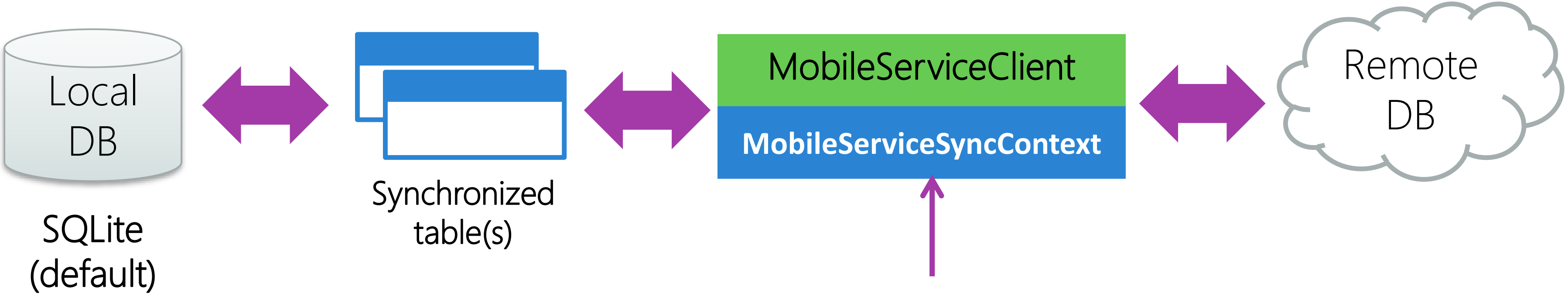
	Name	Version
	Behaviors SDK (XAML)	12.0
	Microsoft General MIDI DLS for Universal Windo...	10.0.105...
	Microsoft General MIDI DLS for Universal Windo...	10.0.102...
	Microsoft Universal CRT Debug Runtime	10.0.105...
	Microsoft Universal CRT Debug Runtime	10.0.102...
	Microsoft Universal CRT Debug Runtime	10.0.101...
	Microsoft Visual C++ 2013 Runtime Package for...	12.0
	Microsoft Visual C++ 2013 Runtime Package for...	14.0
	Microsoft Visual C++ Runtime Package	11.0
	Microsoft Visual Studio Test Core	14.0
	Microsoft Visual Studio Test Core	14.0
	MSTest for Managed Projects	14.0
	MSTest for Managed Projects	14.0
☑	SQLite for Universal Windows Platform	3.13.0
	SQLite for Windows Runtime (Windows 8.1)	3.13.0
☑	Visual C++ 2015 Runtime for Universal Windows...	14.0
	Windows Desktop Extensions for the UWP	10.0.105...
	Windows Desktop Extensions for the UWP	10.0.102...
	Windows IoT Extensions for the UWP	10.0.105...
	Windows IoT Extensions for the UWP	10.0.102...
	Windows Mobile Extensions for the UWP	10.0.105...
	Windows Mobile Extensions for the UWP	10.0.102...
	Windows Team Extensions for the UWP	10.0.105...

Name:
SQLite for Universal Windows Platform
Version:
3.13.0
Targets:
UAP 10.0
SDK Dependencies:
Microsoft.VCLibs, version=14.0
More Information

SQLite runtime is a native binary written in C/C++ and will require that your platform project target either x86 or x64 once it is installed into the project

Synchronization actors

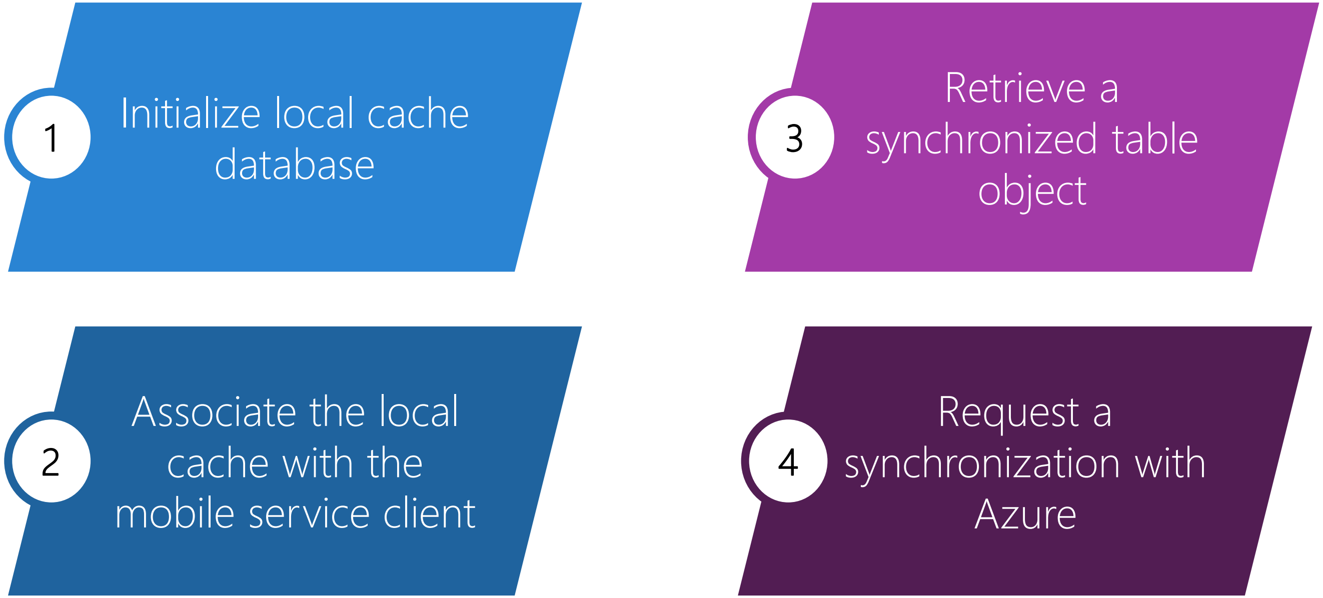
- Several participants when dealing with offline synchronization



Core logic is contained in the **SyncContext** which is owned by the mobile service client; this manages the synchronization and conflict resolution between the DBs

Steps to add offline sync support

- Add support for offline synchronization to your app in four steps:



1 Initialize local cache database

2 Associate the local cache with the mobile service client

3 Retrieve a synchronized table object

4 Request a synchronization with Azure

Initialize the SQLite local cache

- Need a **`MobileServiceSQLiteStore`** to manage the local cache – this identifies the local file which will be used to store a cached copy of the data for offline access

```
mobileService = new MobileServiceClient(AzureEndpoint);
...
var store = new MobileServiceSQLiteStore("localstore.db");
```

Must pass in a filename which will be created on the device's file system

Initialize the SQLite local cache

- Next, define the table structure based on your entity object; this must be done once per app-launch for each entity to ensure the SQLite store knows how to map columns to entity properties

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

↑

this will use reflection and generate an internal SQL table mapping definition for the type referenced

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());
```

SyncContext property is used to perform synchronization requests, note that this method is async – it will initialize the DB store and potentially create

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());
```

IMobileServiceSyncHandler is an extension point to process each table operation as it's pushed to the remote DB and capture the result when it completes

What if I don't want to use SQLite?

- Store is actually an **IMobileServiceLocalStore** interface – can define your own implementation to use something other than SQLite

```
class MyCustomXMLStore : IMobileServiceLocalStore
```

```
mobileService = new MobileServiceClient(AzureEndpoint);
...
var store = new MyCustomXMLStore("localstore.xml");
...
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
}
```

The difference is that this now works even if we aren't connected to the network!

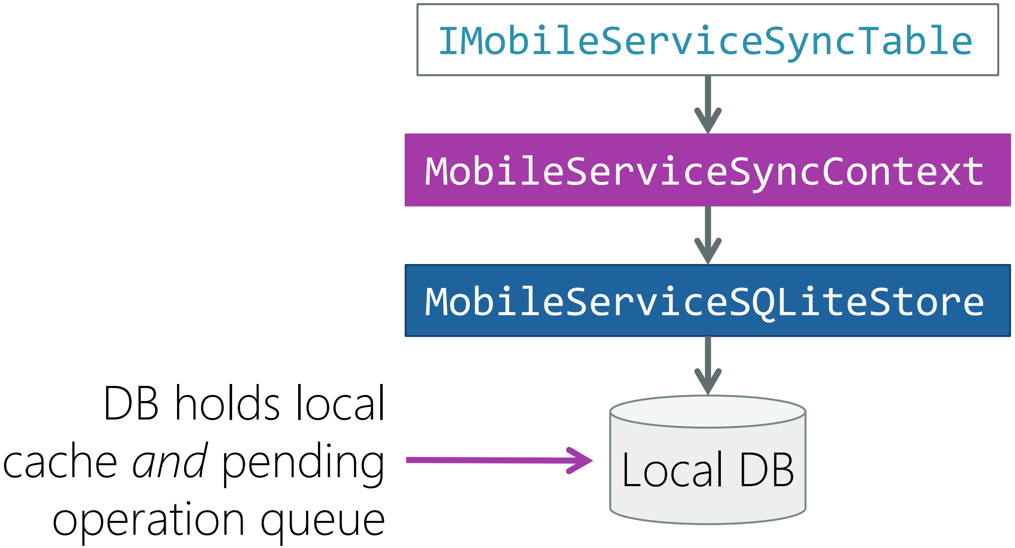
Individual Exercise

Add support to our app for offline data caching

Xamarin University

Last step: synchronize our changes

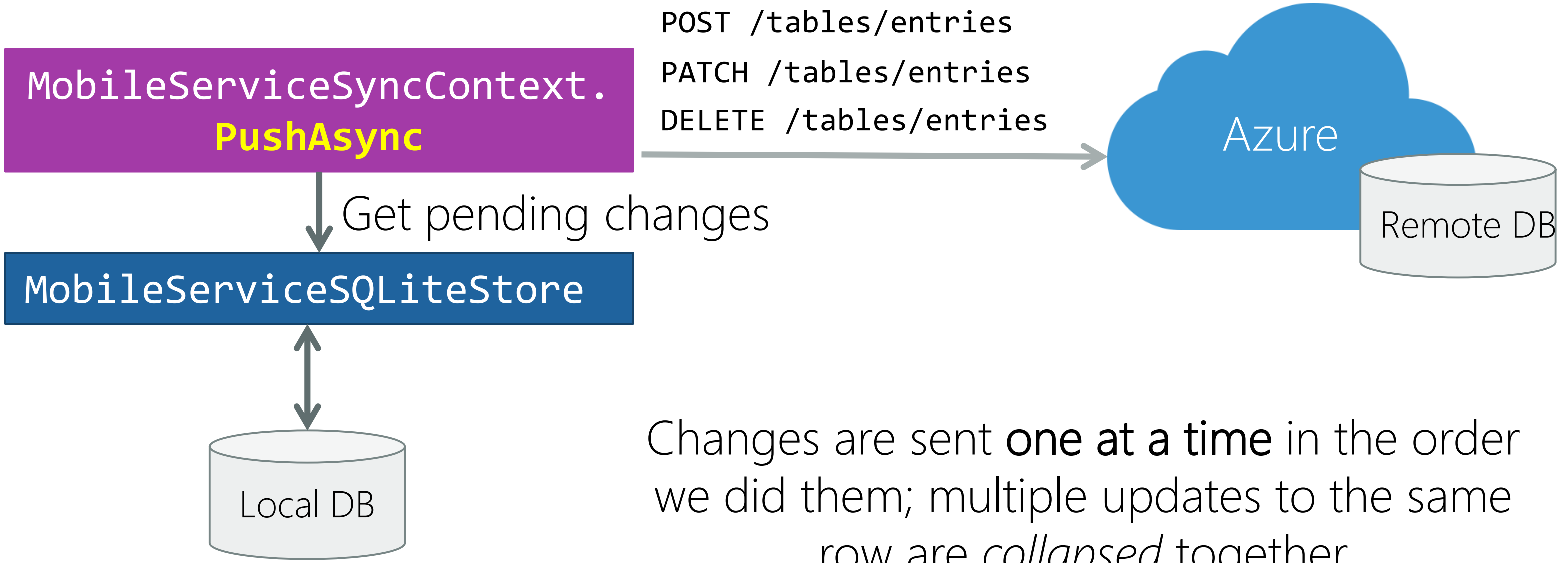
- When a synchronization context is initialized with a local data store, all your queries and updates are always performed **locally** and then queued up for synchronization to Azure



DB holds local cache *and* pending operation queue

Last step: synchronize our changes

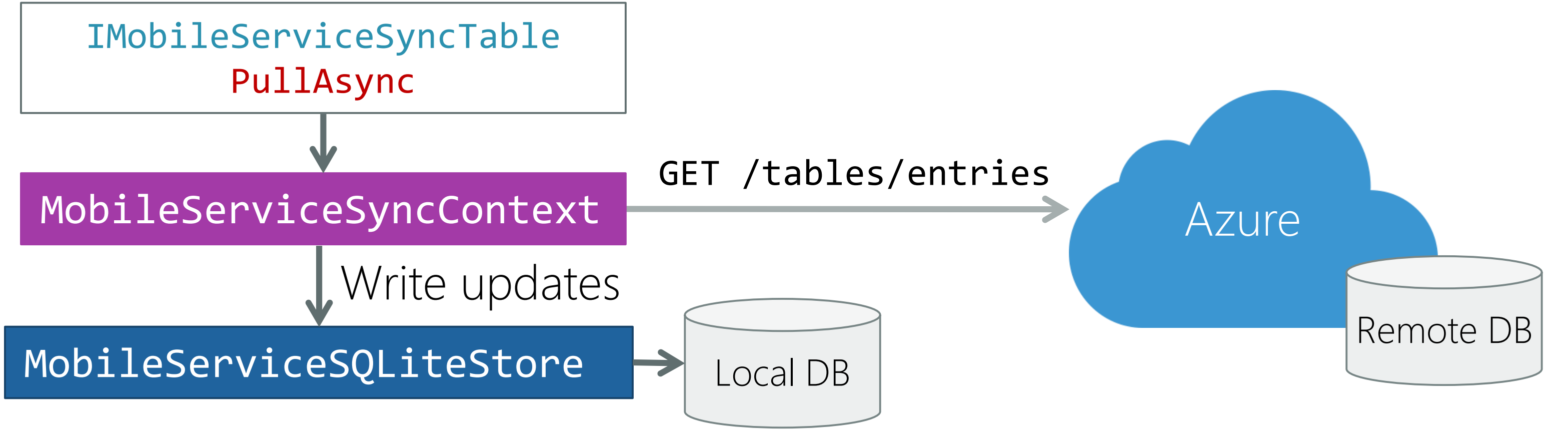
- To synchronize to the Azure remote database, your code must perform two operations; first we *push* all pending changes up to the remote DB



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

Last step: synchronize our changes

- Next, we *pull* new and updated records from the remote DB back to our local copy on a *table-by-table* basis using the **`IMobileServiceSyncTable`**



Optimizing the network traffic

- Can direct the pull operation to use an *incremental sync* which utilizes the **updatedAt** column to only return the records after that timestamp

```
IMobileServiceSyncTable
 PullAsync ("queryId")
```

This feature is activated by passing a query id to **`PullAsync`**

```
GET /tables/diary_entry?$filter=updatedAt%20ge%20value$skip=0&$take=50
GET /tables/diary_entry?$filter=updatedAt%20ge%20value$skip=50&$take=50
GET /tables/diary_entry?$filter=updatedAt%20ge%20value$skip=100&$take=50
```

Example: synchronizing the DB

- Should perform a synchronization on startup to sync up the local DB and then each time you make a change to the database

```
public async Task<DiaryEntry> UpdateAsync(DiaryEntry entry)
{
    // Update local DB
    await diaryTable.UpdateAsync(entry);

    // Our method to push changes to the remote DB
    await SynchronizeAsync();

    return entry;
}
```

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
    }
}
```

Synchronizing the DB

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

    try
    {
        await MobileService.S
        await diaryTable.Pull                                      ));
    }
    catch (Exception ex)
    {
        // TODO: handle error
    }
}
```

Can use Connectivity NuGet plug-in to check for network availability; don’t attempt synchronization if we don't have a network connection

https://www.nuget.org/packages/Xam.Plugin.Connectivity/ See XAM150 for more info

Synchronizing the DB

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

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

Always push local changes first – this can fail if something else updated one or more of our locally changed records

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
    {
        // TODO: handl
    }
}
```

Then pull remote changes for each table, must pass text identifier and query to execute remotely

Synchronizing the DB

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

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

Can omit call to **PushAsync** if you are going to pull changes back – system will automatically do an implicit push if you don't

Pulling data from the server

- Must pass a **query** to define the records to pull from the remote database

```
await diaryTable.PullAsync(null,
              diaryTable.CreateQuery());
```

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

Can provide filtered query to pull down a subset of the records you want to refresh in your local copy

Pulling 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

Forcing a full synch

- Can force the client to throw away local cache and refresh completely from the server if it has stale data by calling **`PurgeAsync`**

```
await diaryTable.PurgeAsync();
```

Can purge all records for a table

```
await diaryTable.PurgeAsync("purgeAllPrivate",
         diaryTable.Where(d => d.IsPrivate);
```

Or can specify a query to purge specific records

This is particularly important if soft deletes are *not* enabled on the server because deleted records will not be removed from the local cache

Individual Exercise

Synchronizing to the remote database

Xamarin University

Updating things while offline

- Changing data while offline has some risk – Azure *optimistically* just assumes it will all work .. but what if …

While offline, a client changes a row and sometime later pushes the changed record to Azure, but the row has been changed by someone else …

While offline, a client deletes a record and when the app tries to push the delete to Azure, it finds the record was changed by someone else…

While online, the client makes a change to a row that causes a constraint failure in the remote database so the remote DB cannot apply the change

Updating thi ffline

- Changing d Azure *optimistically* just assumes i

While of
sometime
Azure, b

While online, the
row that causes a co
remote database so the
apply the change

e, a client deletes a record and
app tries to push the delete to
record was changed by
ne else…

Automatic conflict resolution

- Azure supports automatic conflict resolution in cases where the same record is modified by two clients through the **version** column; however to turn this feature on you have to map it in your DTO shape

```
public class DiaryEntry
{
    ...
    [Version]
    public string AzureVersion { get; set; }
}
```

```
{
    "id":"5c6e6617-117a-...",
    "createdAt":"...",
    "updatedAt":"...",
    "version":"AAAAAAAAB/4=",
    "deleted":false,
    "text":"Hello, Diary"
}
```

Adding the property ensures we send it *back* to the server, otherwise our record will always just replace the server record

Dealing with failure

- If Azure detects a conflict (using version), it will respond with an HTTP error which is translated to an exception

```
try
{
   await MobileService.SyncContext.PushAsync();
   ...
}
catch (MobileServicePushFailedException ex)
{
   // TODO: handle error
}
```

Getting the result of the push

- **`MobileServicePushFailedException`** includes a **`PushResult`** property which includes a status and a collection of table errors which occurred as a result of the push request

```
public class MobileServicePushCompletionResult
{
    public MobileServicePushStatus Status { get; }
    public List<MobileServiceTableOperationError> Errors { get; }
}
```

Each conflict is described by a **table error** – this contains the passed client value, the server value and details about the operation so we can decide what to do

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);
      }
   }
}
```

How do you handle conflict?

- There are several valid options you can take when a conflict is reported from Azure

Last Man (update) Wins!

Allow the user to select the one they want

Merge the client and server records

Cancel the update and use the server version

Conflict resolution possibilities

- Table error includes methods to resolve conflict; app must decide what to do based on the data and business requirements

I want to	Use this method
Throw away my local changes and revert back to my initial version	`CancelAndDiscardItemAsync`
Throw away my local changes and updates to the server version	`CancelAndUpdateItemAsync`
Update my local item with a new version and re-sync to the server	`UpdateOperationAsync`

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<DiaryEntry>();
    var localItem = error.Item.ToObject<DiaryEntry>();
    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));
    }
}
```

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<DiaryEnt
    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

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<DiaryEntry>();
    var localItem = error.Item.ToObject<Dia
    if (serverItem.Text == localItem.Text)
        // Items are the same, so ignore th
        await error.CancelAndDiscardItemAsy
    }
    else {
        // Always take the client; update the Version# and resubmit
        localItem.AzureVersion = serverItem.AzureVersion;
        await error.UpdateOperationAsync(JObject.FromObject(localItem));
    }
}
```

Otherwise, always assume our copy is the best one – copy the version over and re-submit to Azure

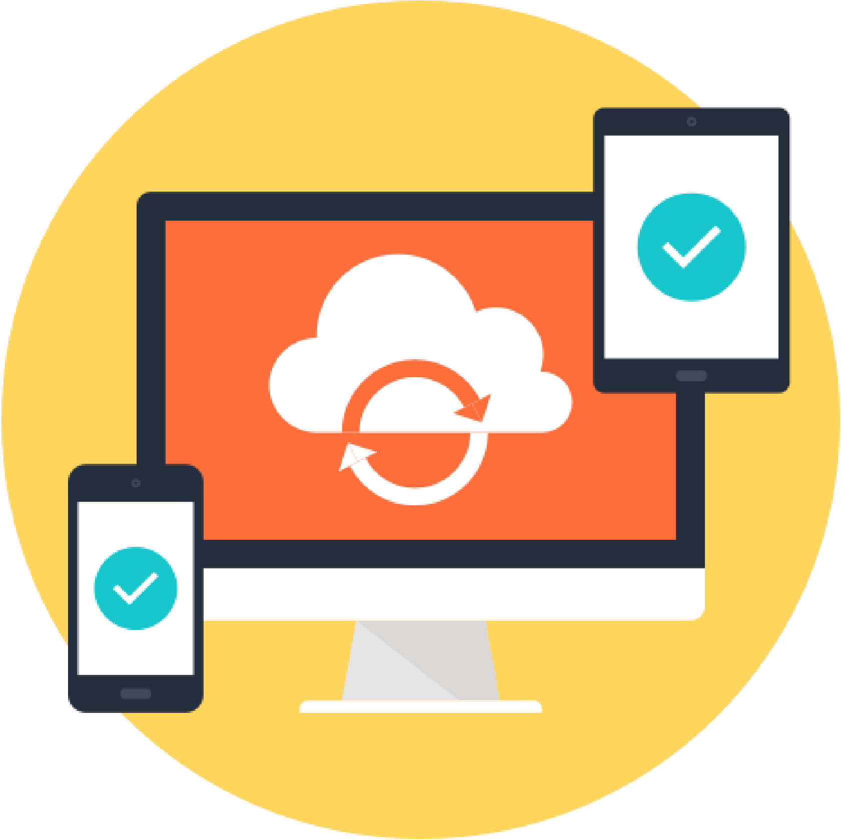
Homework Exercise

Add error recovery code to support conflicts

Xamarin University

Summary

1. Explore the benefits of offline synchronization
2. Include support for SQLite
3. Setup the local cache
4. Synchronize to the online database

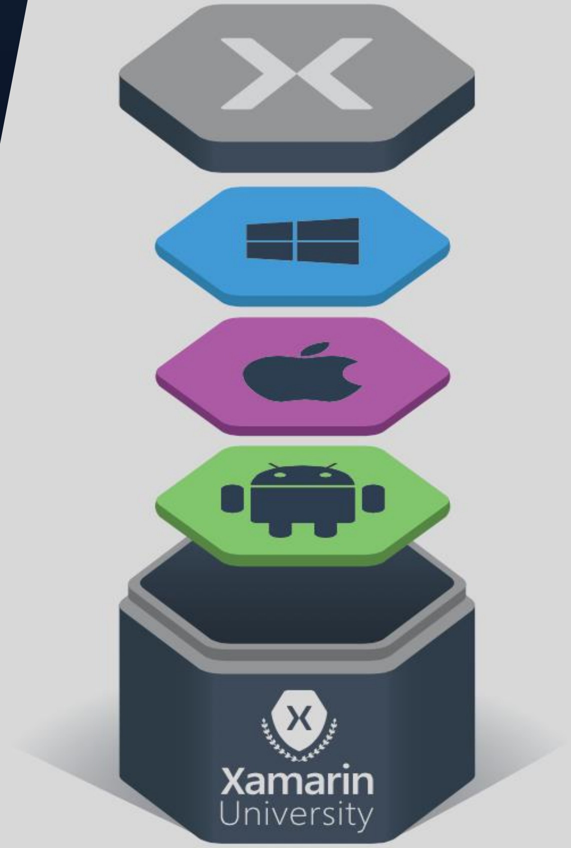


Next Steps

- We've covered the basics of building a mobile app with Azure support
- In the next set of classes we will add to this knowledge by supporting authentication, and push notifications

Thank You!

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



Microsoft