Introduction to Azure App Services

What is Azure?

- Azure is Microsoft's subscription-model cloud-computing platform
  - Processing
  - Storage
  - Networking
  - API services

Azure datacenters

Azure has regional data centers all over the world which supports your infrastructure needs such as placing your services near your customers

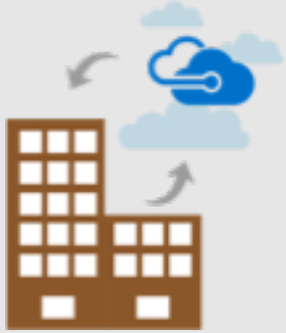
See: https://azure.microsoft.com/en-us/regions/#services for current list

Why use Azure?

There are several key benefits to moving your systems to Azure

Maintenance is reduced

Better security



Supports hybrid deployment

99.9% uptime SLA

Azure services

Azure offers a comprehensive set of services – this shows only a small selection of what is available

Compute
- Virtual Machine
- Service Fabric
- Batch
- Containers

Networking
- Virtual Network
- Load Balancer
- VPN Gateway
- DNS

Storage
- BLOB Storage
- File Storage
- Backup
- Site Recovery

Web + Mobile
- Web Apps
- Mobile Apps
- Notification Hubs

Databases
- SQL Database
- SQL Data Warehouse
- Redis Cache
- DocumentDB

Intelligence + Analytics
- HDInsight
- Machine Learning
- Stream Analytics

Internet of Things
- IoT Hub
- Event Hubs

Enterprise Integration
- Biztalk Services
- Service Bus
- Data Factory

Security + Identity
- Key Vault
- Active Directory
- Multi-Factor Auth

Developer Tools
- VS Team Services
- API Management
- HockeyApp

Monitoring + Management
- Portal
- Automation
- Application Insights

Azure App Service

Azure App Service groups together several Azure services that support the development of several common app types

Web Apps
Web apps that scale with your business

Mobile Apps
Build mobile apps for any device

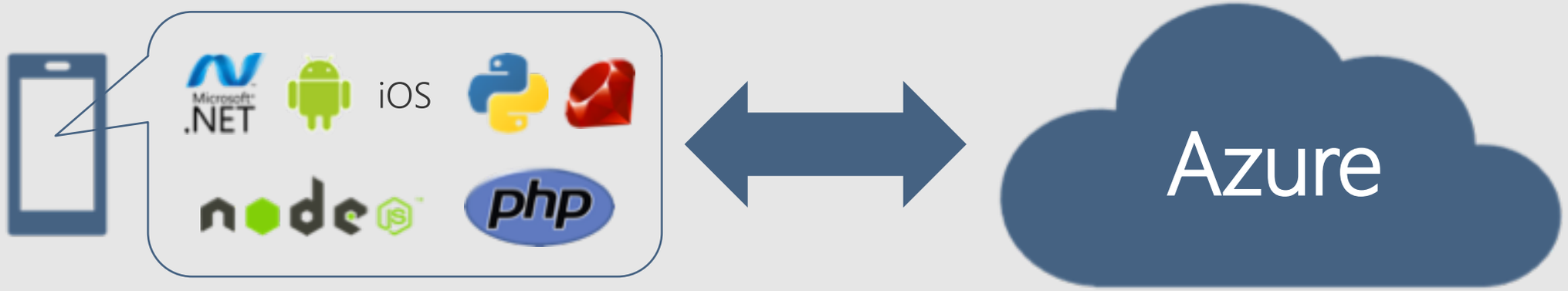
Logic Apps
Automate business processes across SaaS and on-premises

API Apps
Build and consume APIs in the cloud

Functions
Listen and react to events across your stack

Client SDKs

Microsoft provides client-side SDKs for most common technologies (available from https://azure.microsoft.com/downloads/)



Developer builds their client app with the SDK that matches their preferred language

The SDK gives programmatic access to Azure services

Demonstration

Reminder: Azure Apps

Azure App Services is a PaaS offering for web, mobile and integration scenarios

Web App

Used to create a hosted IIS-based website

Function App

Server-side scheduled jobs

Mobile App

WebAPI + data app for apps using Mobile Client SDK

API App

Used to create a hosted RESTful web service

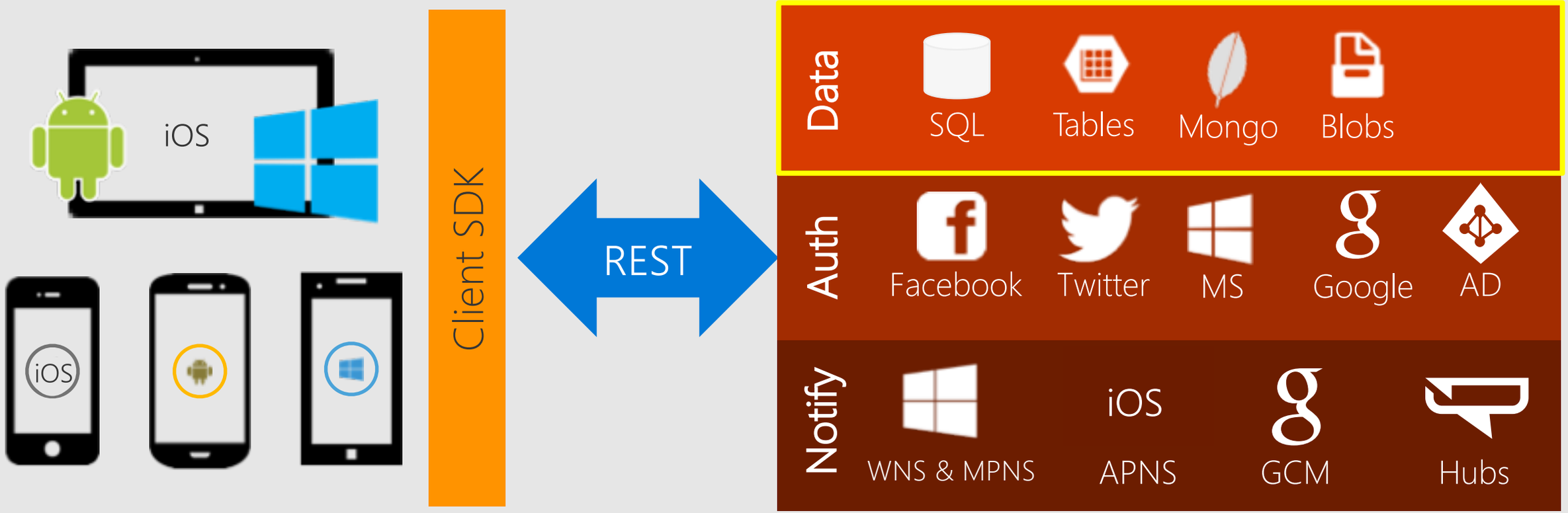
Logic App

Workflow integration with data conversion + logic

Could also use **API App** and access the service using the techniques shown in **XAM150**

Features of an Azure Mobile app

Azure Mobile App provides a set of pre-built services you can activate for your mobile app; exposed using web service endpoints



Implementing the back-end service

Azure App services support two back-end technologies on top of IIS, can use either one to define your service code

Uses JavaScript to define and customize server endpoints and point-and-click creation but lacks type safety - This is the default when you use the portal to create your app

Implementing the back-end service

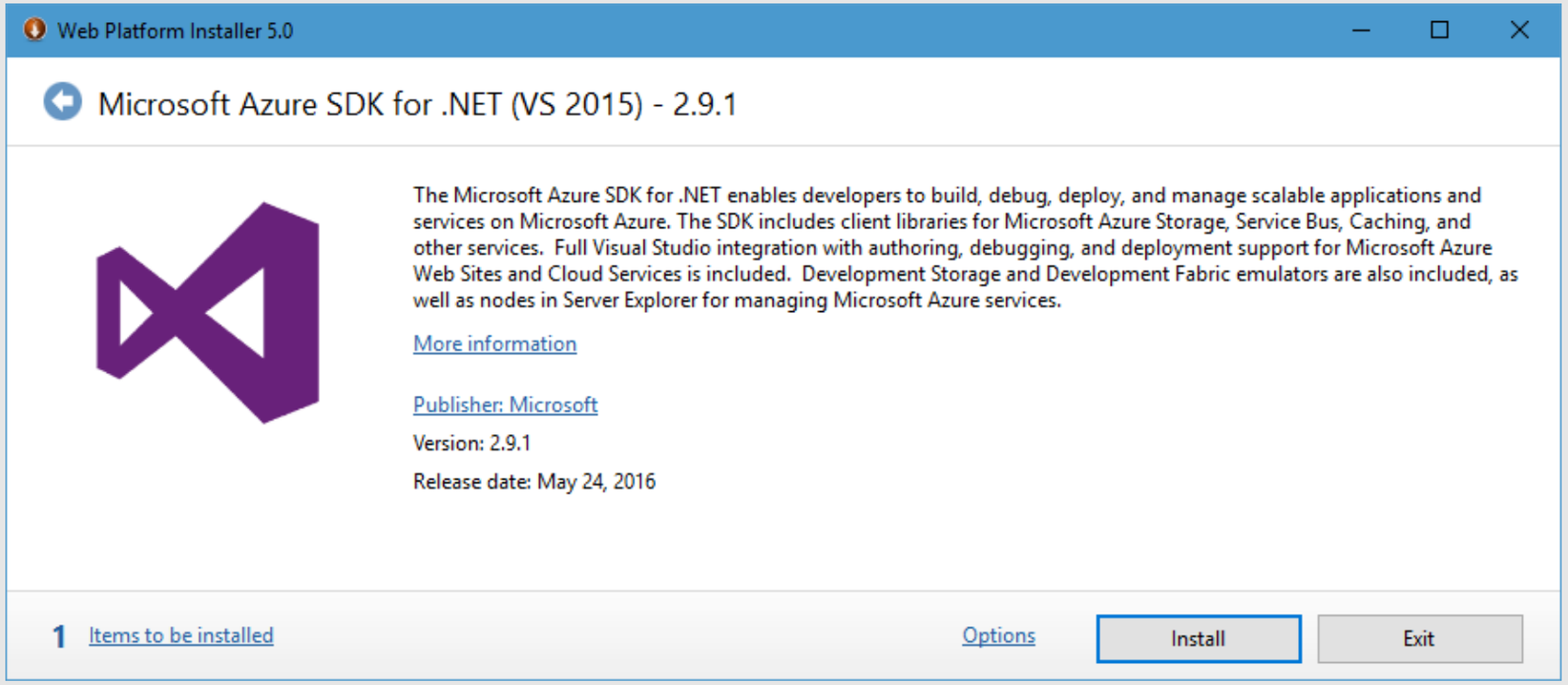
Azure App services support two back-end technologies on top of IIS, can use either one to define your service code

Supports a fully typed programming model (.NET) with a robust and powerful framework but requires more code to setup

ASP .NET

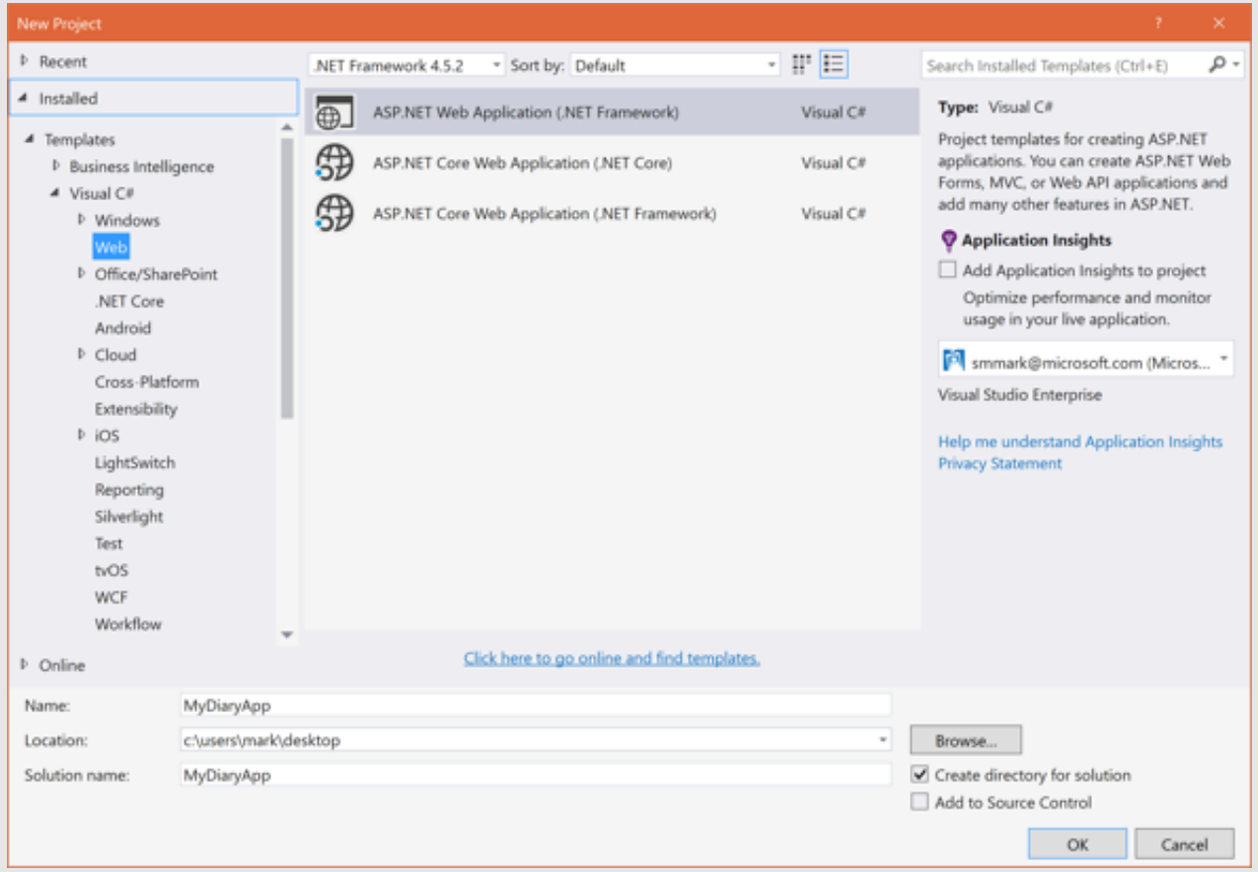
Install the Azure SDK for VS

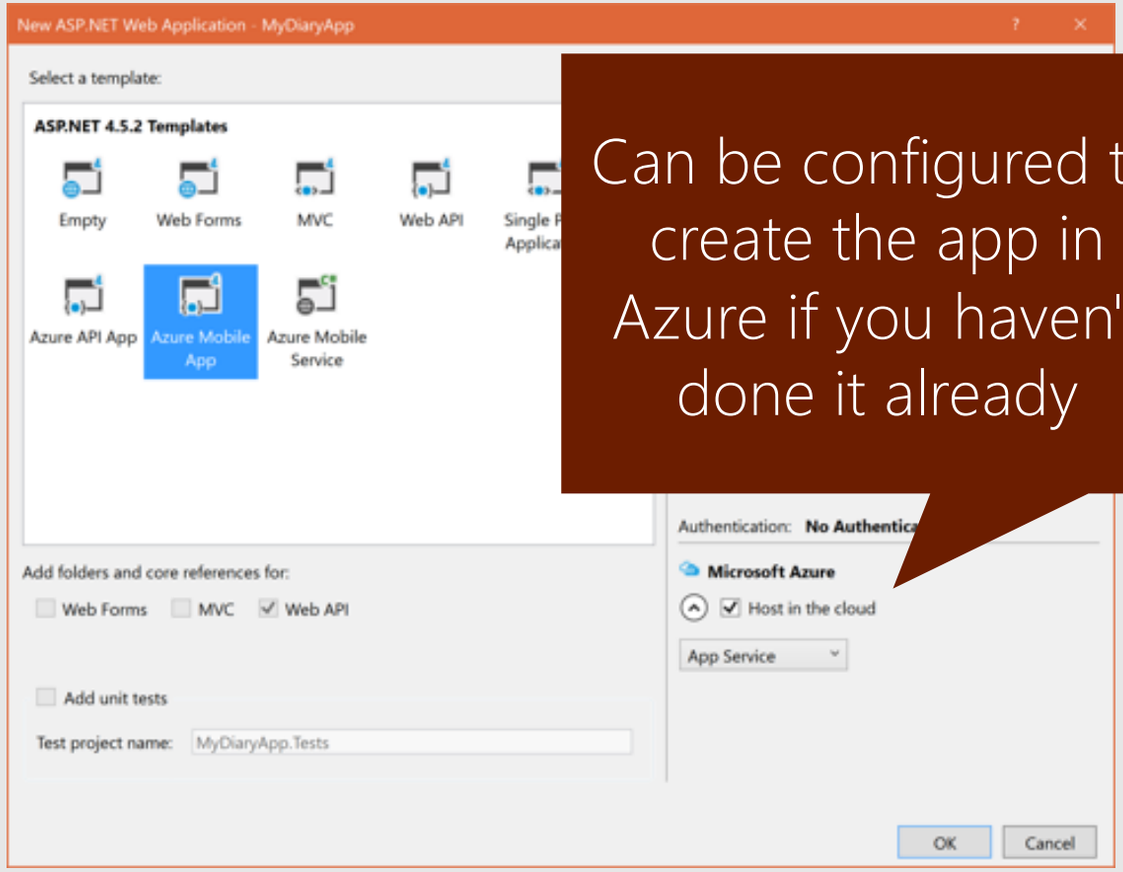
Must install the Azure SDK for .NET for Visual Studio from azure.microsoft.com/downloads to get the components, templates and simulators for Azure



Creating an Azure App service in .NET

Use the **Azure Mobile App** template to create a web service in VS



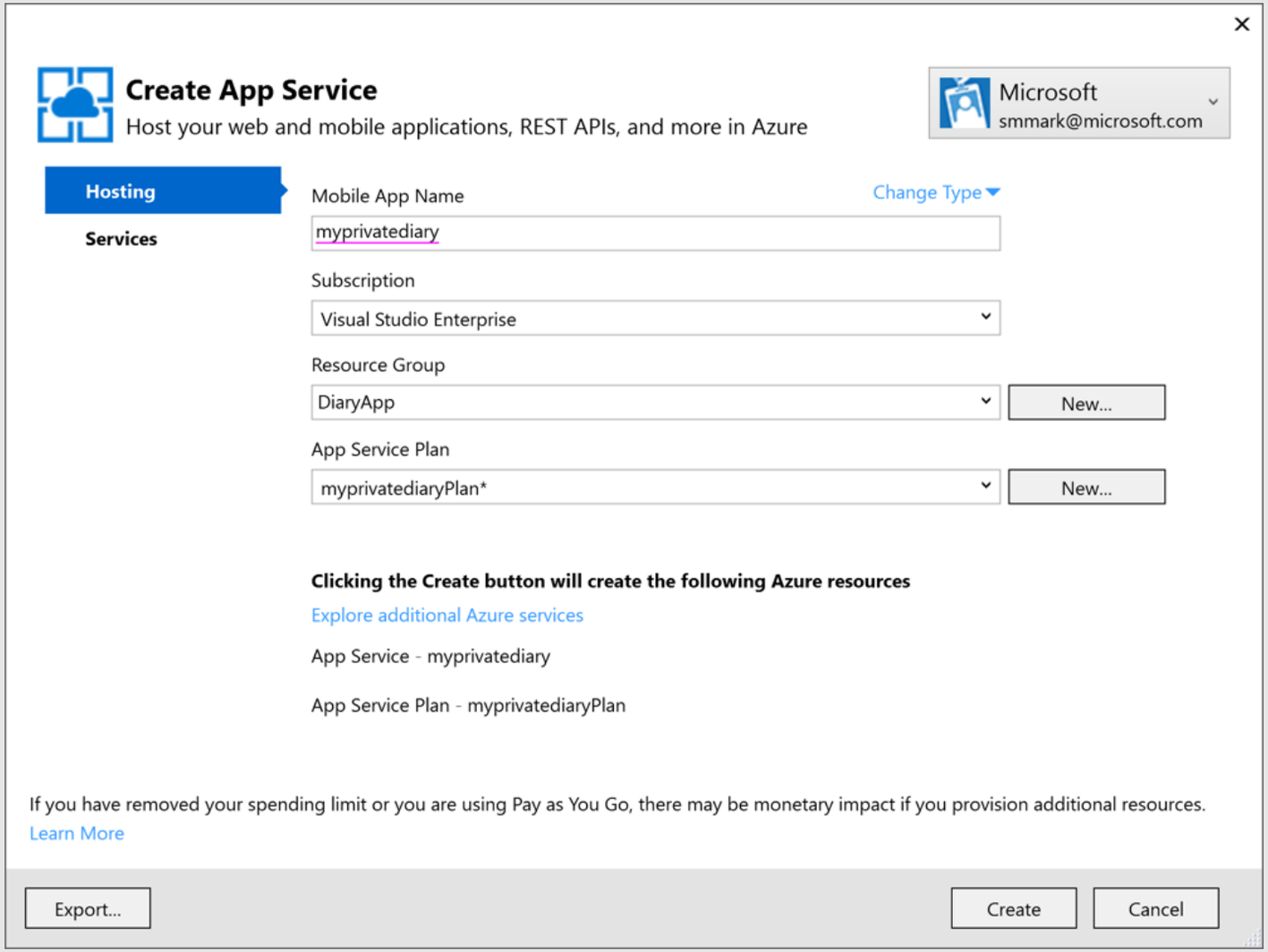


Can be configured to create the app in Azure if you haven't done it already

Defining the Azure app in VS

When VS creates the app, it lets you set all the same options found in the Azure portal wizard

Adds web deployment record to the solution to let you manually publish to Azure with **Build > Publish** menu option

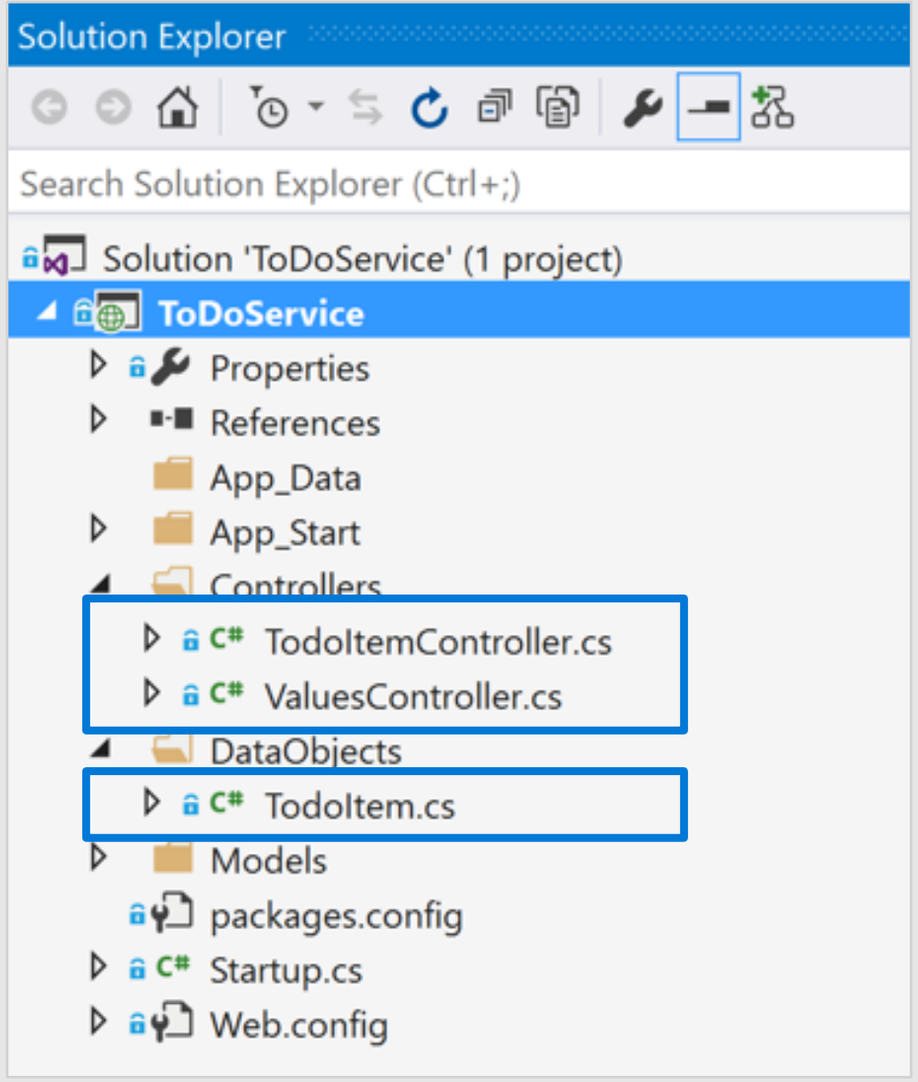


Visual Studio server project

Template creates an ASP.NET starter app

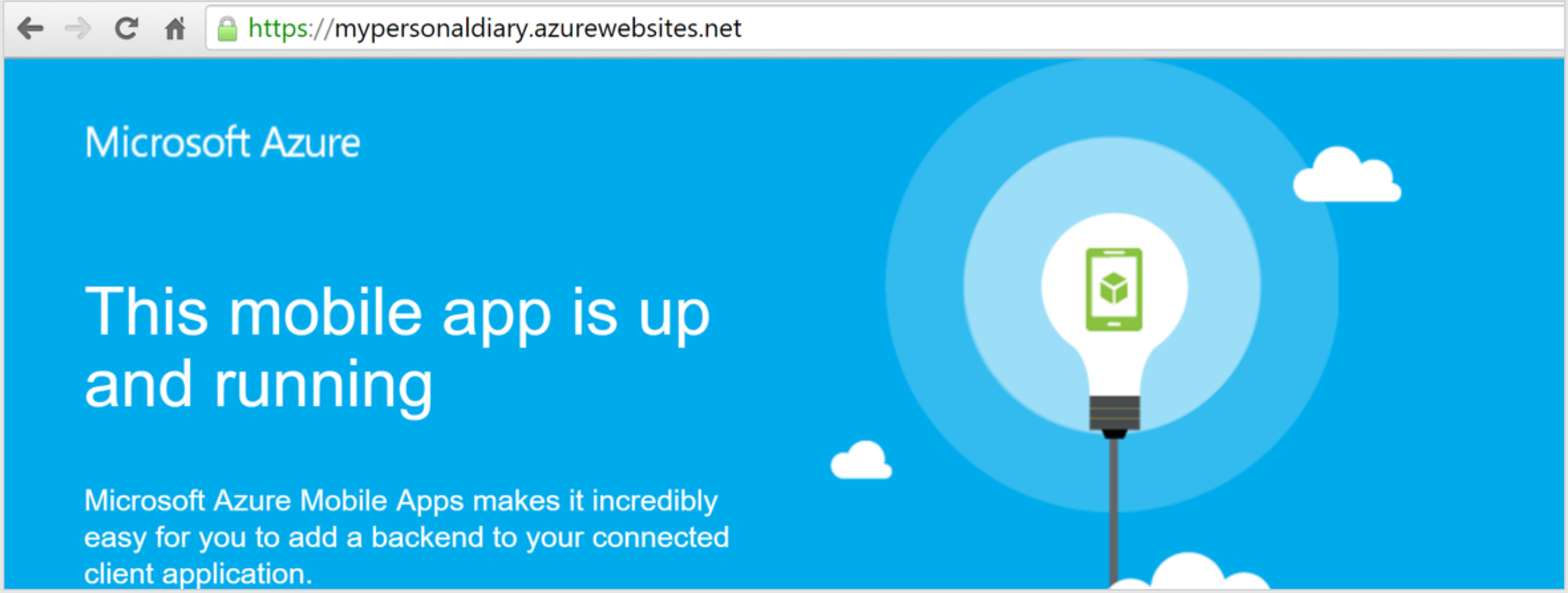
Includes two controllers – one for a database table and another for a basic web service

Defines a Data Transfer Object (DTO) to hold a **TodoItem**



Check your service

Once the service is started, it will respond to GET requests on the URL



Exercise #10

Create the Azure Mobile App Project and Publish It

Supplying data to your app

Most applications will utilize some sort of server-side data - there are several questions to think about as you decide how to store the data

What type of data is it? How is it queried?

How much data will you be storing?

Is the data binary?

Azure data styles

Azure provides several managed storage choices for apps

SQL Database
(relational)

Traditional SQL Server consisting of related tables with columns and rows; supports complex queries and everything SQL Server has to offer (e.g. transactions, indexes, constraints, stored procedures, etc.)

Azure data styles

Azure provides several managed storage choices for apps

SQL Database
(relational)

Table Storage
(NoSQL key/value)

Fast, indexed table retrieval of structured NoSQL data; Table storage tends to cost less than SQL storage for similar volumes

Azure data styles

Azure provides several managed storage choices for apps

SQL Database
(relational)

Large and unstructured file storage; useful for storing media assets and other bits of opaque non-textual data

Blob Storage
(unstructured files)

Azure data styles

Azure provides several managed storage choices for apps

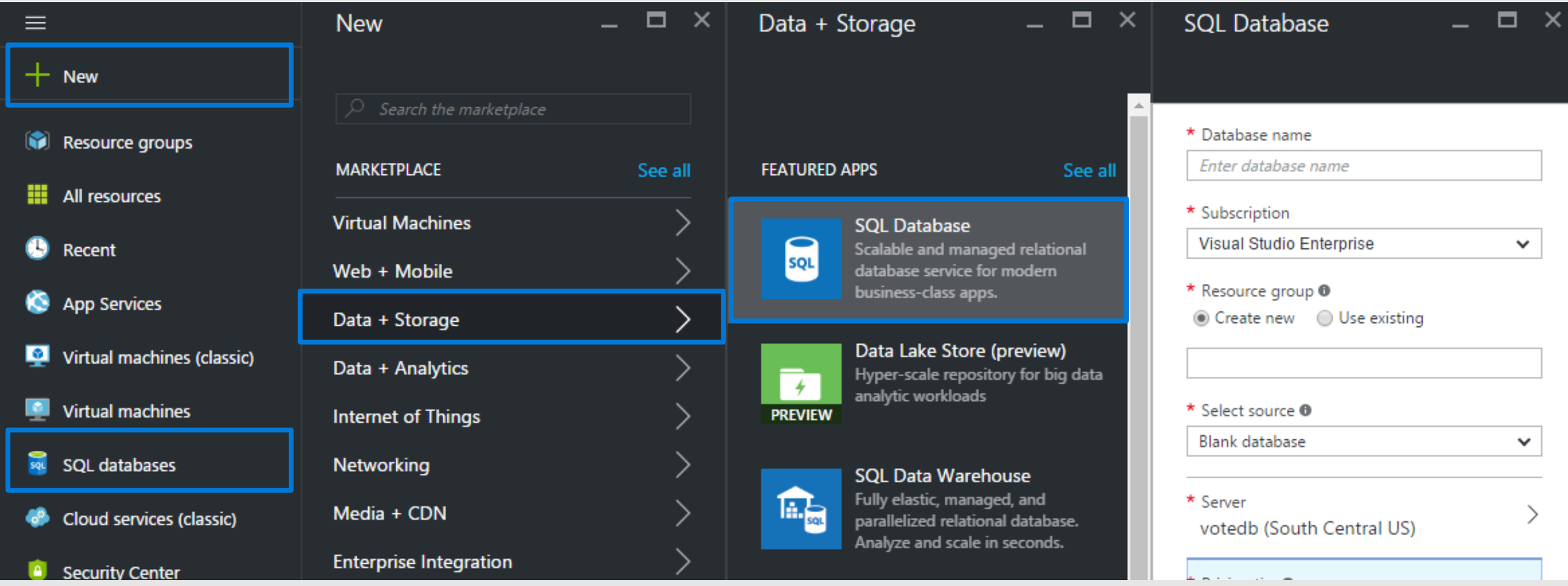
SQL Database
(relational)

Table Storage
(NoSQL key/value)

Blob Storage
(unstructured files)

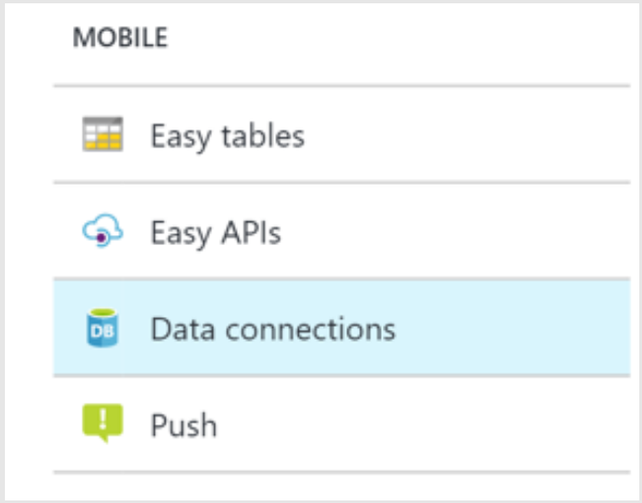
Creating a SQL database in Azure

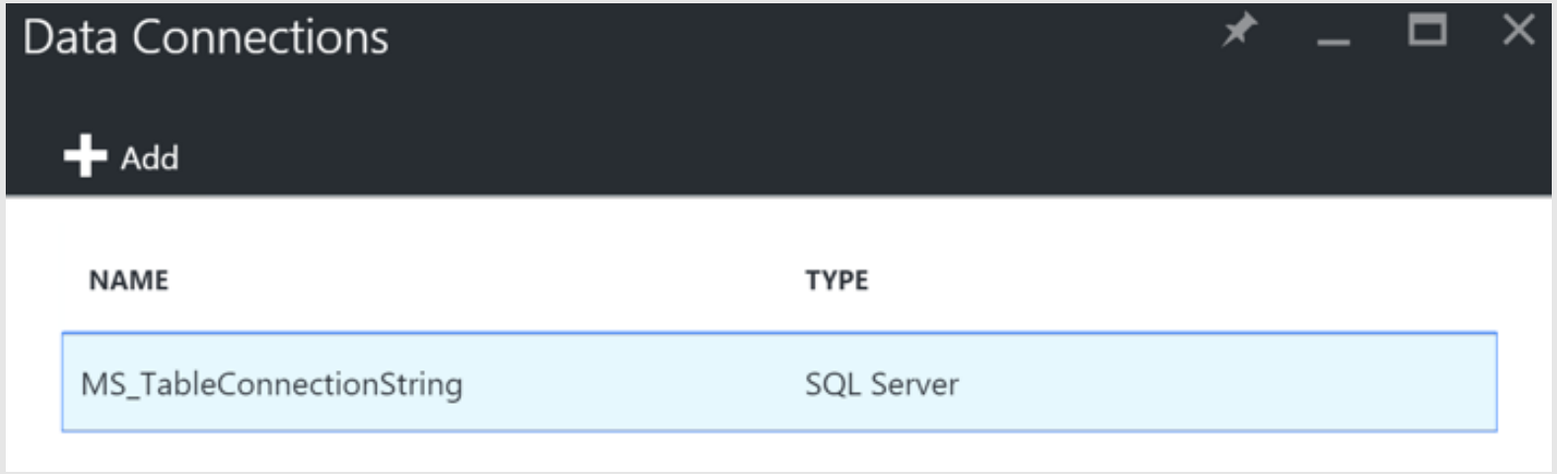
Must have a SQL Server database resource in your Azure portal



Adding a SQL database to your app

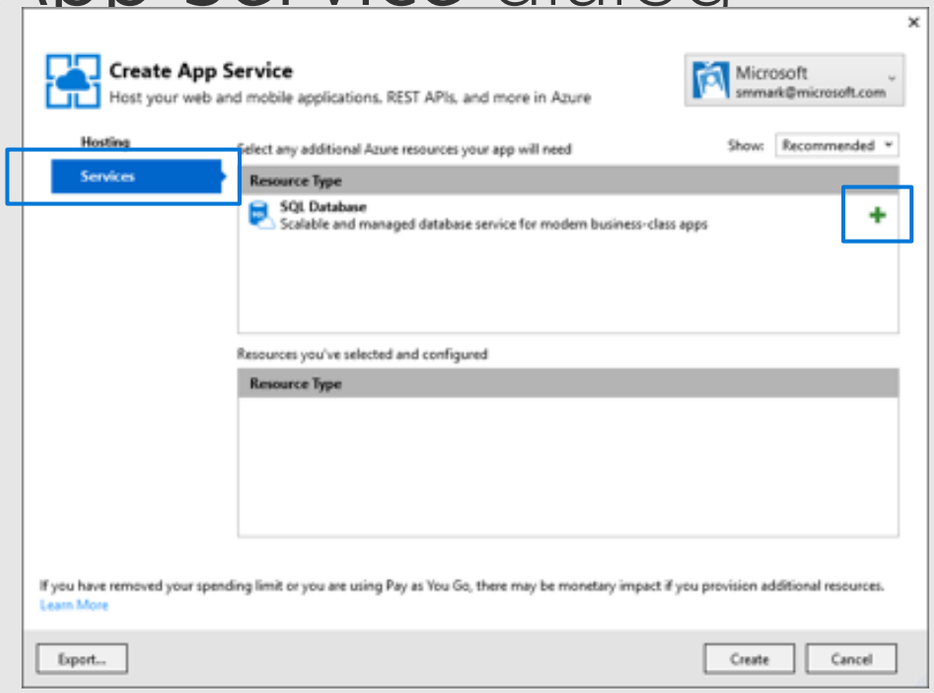
Must add a connection string to the app service; default expected name is **MS_TableConnectionString**

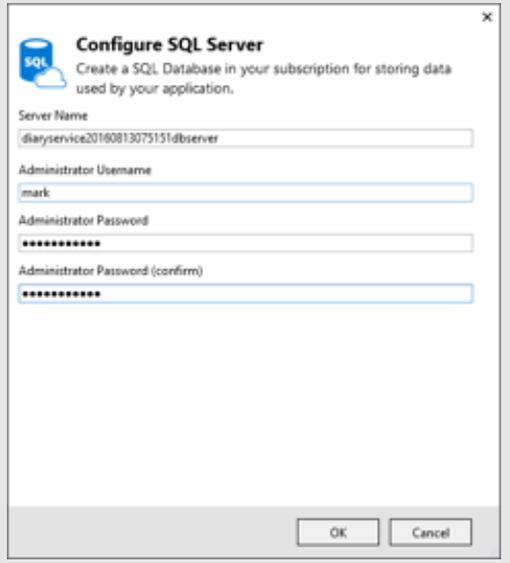




Adding a SQL database with VS

Can also add a SQL database to your application when created through Visual Studio as part of the Azure setup; this must be done at app creation time using the **Services** tab on the **Create App Service** dialog





Exercise #11a

Add a SQL Database

Adding a table to your mobile service

Depending on your back-end, the process for adding a new table will be different however the exposed endpoint will be the same

ASP.NET requires a controller be created to access the database and expose it over a RESTful endpoint; provides complete control over the endpoint and server-side logic applied

Node.js provides a no-code option which is configurable from the Azure management portal; includes some basic extension points for the table operations (read/insert/update/delete)

Adding a table to a .NET back end

ASP.NET projects use a *controller* to expose a SQL server table as an OData web service endpoint; requires two things:

1. Data Transfer Object (DTO)
2. Table Controller (`TableController<T>`)

Step 1: Define the DTO

Data Transfer Objects (DTO) provide the *shape* of the data that will be passed to the client

Must derive from Azure SDK base class which provides DB access support

```
public class TodoItem : EntityData
{
    public string Text { get; set; }
    public bool Complete { get; set; }
}
```

You add custom **public properties** to define your custom data to be stored in the database

What is EntityData?

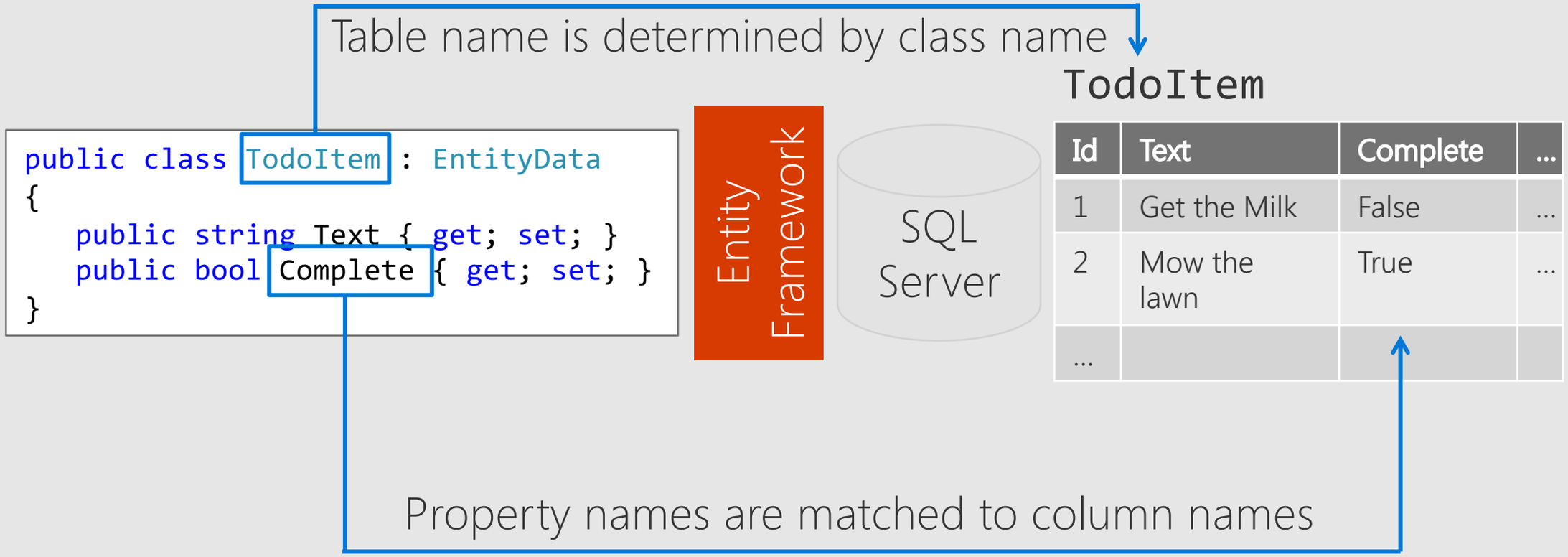
`EntityData` base class provides primary key and required synchronization data which is used/expected by the client/server communication

- Can add these columns to an existing DB, or let EF code-first create them which is the default behavior

```
public abstract class EntityData : ITableData
{
    [Key, TableColumn(TableColumnType.Id)]
    public string Id { get; set; }

    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    [Index(IsClustered = true)]
    [TableColumn(TableColumnType.CreatedAt)]
    public DateTimeOffset? CreatedAt { get; set; }

    [TableColumn(TableColumnType.Deleted)]
    public bool Deleted { get; set; }

    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    [TableColumn(TableColumnType.UpdatedAt)]
    public DateTimeOffset? UpdatedAt { get; set; }

    [TableColumn(TableColumnType.Version), Timestamp]
    public byte[] Version { get; set; }
}
```

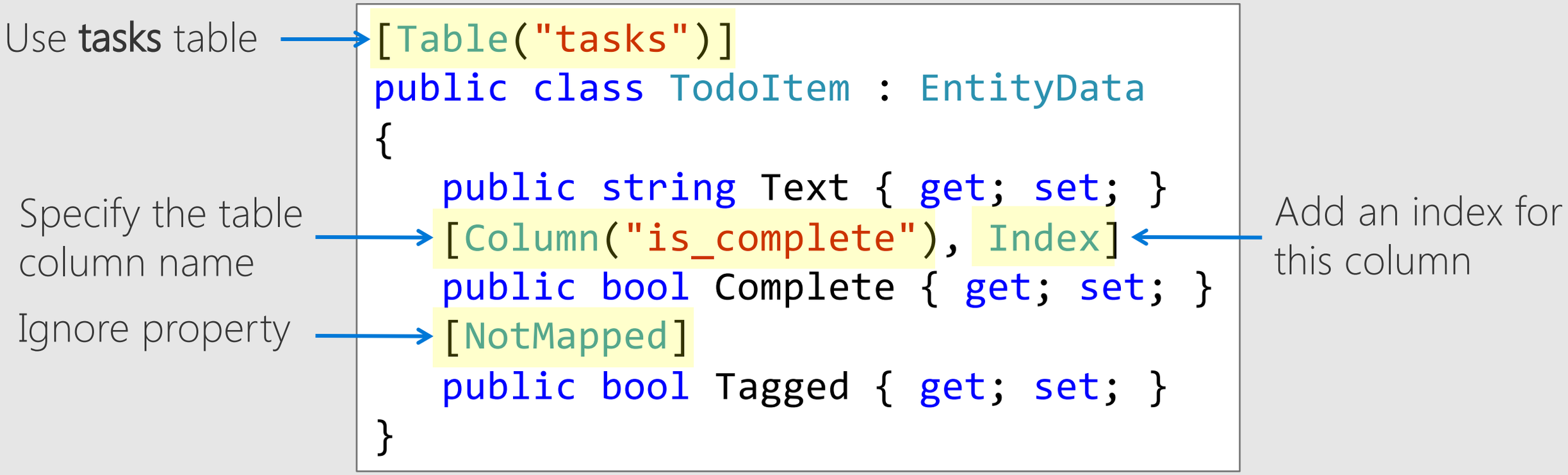
Mapping the DTO to a DB table

DTO is mapped to a single database table using Entity Framework (EF); rows are exposed as instances of the DTO



Customizing the mapping

Can apply **attributes** to customize how the DTO is mapped to the table



JSON attributes

Can change the shape of the object passed over the wire using standard JSON attributes; remember to coordinate with the client!

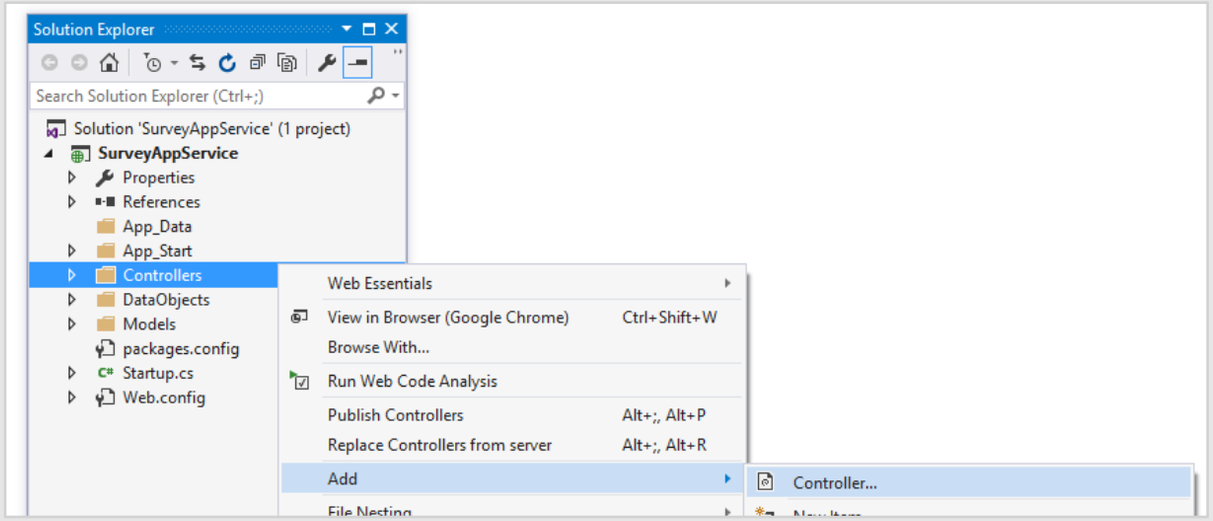
```
public class TodoItem : EntityData
{
    [JsonProperty(Name="todo")]
    public string Text { get; set; }
    public bool Complete { get; set; }
}
```

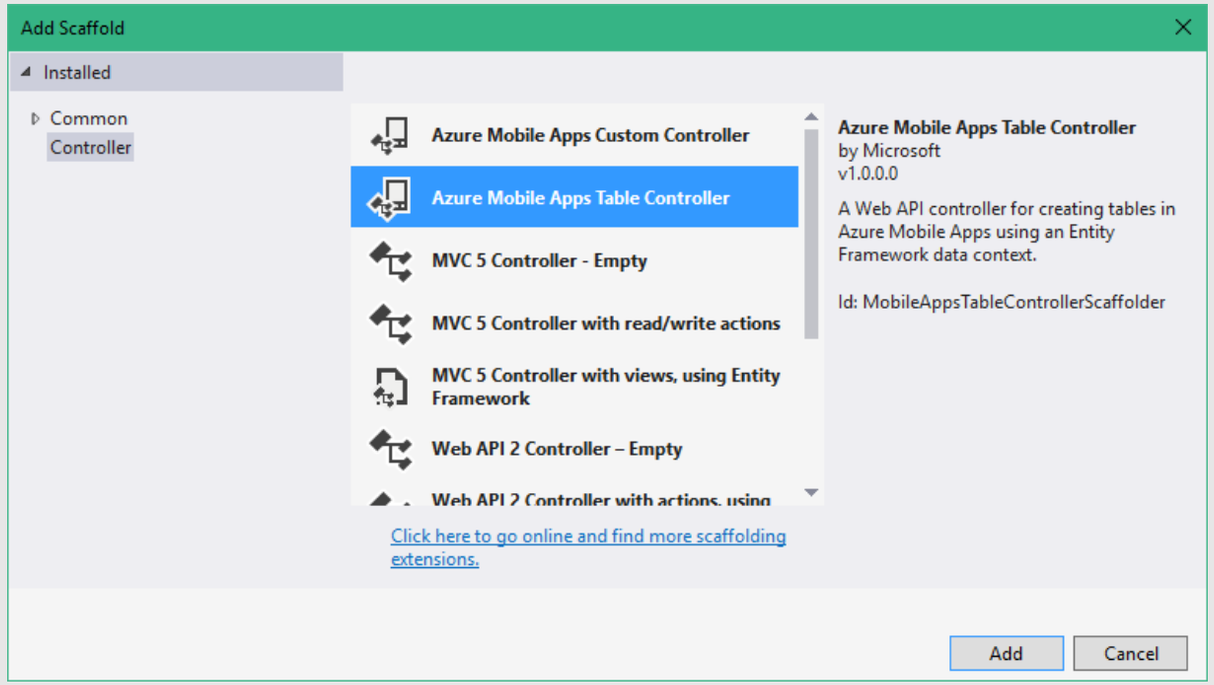
```
{
    "id": ...
     ...
    "complete": false,
    "todo": "My task"
},
```

You can use either the JSON.net attribute (as shown here), or the data annotation (**DataMember**) attribute to change the names of the passed JSON fields

Step 2: define the Table Controller

Must define a new controller to provide HTTP access to your table – easiest way to do this in VS is to use the **Add Scaffold** wizard





Select **Controller** from the **Add** menu and then select **Azure Mobile Apps Table Controller** from the dialog

What is a Table Controller?

Table Controller provides the REST endpoint + EF database connection for a single DTO through a set of methods

```
public class TodoItemController : TableController<TodoItem>
{
    protected override void Initialize(HttpControllerContext context) {...}
    // GET tables/TodoItem
    public IQueryable<TodoItem> GetAllTodoItems() {...}
    // GET tables/TodoItem/{id}
    public SingleResult<TodoItem> GetTodoItem(string id) {...}
    // PATCH tables/TodoItem/{id}
    public Task<TodoItem> PatchTodoItem(string id, Delta<TodoItem> patch) {...}
    // POST tables/TodoItem
    public async Task<IHttpActionResult> PostTodoItem(TodoItem item) {...}
    // DELETE tables/TodoItem/{id}
    public Task DeleteTodoItem(string id) {...}
}
```

Table Controller: initialize

Initialization method is responsible for creating the **domain manager** which maps and implements all the CRUD operations for the database and table used by the DTO

```
public class TodoItemController : TableController<TodoItem>
{
    protected override void Initialize(HttpControllerContext context)
    {
        base.Initialize(context);
        MobileServiceContext dbContext = new MobileServiceContext();
        DomainManager = new EntityDomainManager<TodoItem>(dbContext, Request);
    }
    ...
}
```

Table Controller: actions

Table controller exposes an **async method** for each supported HTTP verb and (by default) delegates work to base class methods

```
public class TodoItemController : TableController<TodoItem>
{
    public IQueryable<TodoItem> GetAllTodoItems() { return base.Query(); }

    public async Task<IHttpActionResult> PostTodoItem(TodoItem item) {
        TodoItem current = await base.InsertAsync(item);
        return base.CreatedAtRoute("Tables", new { id = current.Id }, current);
    }

    public Task DeleteTodoItem(string id) {
        return base.DeleteAsync(id);
    }
}
```

Customizing the method names

Method name prefix (Get/Post/Patch/Delete) is required to infer proper HTTP action; can use WebApi attributes to customize action/name

```
public class TodoItemController : TableController<TodoItem>
{
    [HttpGet]
    public IQueryable<TodoItem> RetrieveAll() {...}
    [HttpGet]
    public SingleResult<TodoItem> RetrieveOne(string id) {...}
    [HttpPatch]
    public Task<TodoItem> Update(string id, Delta<TodoItem> patch) {...}
    [HttpPost]
    public async Task<IHttpActionResult> Add(TodoItem item) {...}
    [HttpDelete]
    public Task Remove(string id) {...}
}
```

Exercise #11b

DTO, Controller, and Azure Table

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)

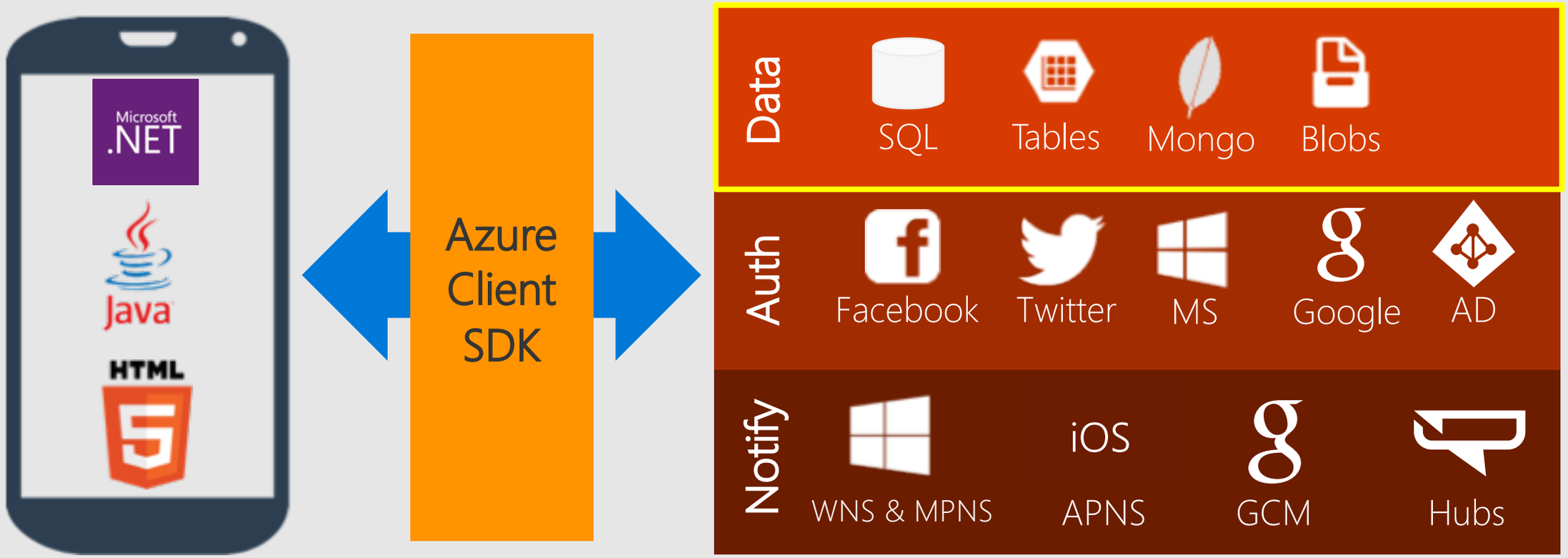
Data is communicated using JSON – can use **standard parsers** to serialize and de-serialize information

```
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



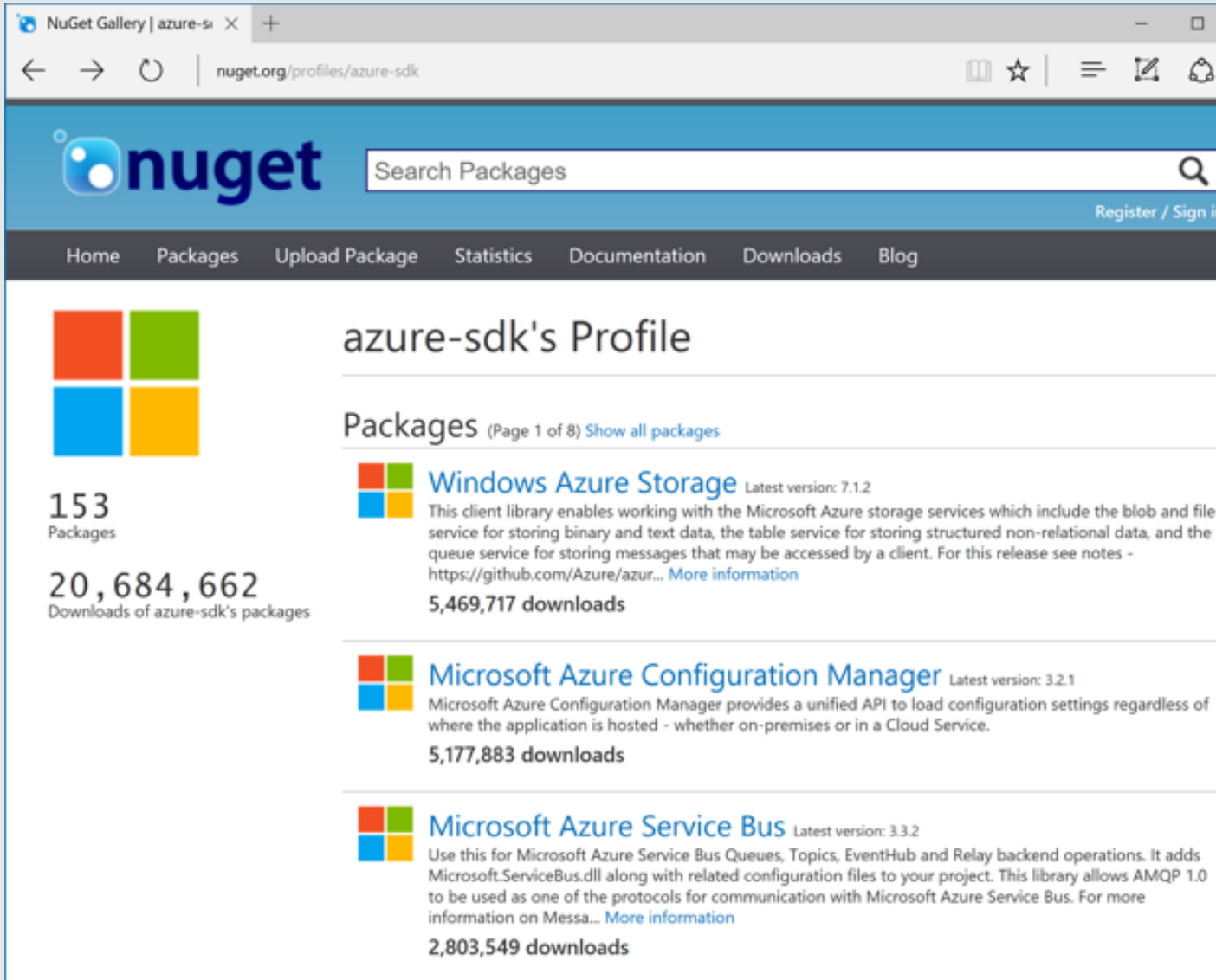
How to add the Azure client SDK

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

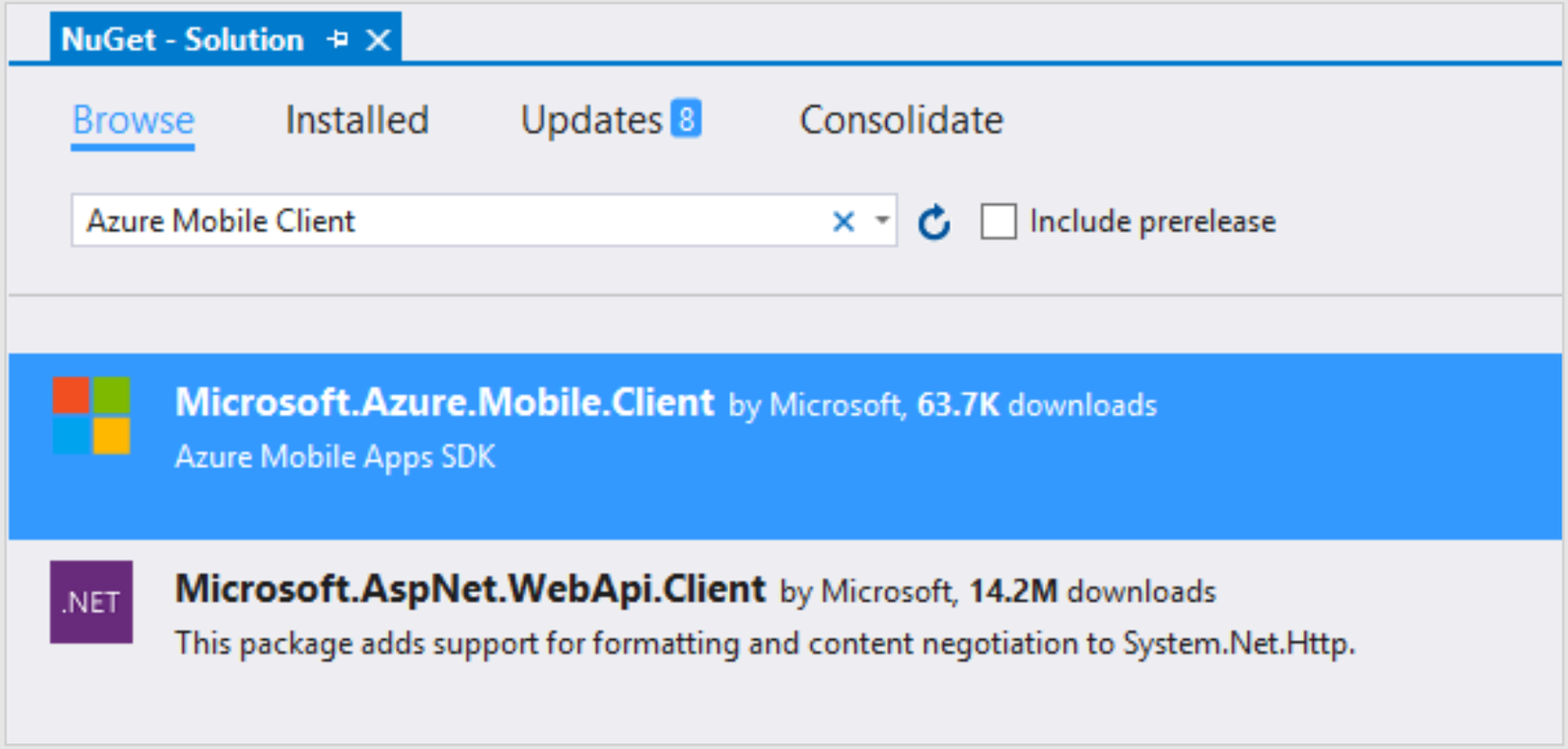
.NET and Xamarin applications can use pre-built client access libraries available from NuGet to access various Azure services

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



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



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

Exercise #12a

Add Azure Support To Our Application

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`**

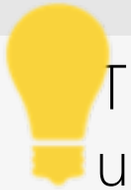
```
service = new MobileServiceClient("https://{site}.azurewebsites.net");
...
IMobileServiceTable table = service.GetTable("{tablename}");

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,
   ...
}
```

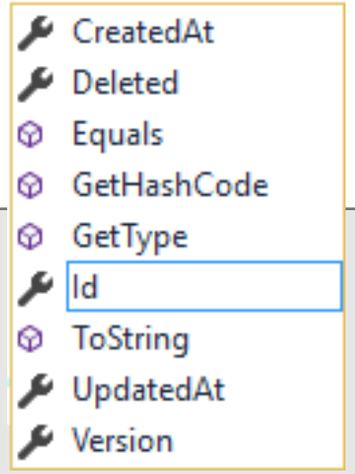

```
public class MyDTO
{
   public string Id { get; set; }
   public DateTimeOffset CreatedAt { get; set; }
   public DateTimeOffset UpdatedAt { get; set; }
   public string Version { get; set; }
   public bool Deleted { get; set; }
   ...
}
```

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
{
    ...
}
```

MobileServiceClient

```
GET /tables/DiaryEntry
```

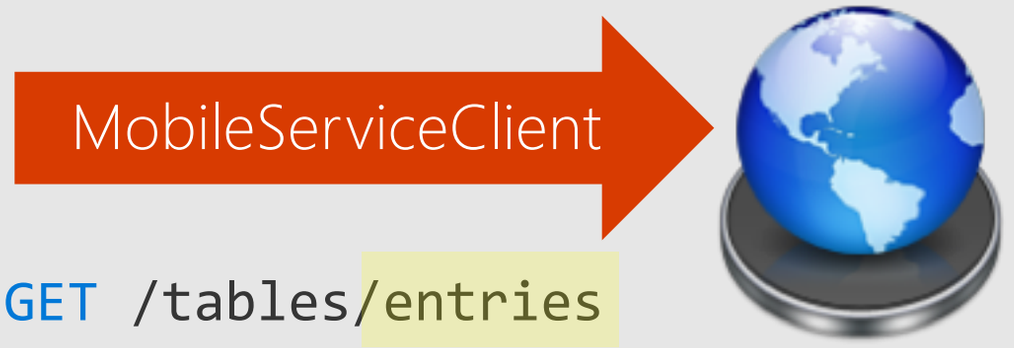
What if the server endpoint is **entries**?
Result is a **404** (Not Found) error!

Identifying the server side table

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

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

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



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
  }
};
```

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);
```

Exercise #12b

Client Side DTO and get Question Data

Summary

- Azure App Services is a reliable, scalable solution to add a back end to your mobile app.
- Adding a SQL Database to your back end in Azure gives you all the required pieces for easy offline sync.
- The Azure Client SDK makes accessing your Azure Tables a snap.