AZR110

Building an Azure Mobile App Service

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

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

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

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

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

Objectives

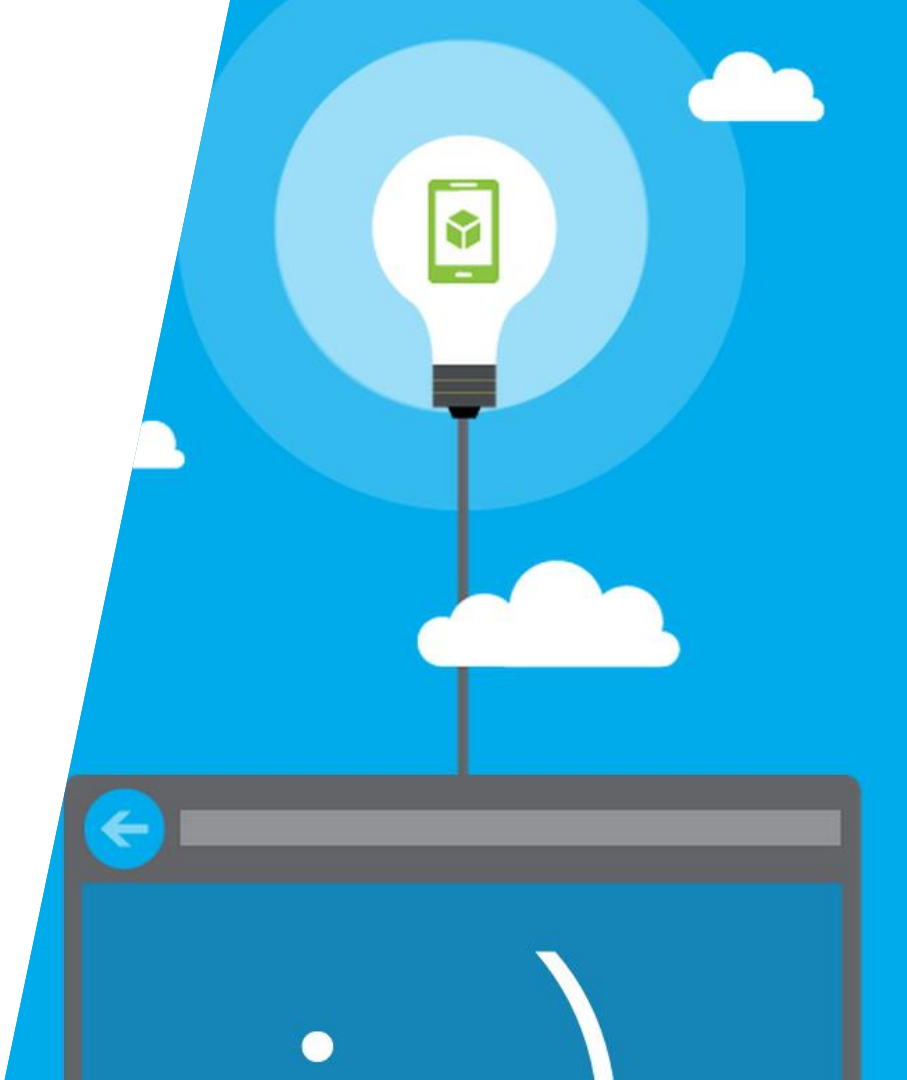
1. Create the mobile app service
2. Add a database to your back end

Create the mobile app service

Xamarin University

Tasks

1. Explore the mobile features provided by Azure App Services
2. Create a new Mobile app in the Azure Management portal
3. Create a new Mobile app in Visual Studio
4. Setup deployment publishing



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

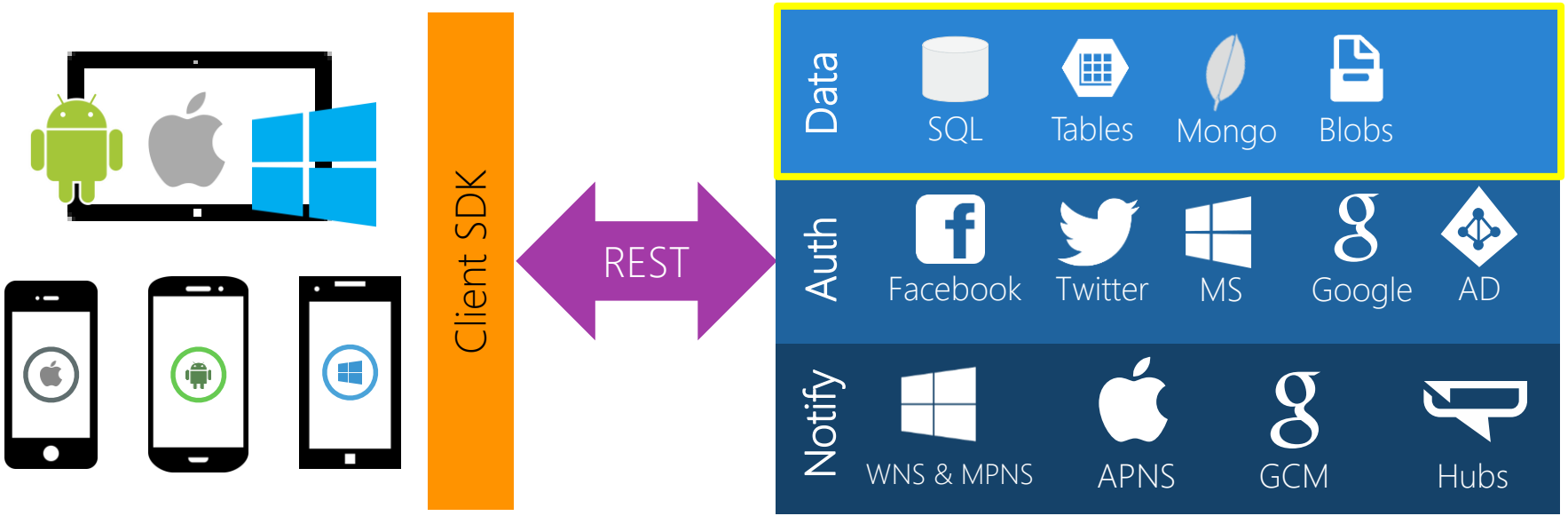
API App
Used to create a hosted RESTful web service

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



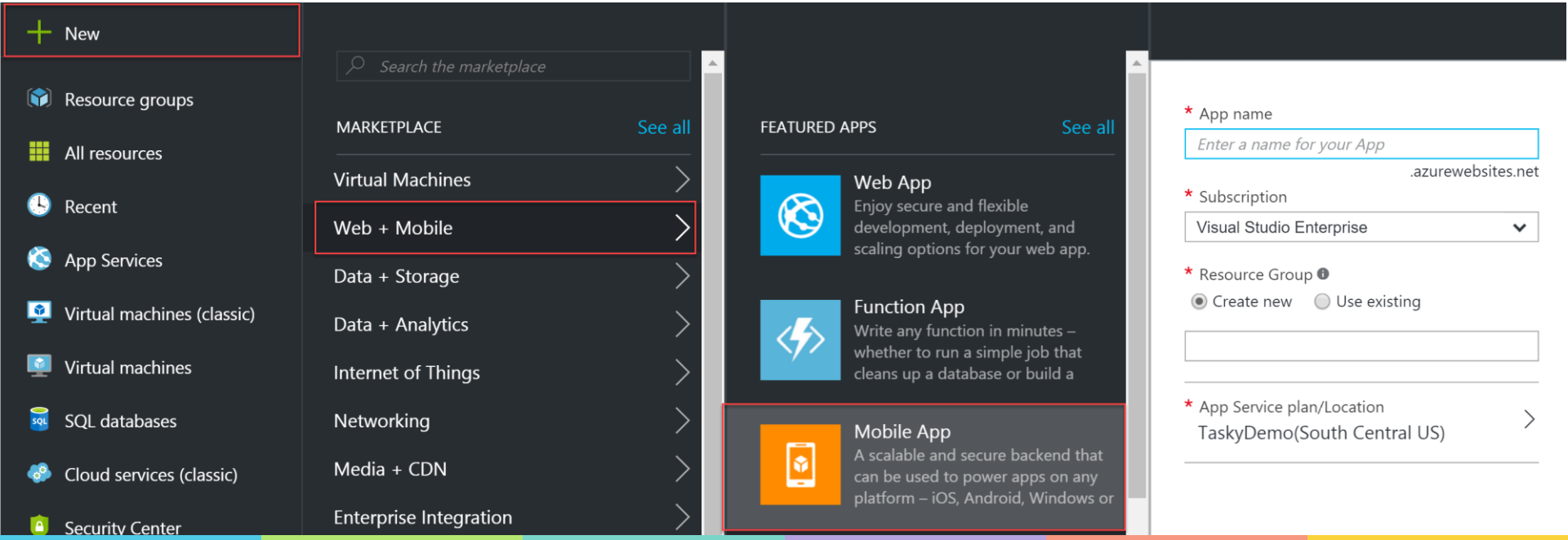
Implementing the back-end service

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

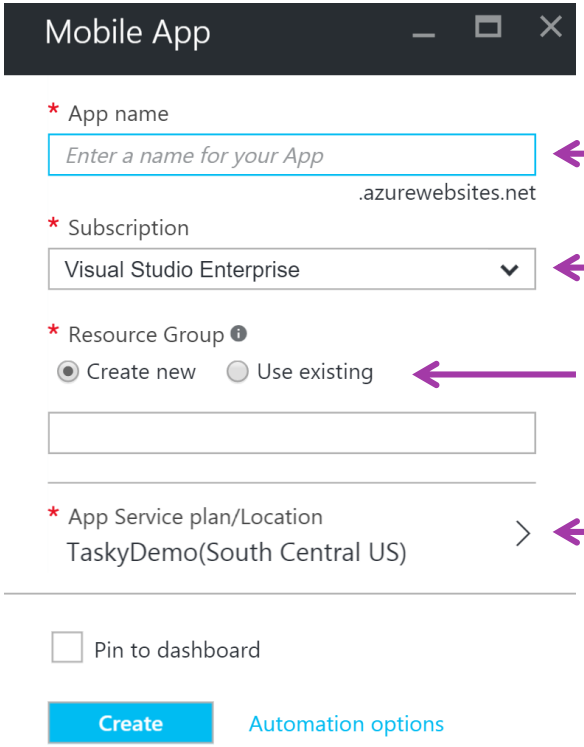
Uses JavaScript to define and customize server endpoints and point-and-click creation but lacks type safety - This is the default when you use the portal to create your app

Recall: creating an app in the portal

- Create a new Mobile App using the template under **Web + Mobile** in the management portal (https://portal.azure.com)



Defining the app



Must supply a **globally unique name** to create the public URL

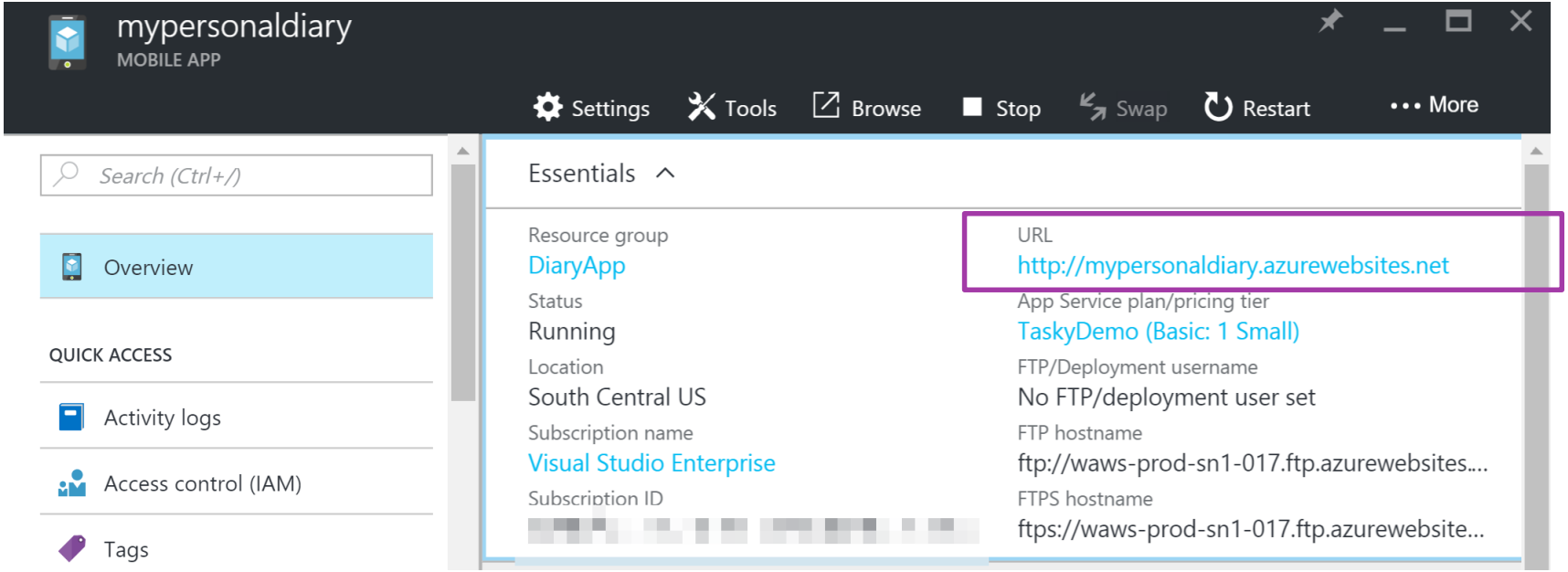
Select Azure billing subscription to use

Should create a new resource group to tie all the app elements together; can also use an existing group

Must select the service tier and regional location where service will be hosted; can change service plan as you shift from development > production

Getting the mobile URL

- Once created, the mobile app dashboard will display URL information needed to connect the mobile app to the service



Creating a node.js back end with CLI

- Can also generate a **node.js** back end on your *local machine* from the command line using Yeoman and then publish to Azure

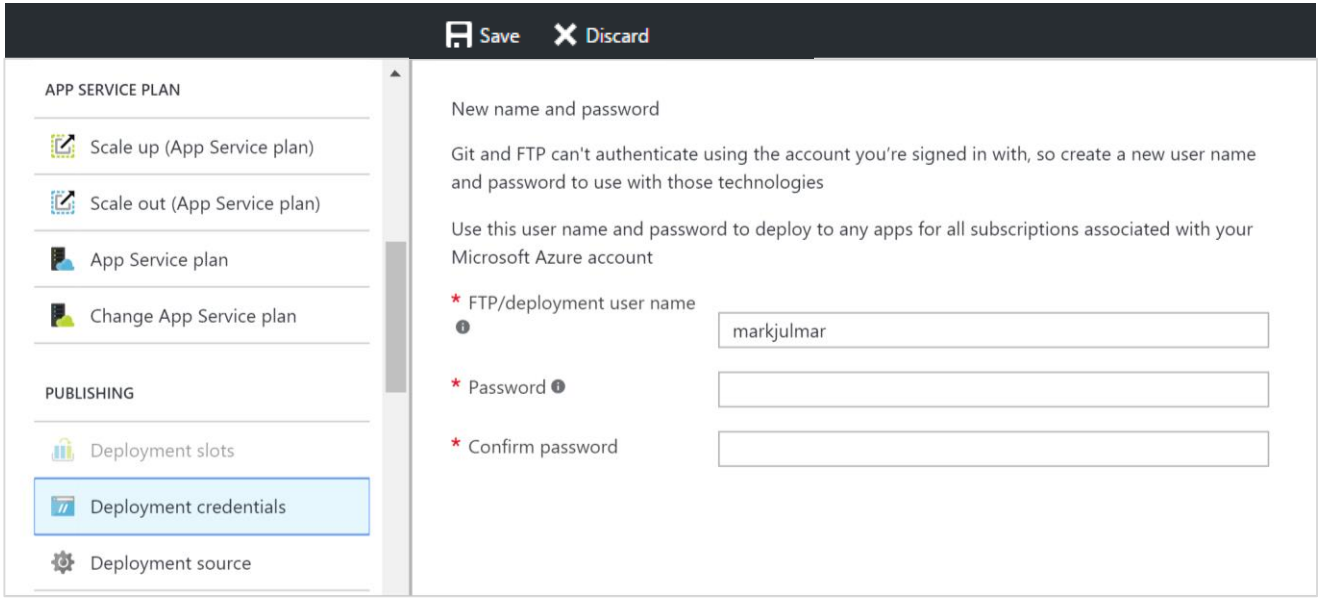
Install the generator (macOS / Linux / WinBash)

```
~ sudo npm install -g yo generator-azure-mobile-apps
...
~ yo azure-mobile-apps
```

Generate a new Azure mobile app in the current folder – can then upload into Azure using FTP or publishing deployment

Manually publishing to Azure (FTP)

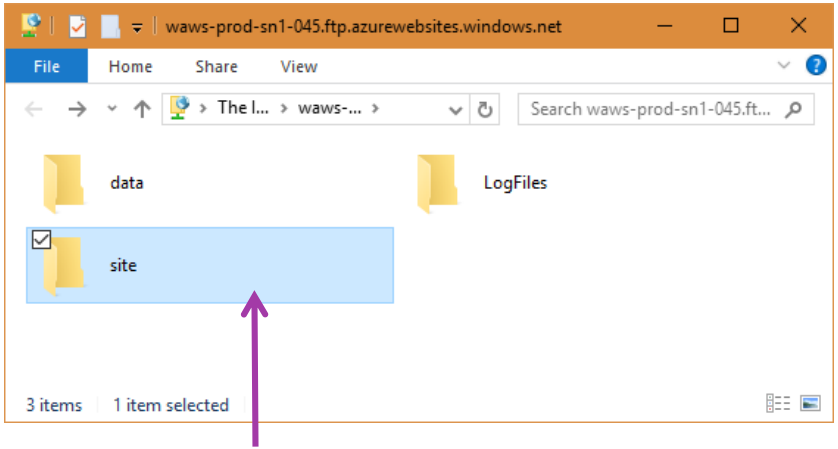
- Can manually publish service using FTP; credentials can be set in the **Publishing** settings of your app portal and URL is available from dashboard



Manually publishing to Azure (FTP)

- Can manually publish service using FTP; credentials can be set in the **Publishing** settings of your app portal and URL is available from dashboard

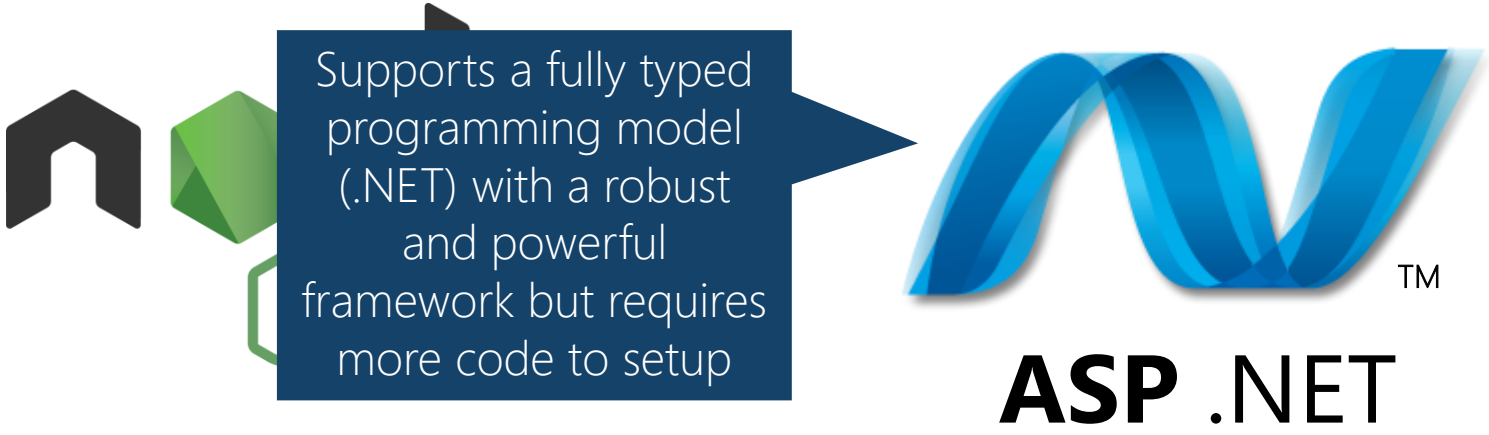
URL
http://taskydemo.azurewebsites.net
App Service plan/pricing tier
TaskyDemo (Basic: 1 Small)
FTP/Deployment username
TaskyDemo\markjulmar
FTP hostname
ftp://waws-prod-sn1-017.ftp.azurewebsites....
FTPS hostname
ftps://waws-prod-sn1-017.ftp.azurewebsite...



Web server files must be placed into the **/site/wwwroot** folder

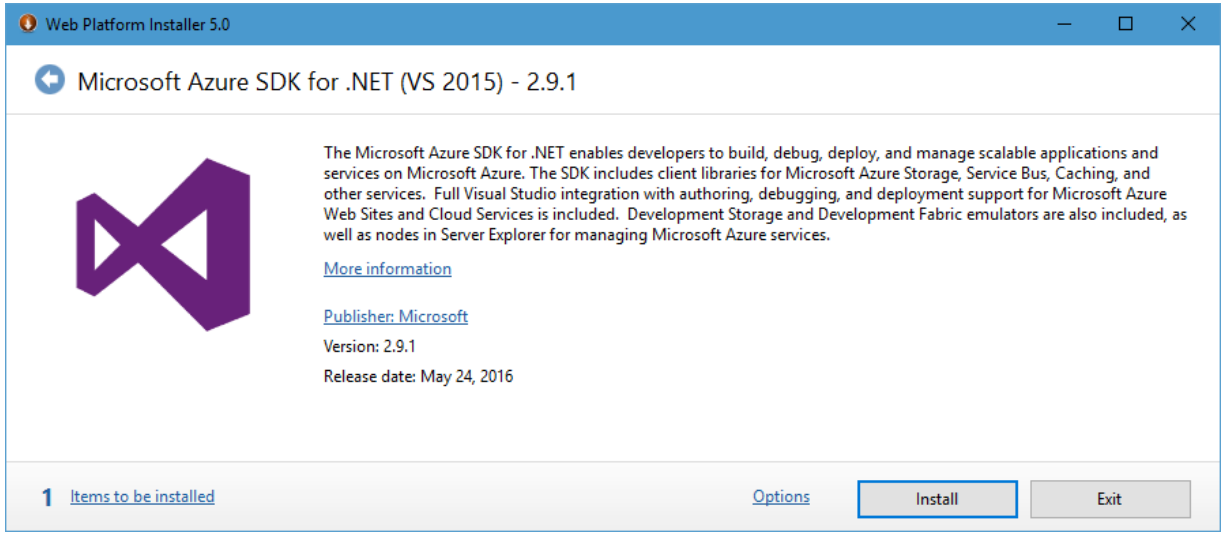
Implementing the back-end service

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



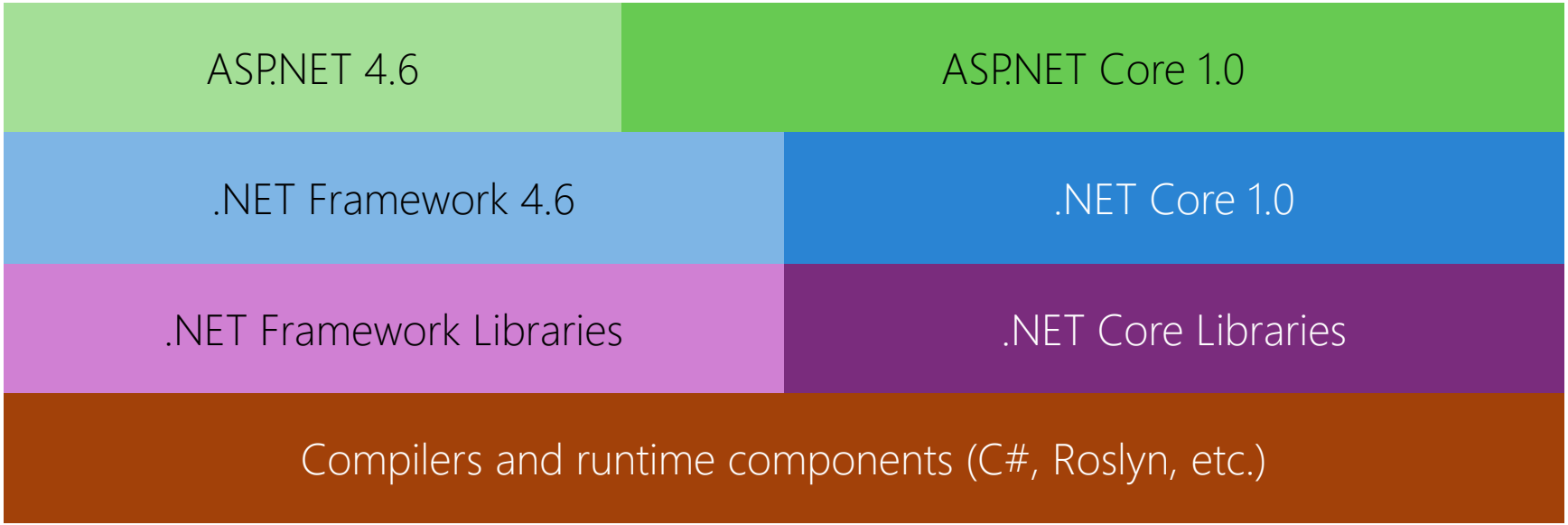
Install the Azure SDK for VS

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



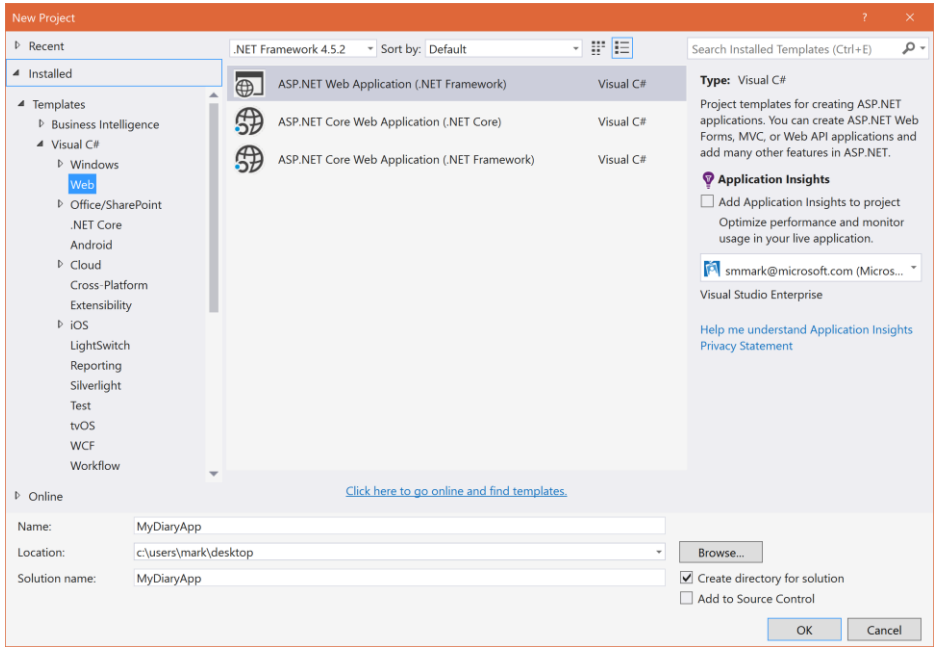
.NET Platforms

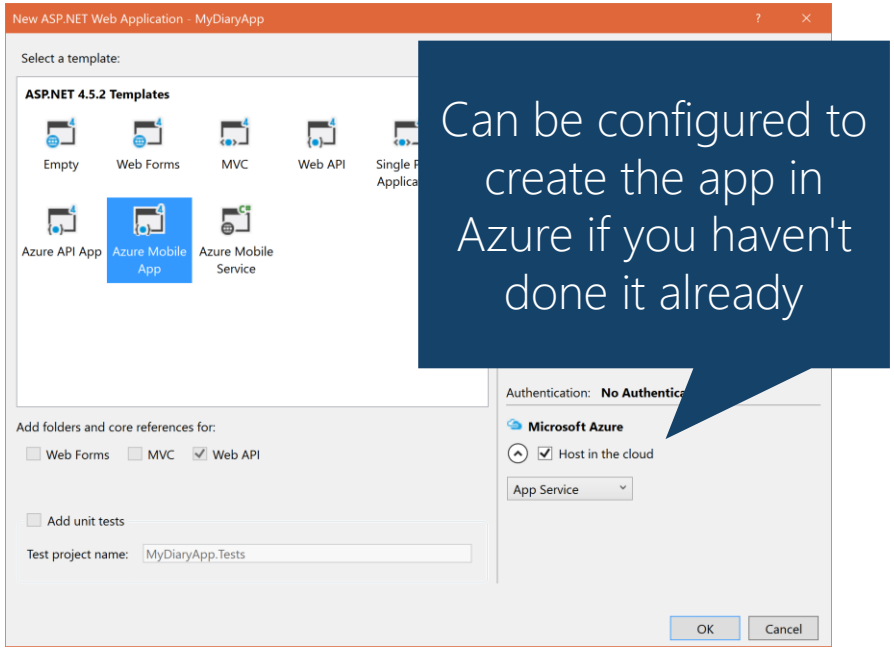
- Service back end can be built on top of traditional ASP.NET or on the newer **ASP.NET Core** platform



Creating an Azure App service in .NET

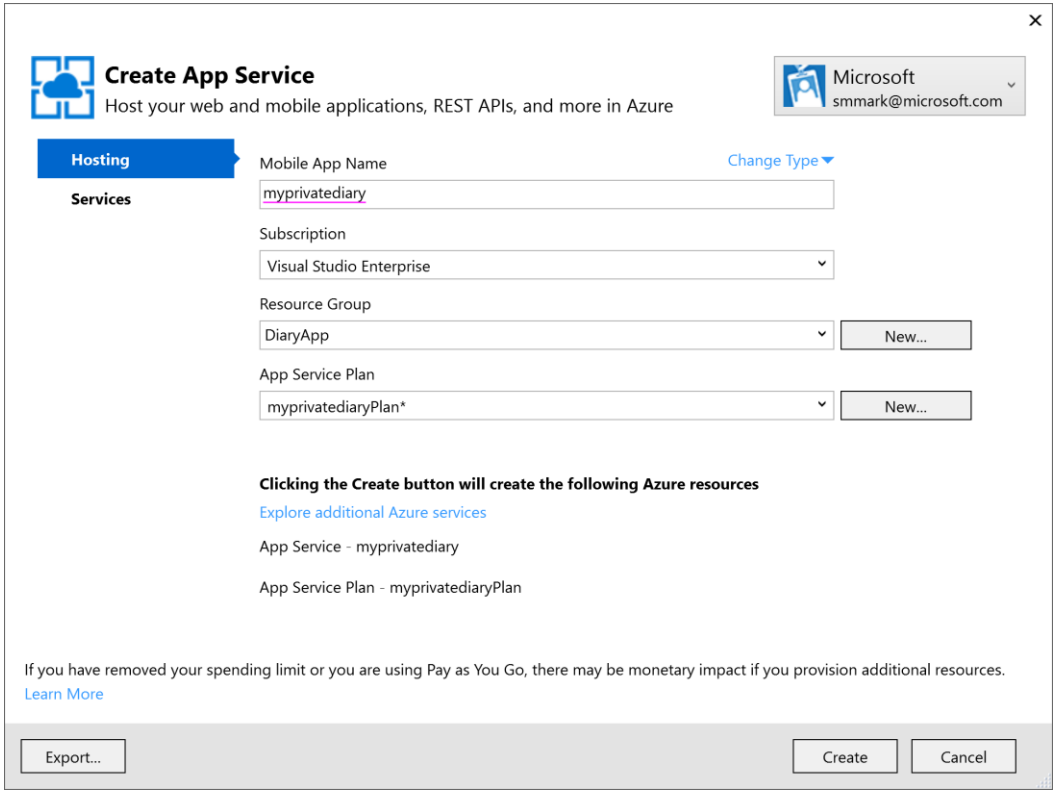
- Use the **Azure Mobile App** template to create a web service in VS





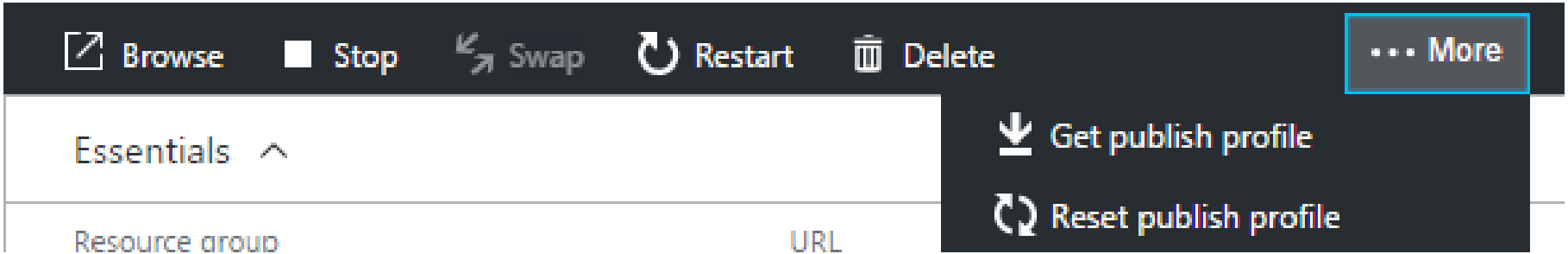
Defining the Azure app in VS

- 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



Resetting the publishing credentials

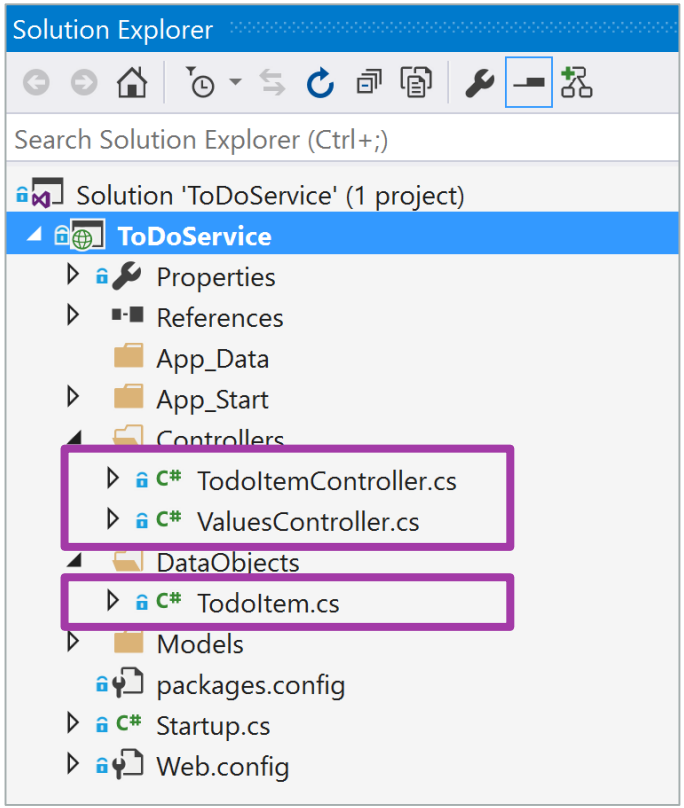
- Can download or reset the publishing profile used by WebDeploy through the **More** menu in the app portal



Beware: passwords in the downloaded publishing profile are stored in clear text!

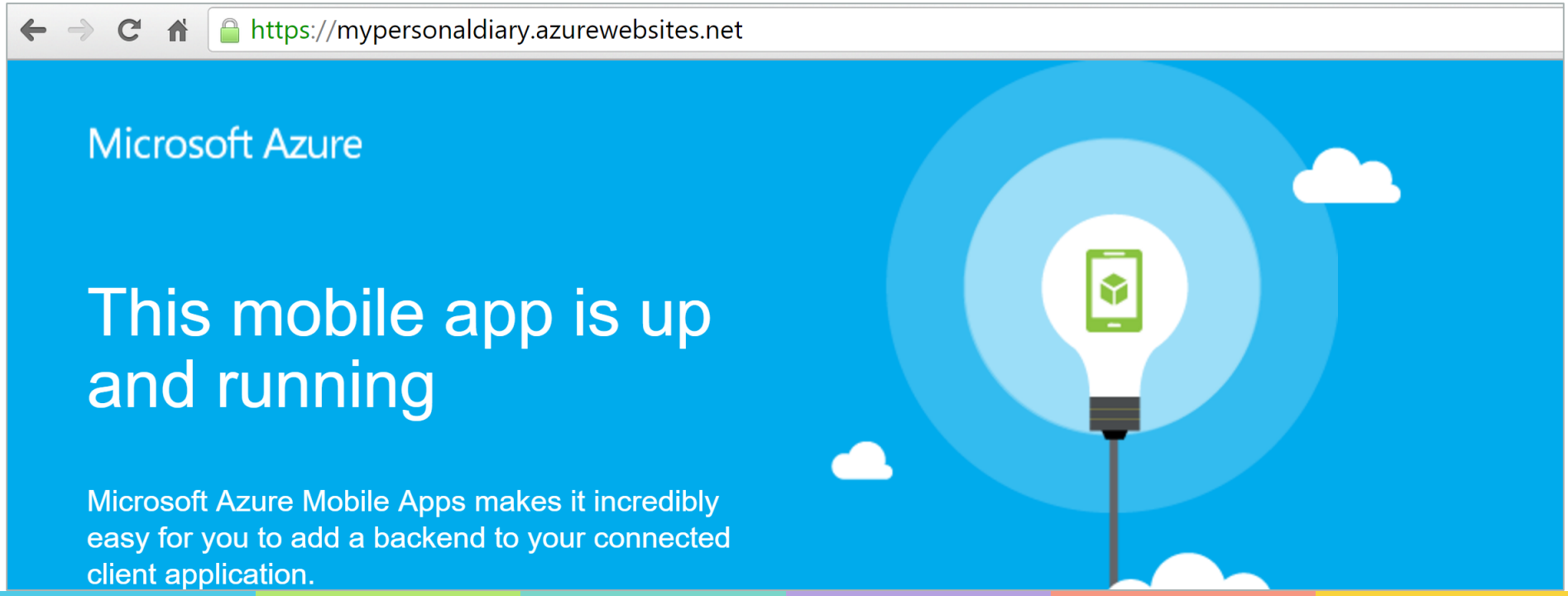
Visual Studio server project

- 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



Individual Exercise

Create a survey app service in Azure

Publishing to Azure

- Azure supports several publishing models to suite any project size

Manual push from client

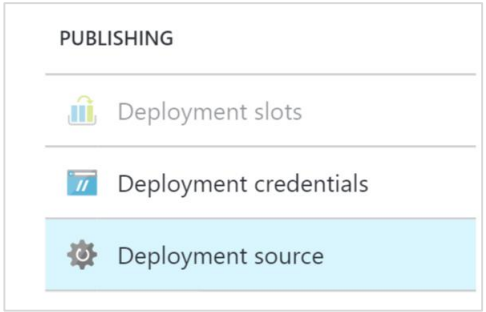
Manual push from VS

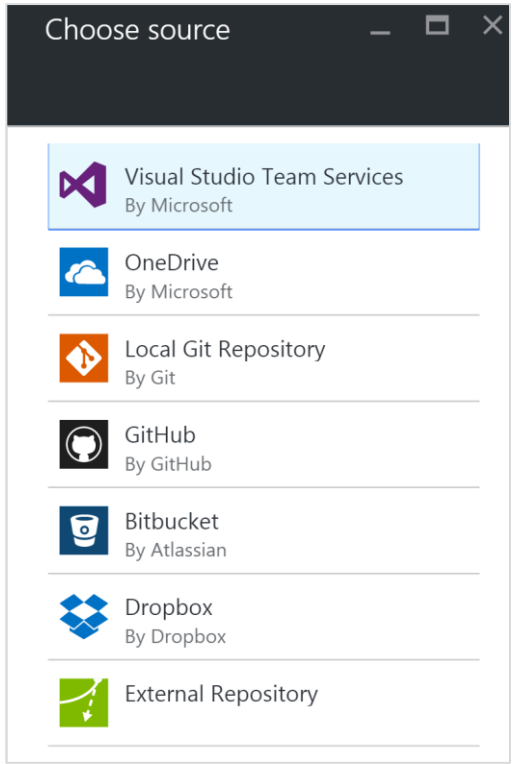
Manual push from cloud or local Git
(OneDrive/Dropbox or Git/Mercurial)

Continuous Deployment
(TFS, Github or BitBucket)

Setting up a deployment source

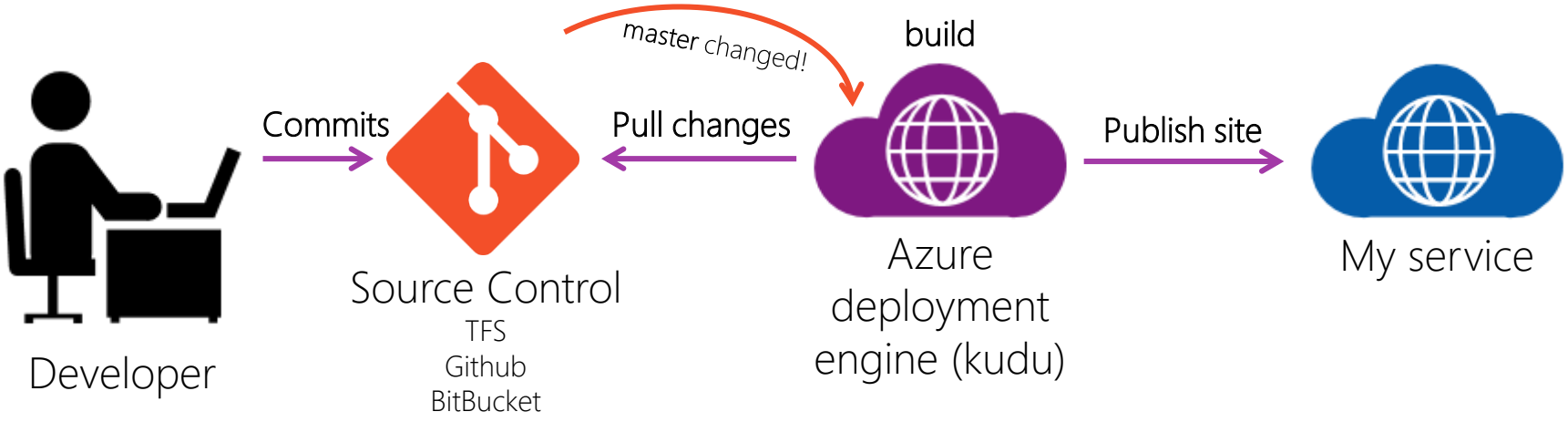
- Can setup a single publishing source through the **deployment source** option in your app publishing properties





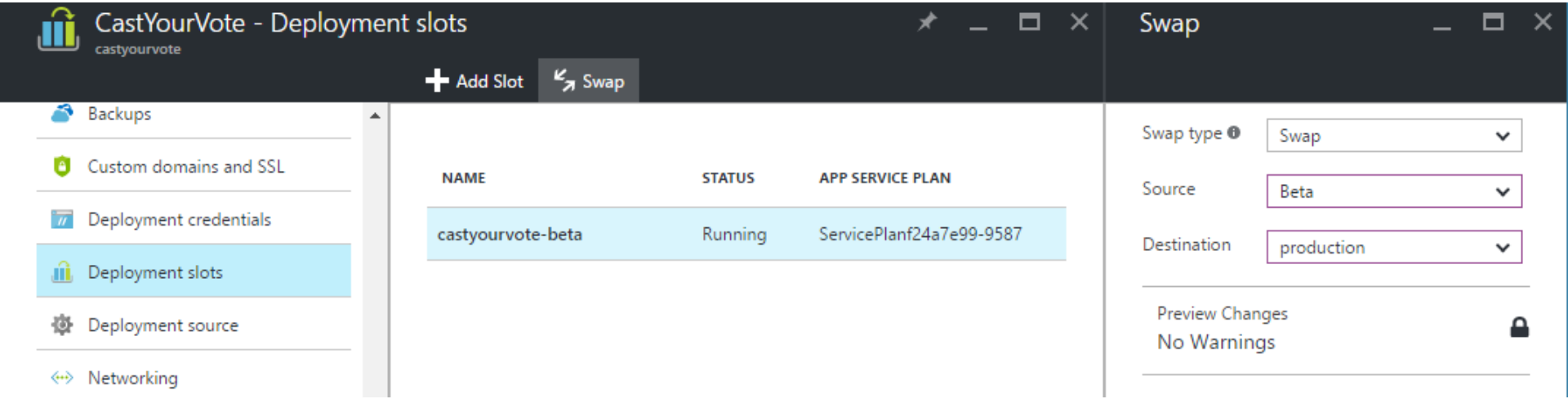
Continuous development

- Can configure Azure size to **automatically pull changes** from TFS, Github or BitBucket when a new commit is detected



Configuring zero downtime

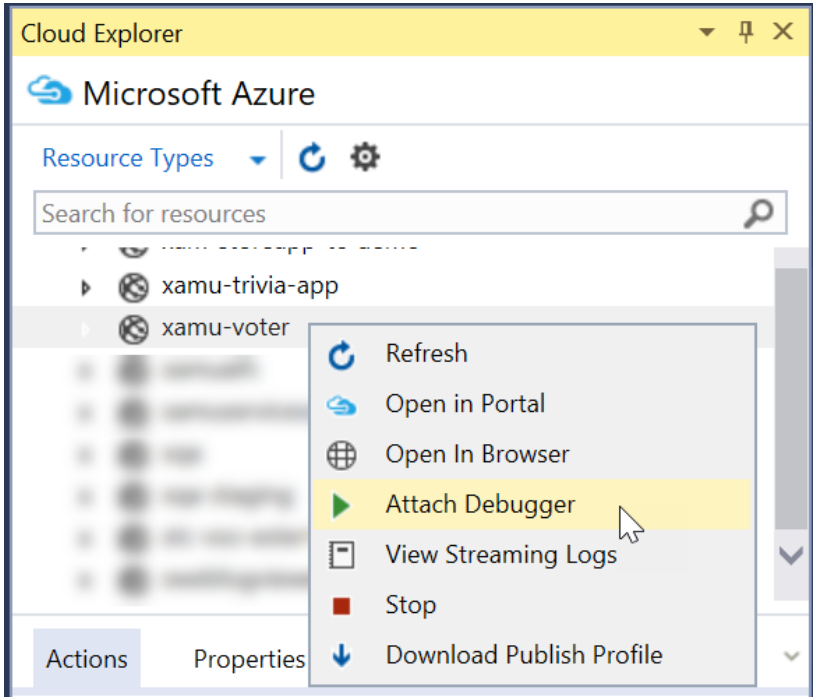
- Depending on your service plan, you can define *deployment slots* which let you create *copies* of your site on unique URLs and then swap them into production with zero down time



Debugging your app service

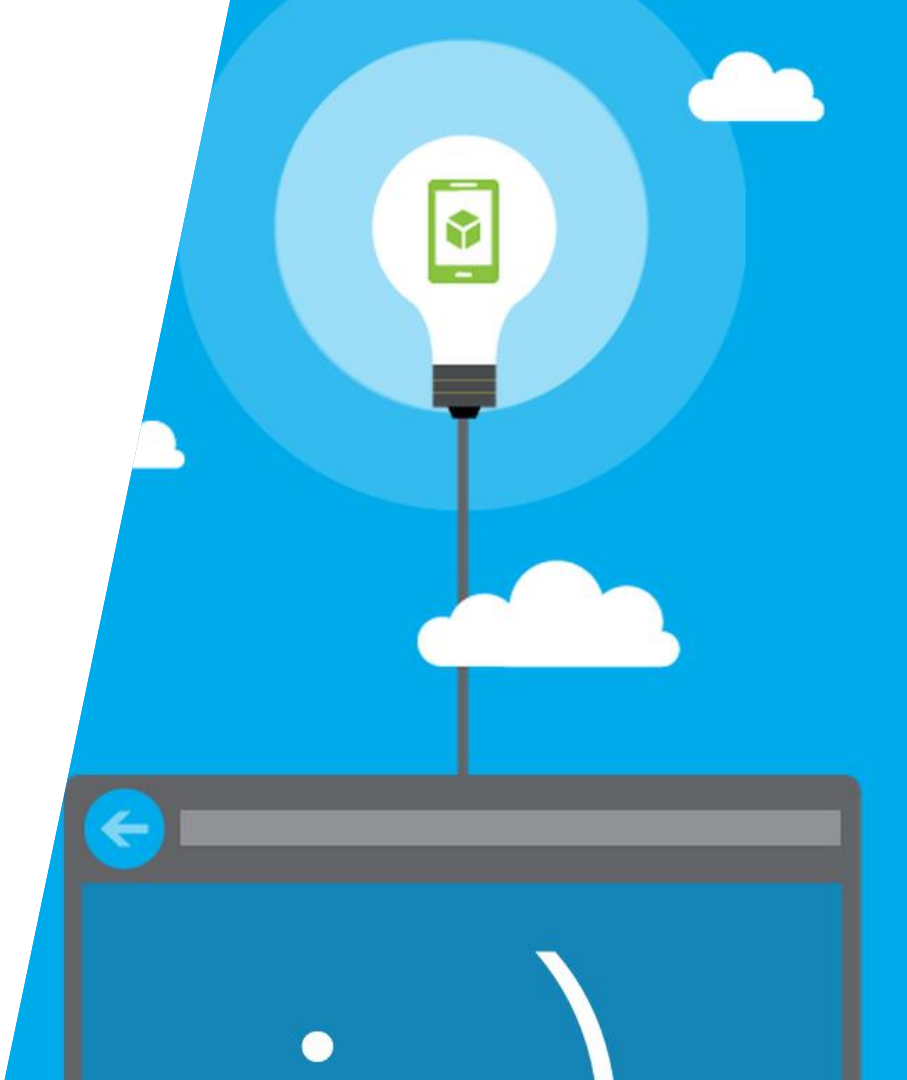
- When using the ASP.NET back end, you can debug your service code through the **Cloud Explorer** pane; right-click on the Web App and select **Attach Debugger**

- Must publish a **DEBUG** build first to get symbolic information – otherwise breakpoints won't resolve



Summary

1. Explore the mobile features provided by Azure App Services
2. Create a new Mobile app in the Azure Management portal
3. Create a new Mobile app in Visual Studio
4. Setup deployment publishing



Add a database to your back end

Tasks

1. Decide the proper type of database to add
2. Create the database + connection
3. Create one or more tables
4. Populate the database (optional)

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)

Blob Storage
(unstructured files)

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

Azure data styles

- Azure provides several managed storage choices for apps

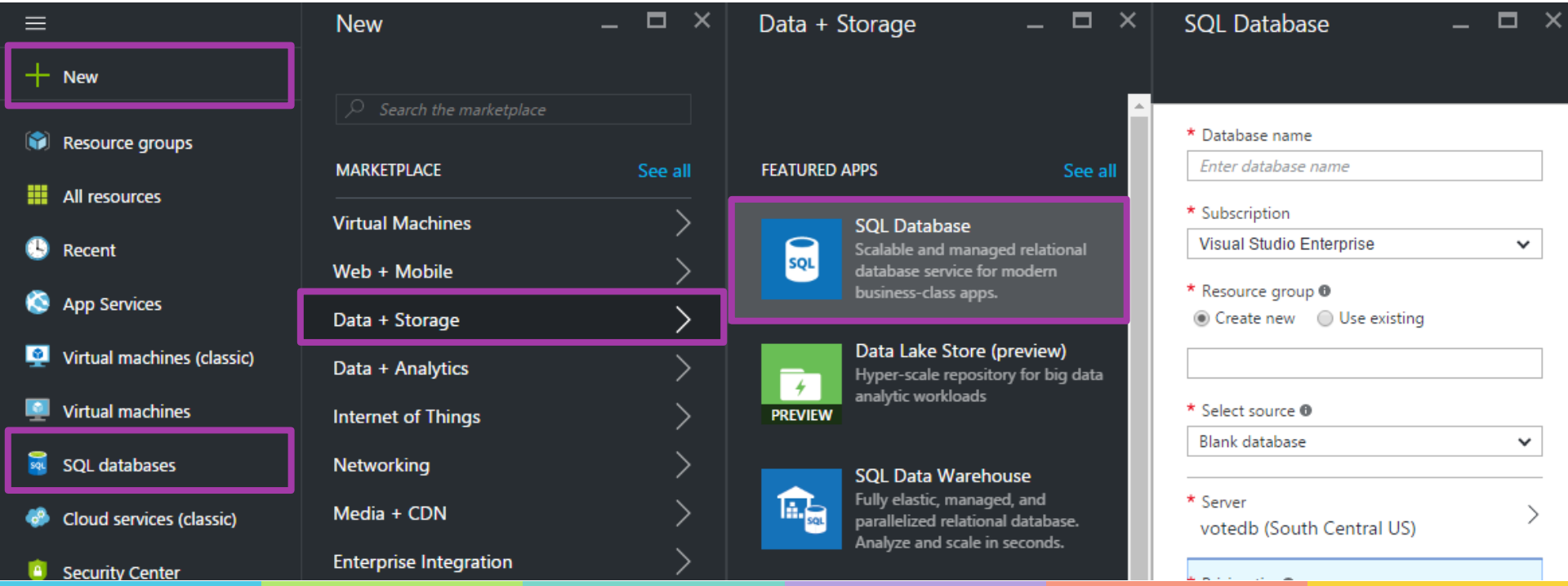
SQL Database
(relational)

Table Storage
(NoSQL key/value)

Blob Storage
(unstructured files)

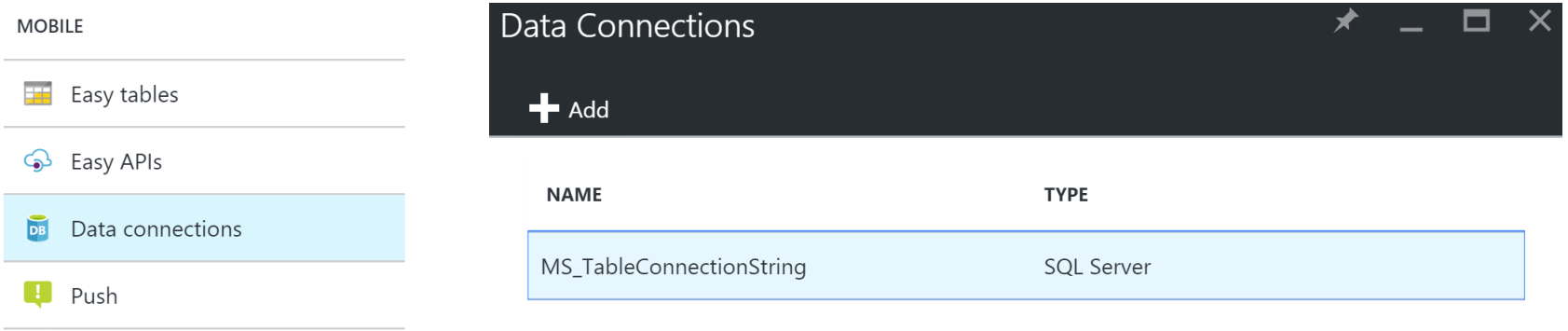
Creating a SQL database in Azure

- Must have a SQL Server database resource in your Azure portal



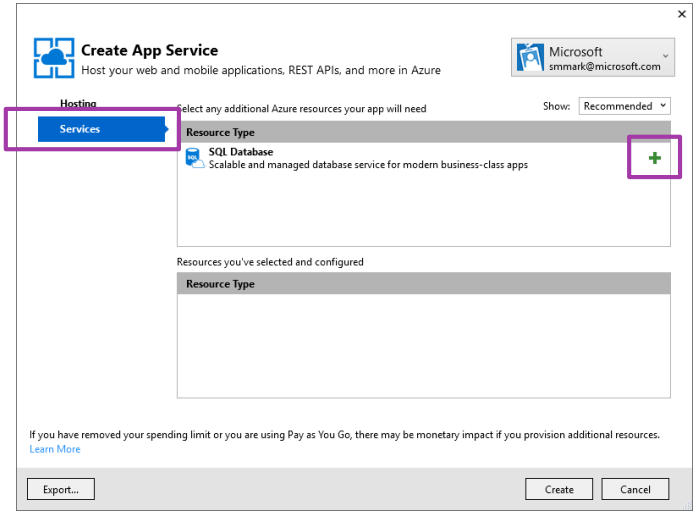
Adding a SQL database to your app

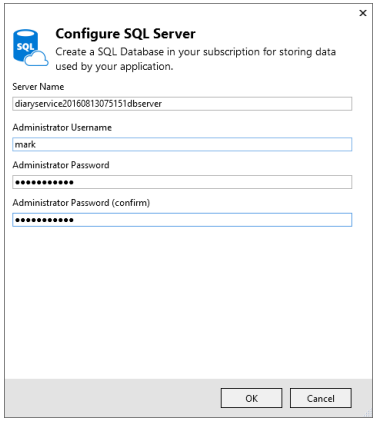
- Must add a connection string to the app service; default expected name is **MS_TableConnectionString**



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





Individual Exercise

Add a database to your Survey service

Xamarin University

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

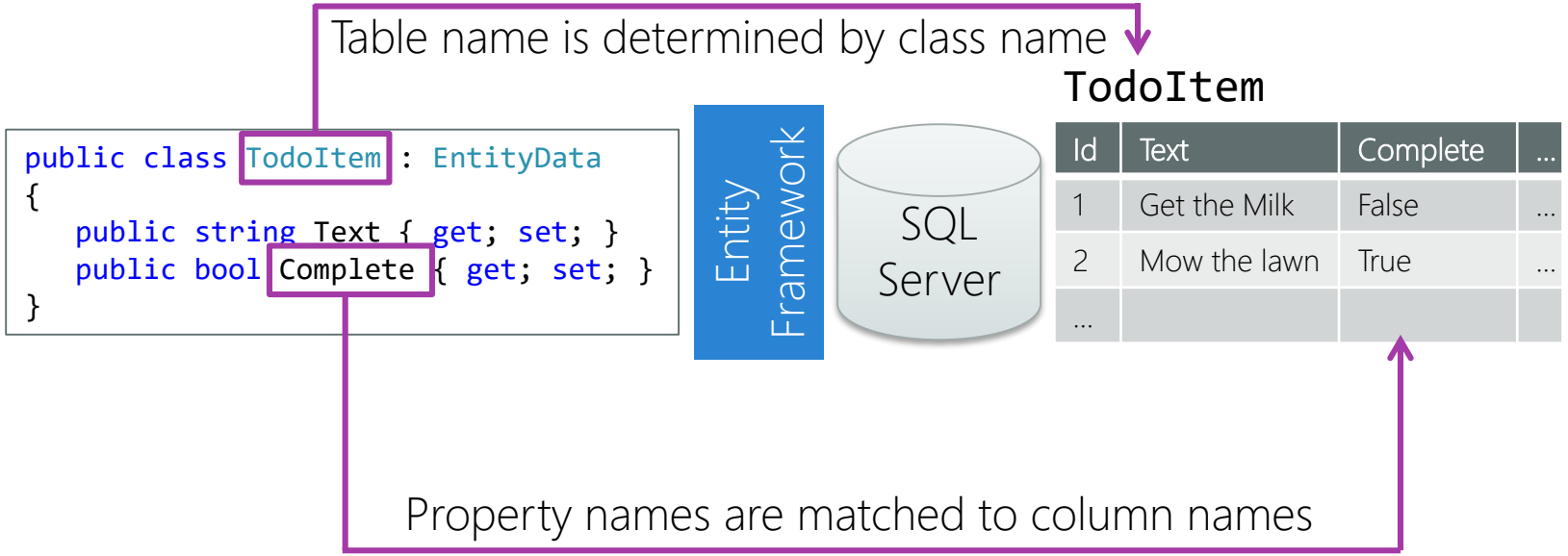
What is EntityData?

- **`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

Use **tasks** table

Specify the table column name

Ignore property

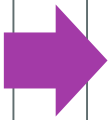
Add an index for this column

```
[Table("tasks")]
public class TodoItem : EntityData
{
    public string Text { get; set; }
    [Column("is_complete"), Index]
    public bool Complete { get; set; }
    [NotMapped]
    public bool Tagged { get; set; }
}
```

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; }
}
```

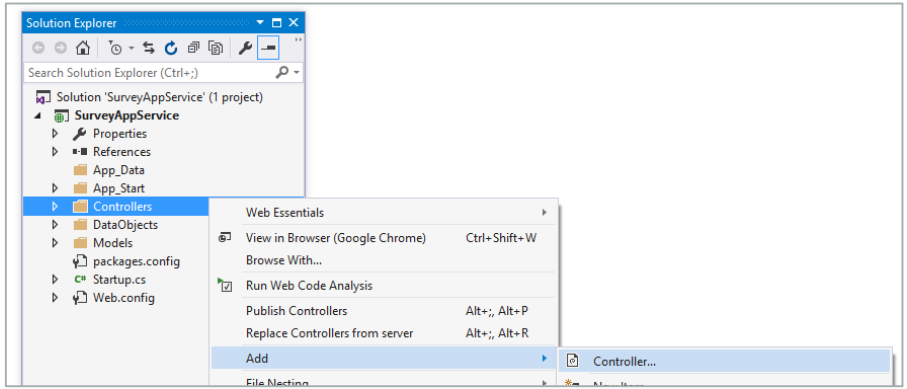


```
{
    "id": ...
     ...
    "complete": false,
    "todo": "My task"
},
```

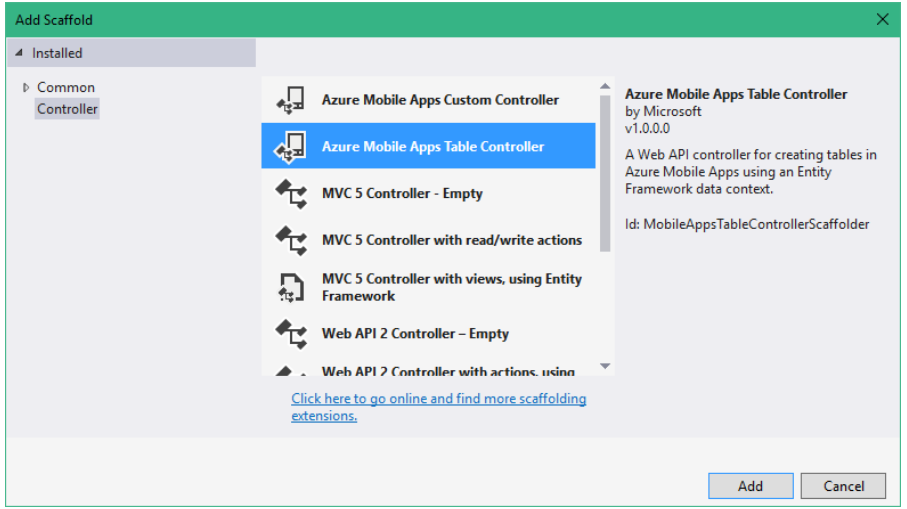

You can use either the JSON.net attribute (as shown here), or the data annotation (**DataMember**) attribute to change the names of the passed JSON fields

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);
    }
}
```

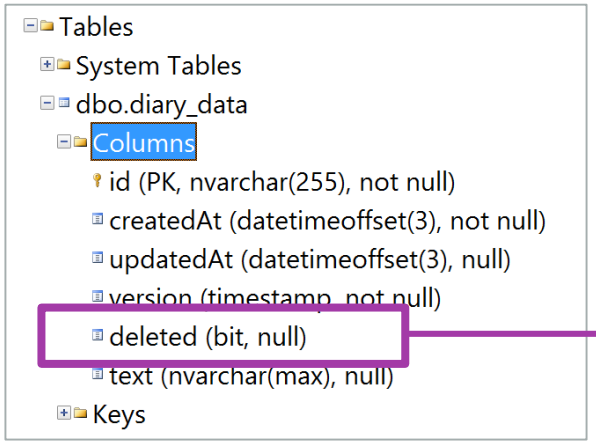
Customizing the method names

- 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) {...}
}
```

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



column is always present regardless of whether the server enables soft delete or not

Do I need soft delete?

- Soft delete means that records are **never deleted from the table**; this has benefits and drawbacks which you should weigh to decide whether you want this feature

Pros	Cons
Simplifies offline synchronization	Databases tend to require more space
Allows records to be "undeleted"	Id must be a string type and not reused
Useful for audit or requirements where records cannot be removed	Must write a server-side Azure Function or SQL trigger to periodically purge records

ASP.NET: turning on soft delete

- Must enable soft delete for each table through the **`EntityDomainManager`** constructor in your table controller initialization

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

Defining a custom API controller

- Can expose traditional REST endpoints with ASP.NET Web API by deriving from the **`ApiController`** base class

```
[MobileAppController]
public class HelloController : ApiController
{
    [HttpGet]
    public string SayHello() {
        return "Hello, Azure!";
    }
    ...
}
```

Should decorate with **`[MobileAppController]`** attribute to integrate with Azure service platform

Flash Quiz

Xamarin University

Flash Quiz

① SQL tables can be exposed from either **node.js** or **ASP.NET** back ends

a) True

b) False

Flash Quiz

① SQL tables can be exposed from either **node.js** or **ASP.NET** back ends

a) True

b) False

Flash Quiz

② You can define the schema for exposed tables through the Azure portal when using an ASP.NET back end

a) True

b) False

Flash Quiz

② You can define the schema for exposed tables through the Azure portal when using an ASP.NET back end

a) True

b) False

Adding a table to a node.js back end

- **Node.js** back end provides "no-code" web access to SQL data through the **easy tables** API which has several key benefits

Uses SQL Azure as the database storage

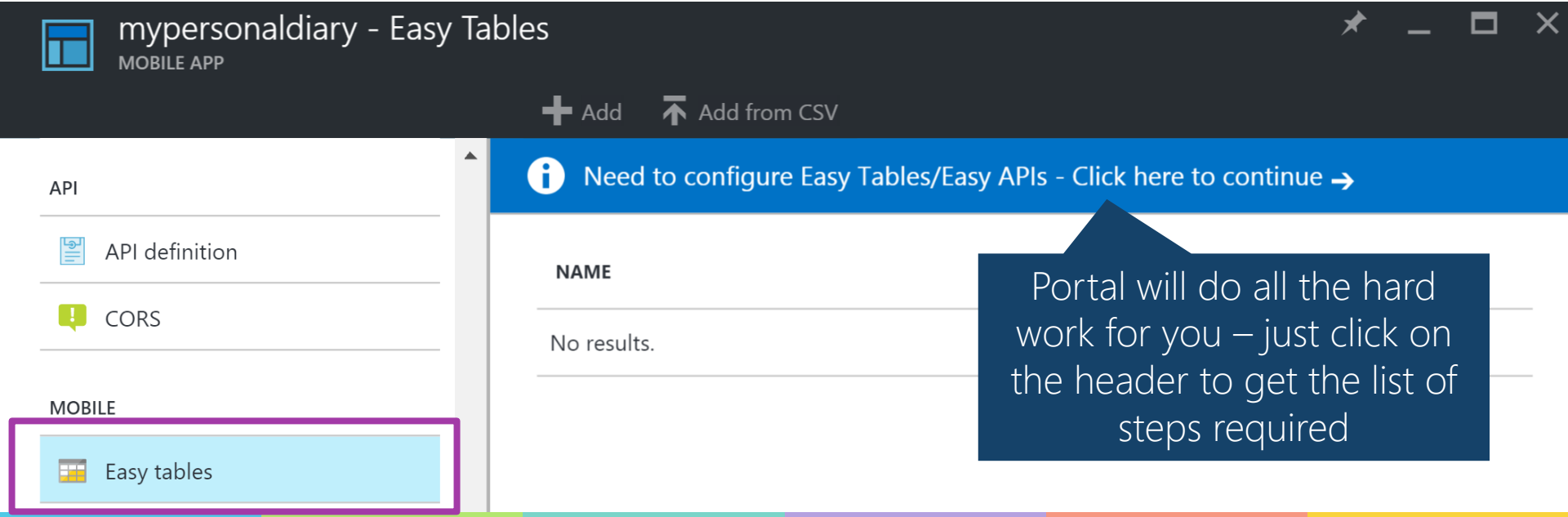
Exposes OData endpoint with no additional code required

Can define and change DB schema in Azure portal

Supports server-side logic for database operations

How to configure Easy tables API

- Easy tables must be configured for your mobile app in the Azure portal to expose the endpoint the app will use to communicate with the database



Setting up Easy table support

- Two required steps to turn on the easy tables support in your app service, portal will walk you through both as part of the setup

1. Create or select the SQL database and the connection string
2. App service must be configured to use easy tables – this is a one-time operation that configures the web server

Creating Easy tables

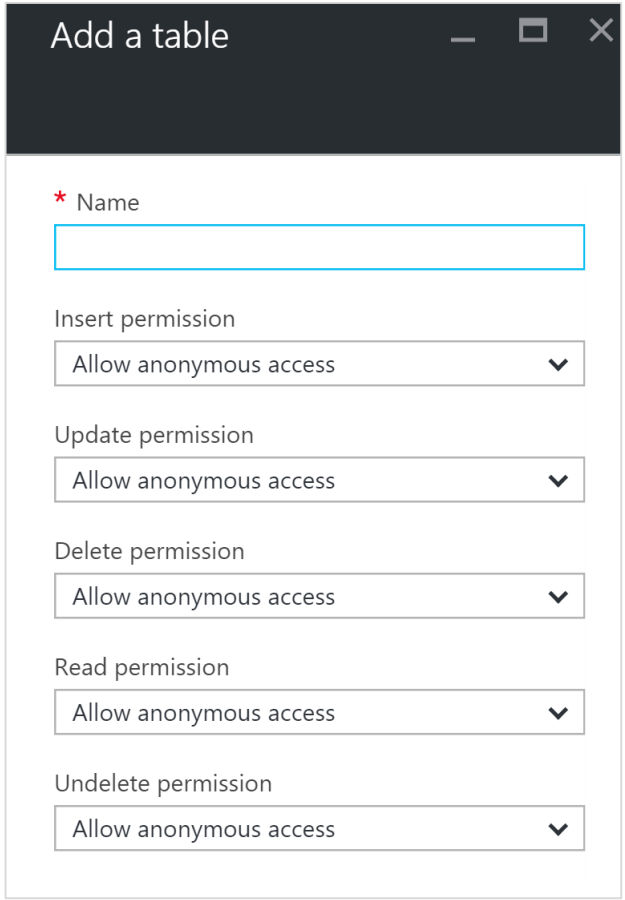
- Once a SQL database connection is setup, you can add one or more Easy table definitions to the database using two approaches

Azure Portal

Web server configuration

Using the Azure portal

- Can use the Azure portal to add new tables to your associated SQL database – this is the easiest option since it's GUI driven
- Must provide a locally-unique table name and define the permissions for CRUD operations (defaults to anonymous)



Schema for an Easy table

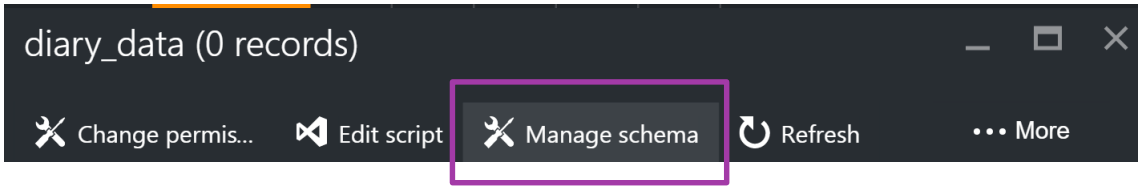
- Just like ASP.NET, each table has 5 required columns to identify each row and support offline synchronization

Column	SQL Type	Description
`id`	`NVARCHAR(255)`	Unique identifier for this record (typically GUID)
`createdAt`	`DATETIMEOFFSET`	Date/time that this record was initially added
`updatedAt`	`DATETIMEOFFSET`	Last date/time that this record was changed
`version`	`TIMESTAMP`	Version of this record, used for synchronization
`deleted`	`BIT`	Set if this record has been deleted, used for sync.

You can add these columns to an existing SQL table to allow it to be used with the API – see http://bit.ly/2aANOTz for more information on the necessary steps

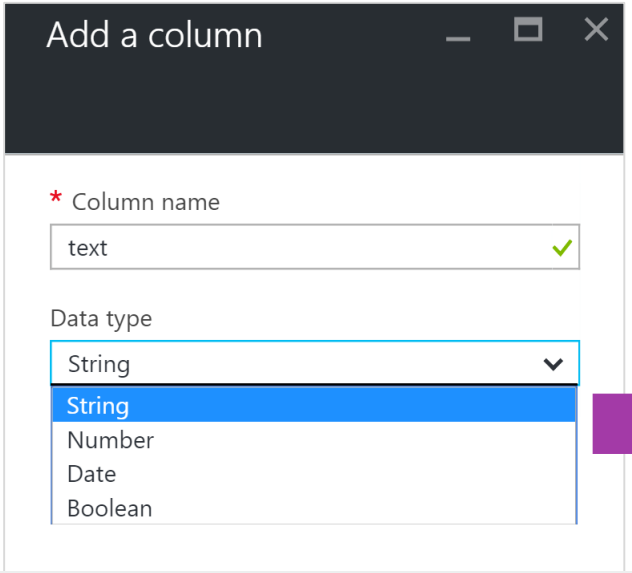
Creating unique data columns

- You must define additional columns using the Easy Tables blade to store your app-specific data, use the **Manage schema** option in the toolbar to open the schema editor



Adding a new column

- Each column has a unique column name (traditional naming rules apply) and a column type which is translated to a SQL data type

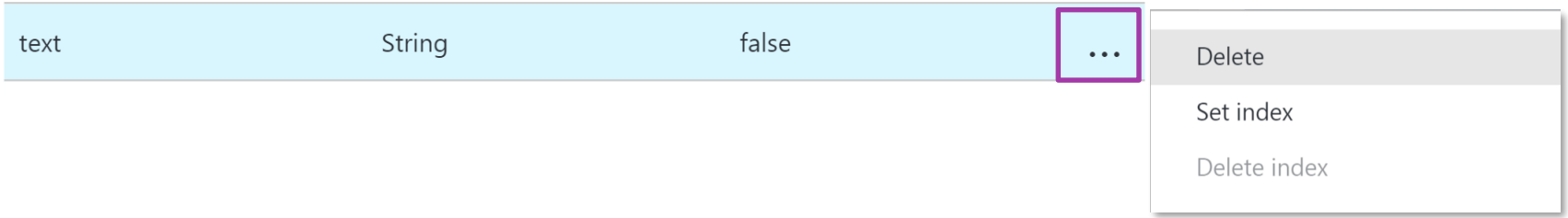


String	`nvarchar(max)`
Number	`float(53)`
Date	`datetimeoffset(7)`
Boolean	`bit`

Columns cannot be altered once you have created them so choose carefully!

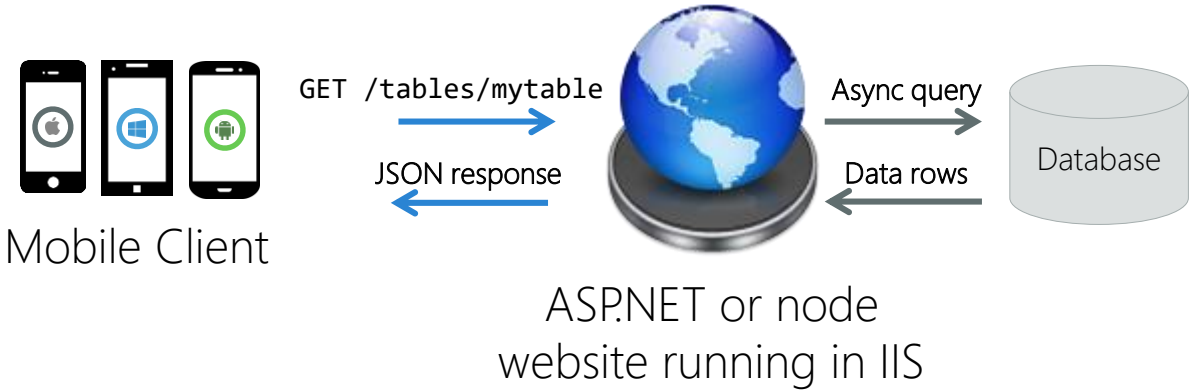
Deleting columns

- Can delete a column or create a DB index on it through the row context menu; deleting and re-creating a column is your portal workaround to changing the column definition



Accessing tables from a client

- App service exposes a hard-coded endpoint (/tables/<tablename>) to allow applications to perform DB queries and operations using HTTP



Testing the service endpoint

- Can use a REST client to interact and test the endpoint; mandatory HTTP header **ZUMO-API-VERSION: 2.0.0** must be included

Postman
(free Chrome plugin)

Paw
(macOS app)

REST client
(Firefox plugin)

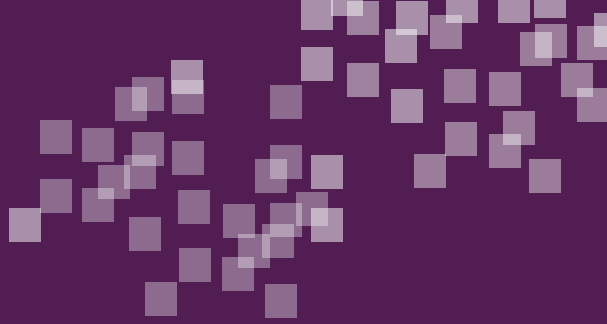
Fiddler
(Windows network analyzer)

```
curl -g -H ZUMO-API-VERSION:2.0.0 <site>/tables/{tablename}
```

Inserting, Updating and Deleting

- **POST** (insert), **PATCH** (update) and **DELETE** require an HTTP body encoded in JSON with a Content-Type set to **application/json**

```
POST /tables/{tablename} HTTP/1.1
Host: <site>
ZUMO-API-VERSION: 2.0.0
Content-Type: application/json

<JSON data goes here>
```



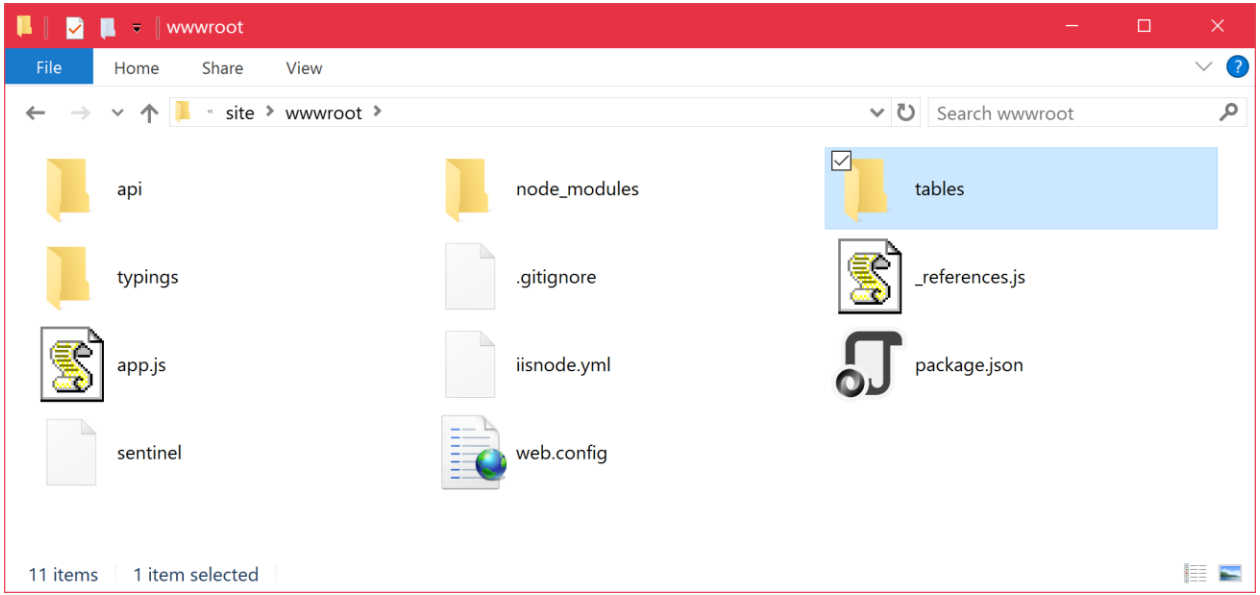
Individual Exercise

Add a new table into the Survey service

Xamarin University

Easy Table web structure

- Node.js looks in hardcoded **tables** folder for Easy Table definitions



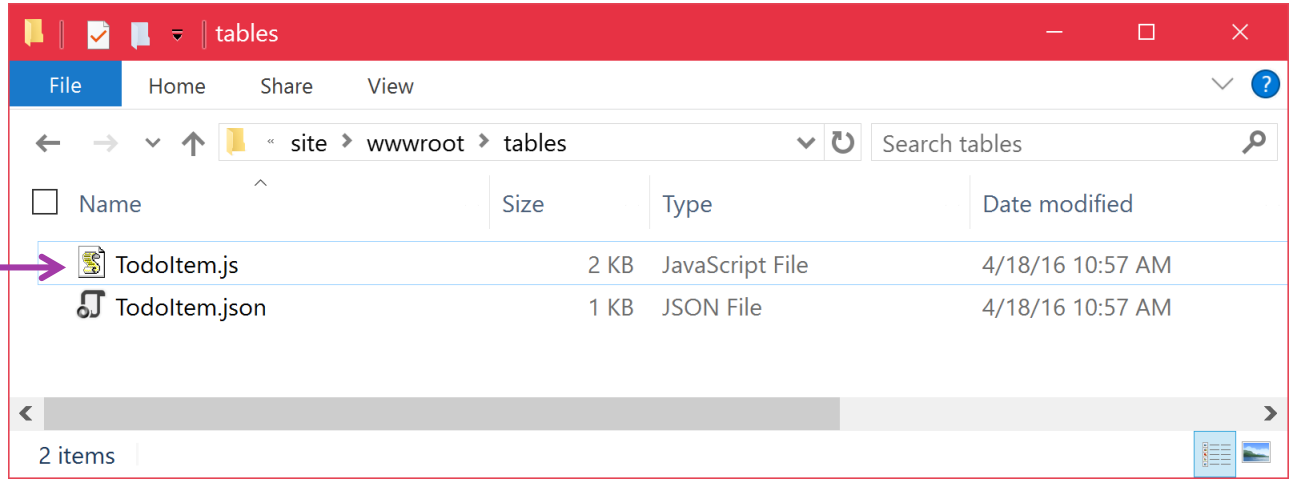
Adding new Easy tables to the server

- Can add or edit tables to Easy Table configuration by manipulating the files in the **tables** folder
- Server will create/edit your tables the next time an HTTP request is processed

Easy table definition

- Easy table is defined by **two files**; the filename is used to locate the proper SQL table and determine the final URL endpoint

Here we are defining the **todoitem** table & **/tables/todoitem** endpoint

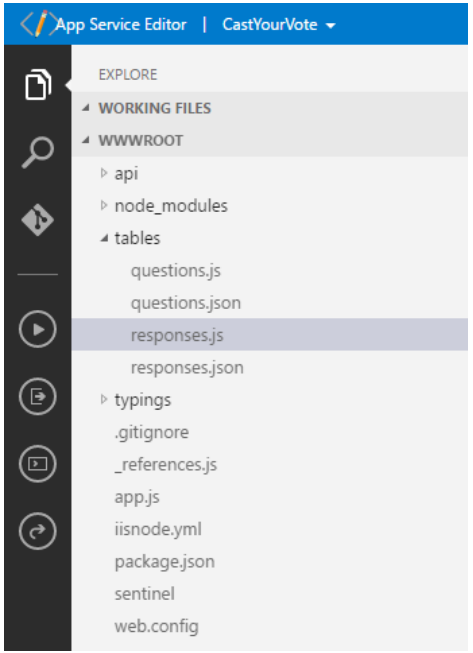


Editing existing Easy Tables

- Can also edit the scripts backing an Easy Table through the Azure portal

Can edit the already-created JSON files directly from the portal through the **Edit Script** toolbar button in each defined Easy Table blade

Opens site in VS online editor



Creating a new table in JSON

```
{
  "softDelete": true,
  "autoIncrement": false,
  "insert": {
    "access": "anonymous"
  },
  "update": {
    "access": "anonymous"
  },
  "delete": {
    "access": "anonymous"
  },
  "read": {
    "access": "anonymous"
  },
  "undelete": {
    "access": "anonymous"
  }
}
```

{table}.json

JSON **description file** provides default property values for table settings

Creating a new table in JSON

```
{
  "softDelete": true,
  "autoIncrement": false,
  "insert": {
    "access": "anonymous"
  },
  "update": {
    "access": "anonymous"
  },
  "delete": {
    "access": "anonymous"
  },
  "read": {
    "access": "anonymous"
  },
  "undelete": {
    "access": "anonymous"
  }
}
```

{table}.json

JS **table script** adds the table endpoint and provides a customization point for table operations

```
var app = require('azure-mobile-apps');
// Create a new easy table definition
var table = app.table();
// Never allow updates to records
table.update.access = 'disabled';

module.exports = table;
```

{table}.js

Defining custom columns

- Can define column structure as part of the table controller (.js) file

```
var app = require('azure-mobile-apps');
var table = app.table();

// Define our columns for the DB table (defined as JSON)
table.columns = {
    "text": "string",
    "isPrivate": "boolean"
};
// Turn off dynamic schema
table.dynamicSchema = false;

module.exports = table;
```

Turning off soft delete

- Node.js back end enables soft delete by default – can turn it off by changing the **softDelete** flag on the specific table

```
{
  "softDelete": false,
  "autoIncrement": false,
   ...
}
```

or

```
var table = module.exports =
   require('azure-mobile-apps')
     .table();
table.softDelete = false;
...
```

Can change the **softDelete** flag in the configuration or table controller source

Populating the DB with data

- Tables are created empty by default – there are several ways to pre-populate the SQL data; either as part of creation, or post-creation

Define schema and insert data from a **comma-separate-value** file

Code-first seed method

SQL Management Studio

Populating Easy Tables

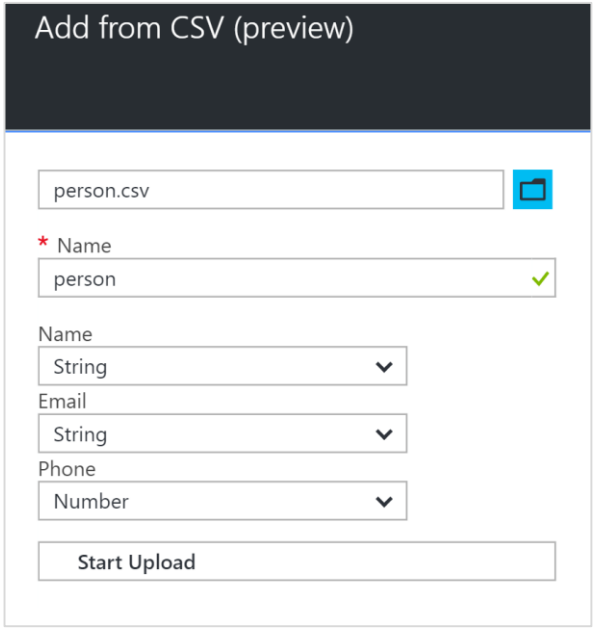
- **Add From CSV** option allows you to define the table structure *and* import records into the database with Easy Tables

person.csv

```
Name,Email,Phone
Mark Smith,mark@julmar.com,9025551212
John Doe,jodo@anonymous.com,2145551212
Jane Austin,jane@greengables.com,9725551212
```

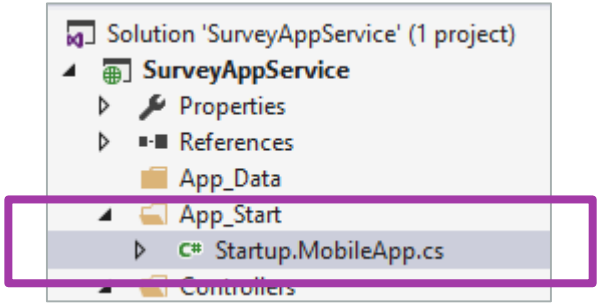
Add from CSV

File *must* have a terminating CR/LF on the final line or that line will not be imported



Populating Entity Framework

- ASP.NET and EF use an *initializer* method to create and seed a table when it does not exist in the target database

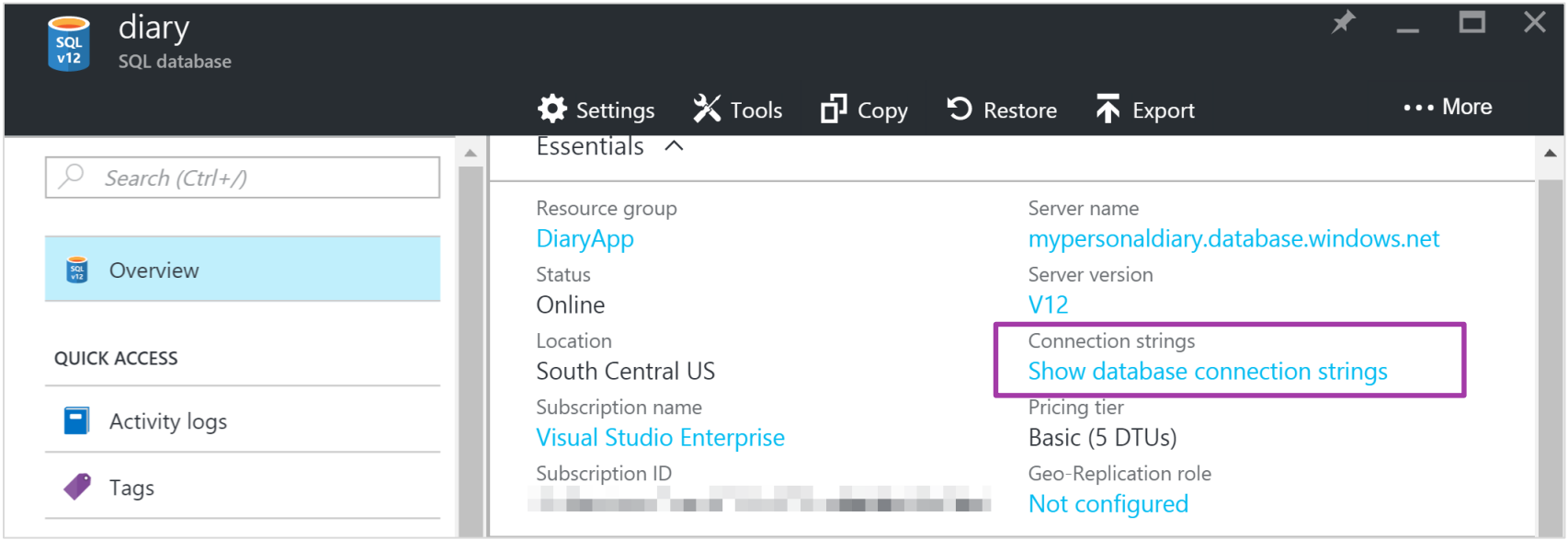



```
public class MobileServiceInitializer
    : CreateDatabaseIfNotExists<MobileServiceContext>
{
    protected override void Seed(MobileServiceContext context)
    {
        ... // TODO: add items to DB context here
    }
}
```

Add code to populate data using passed **DbContext** to the existing **MobileServiceInitializer** class

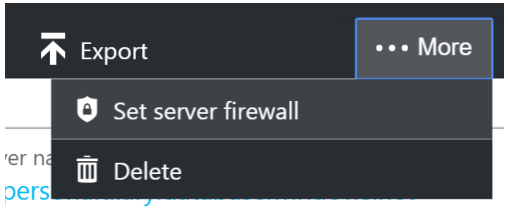
Populating the table with data

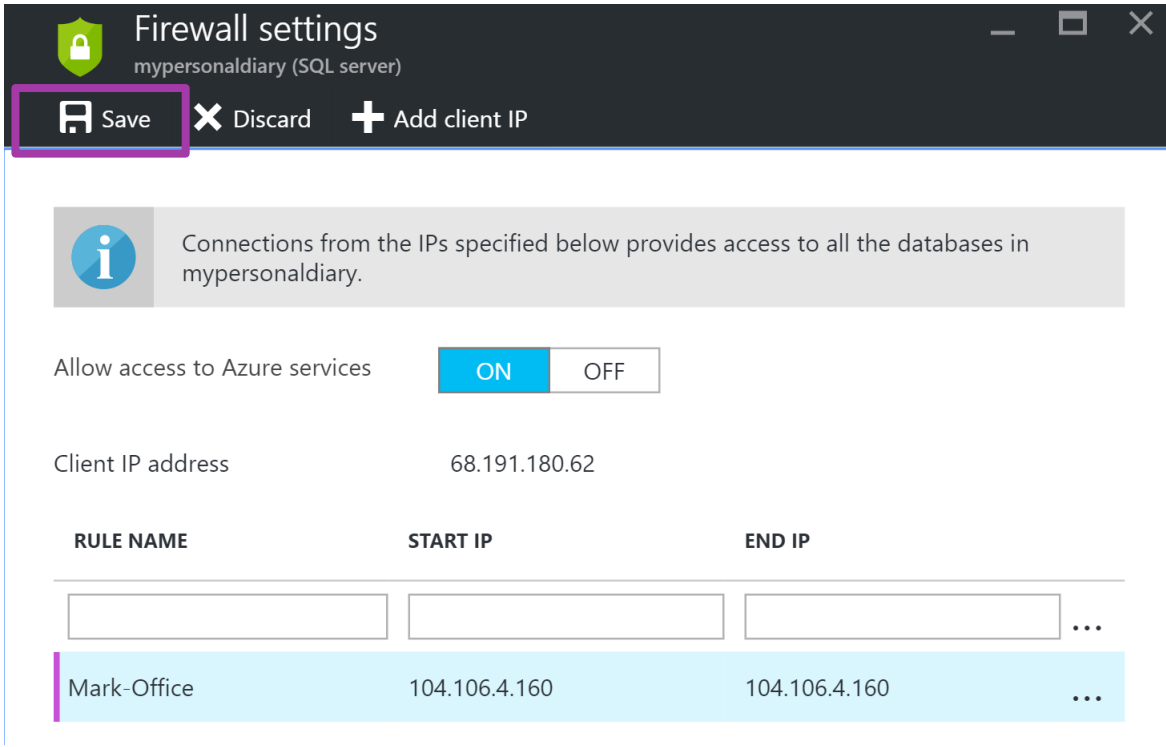
- Can add data directly into the underlying SQL database by accessing it through SQL Server credentials available from **SQL database blade**



Changing the firewall rules

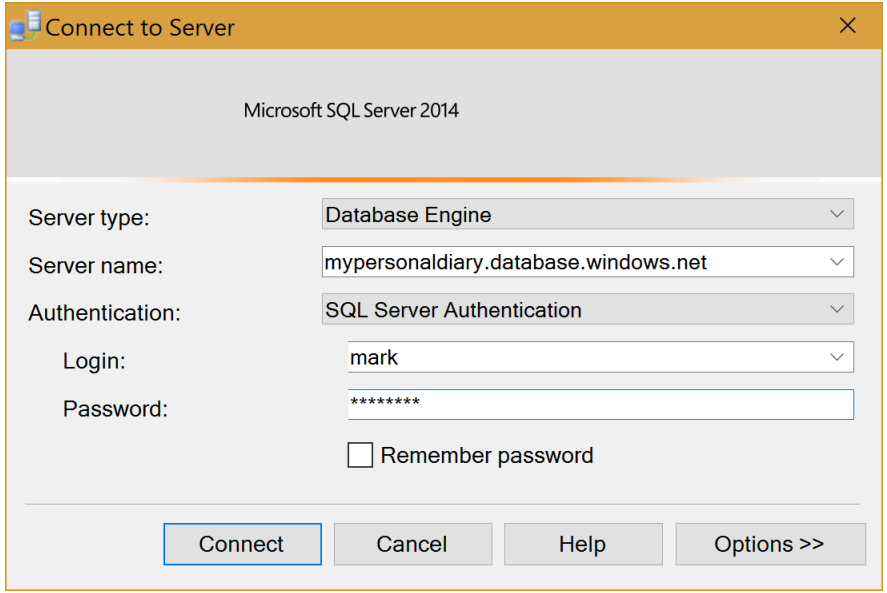
- By default, direct SQL access is limited to Azure, but you can change the firewall rules to allow external access

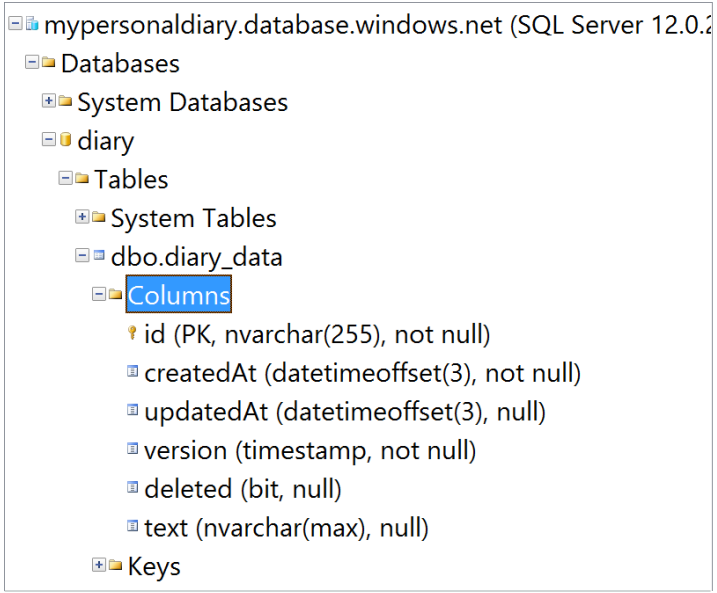




SQL Management Studio

- Azure sets up SQL Server authentication – just need server URL and user/password you setup the database with





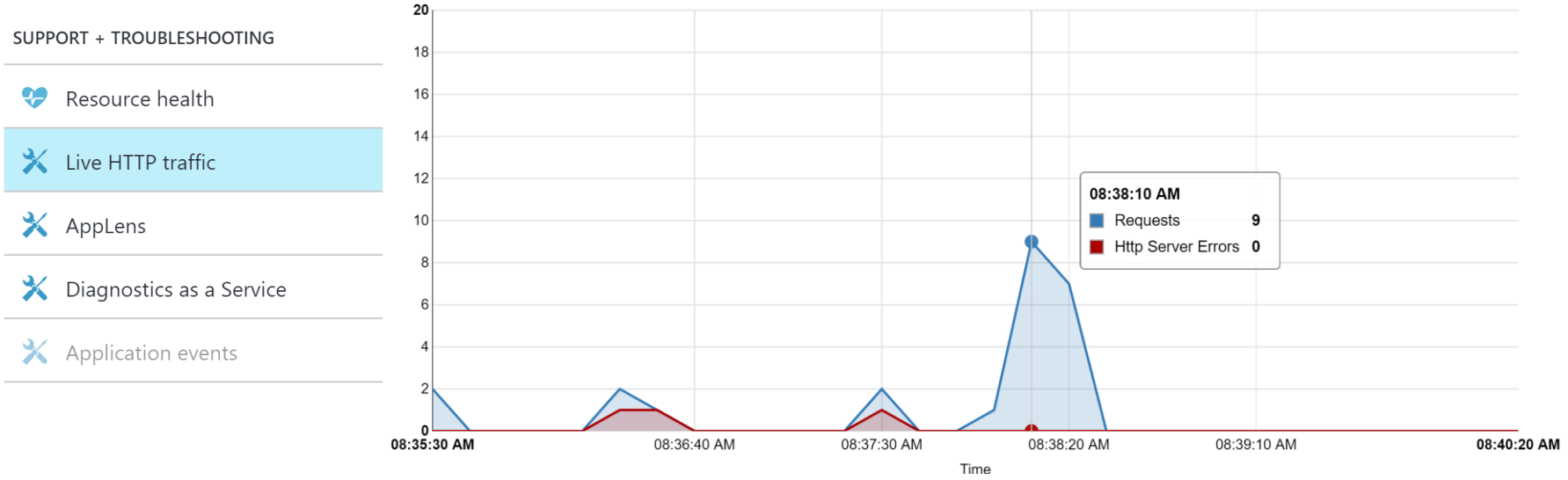
Individual Exercise

Create and populate our survey questions in the Azure portal

Xamarin University

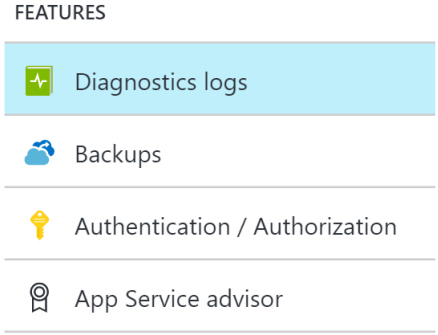
Monitoring the server side traffic

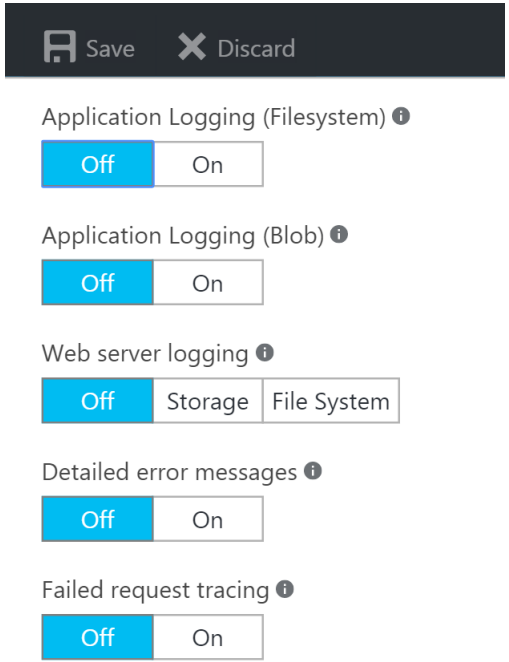
- Can monitor the server-side requests through the app portal blade



Get diagnostic details

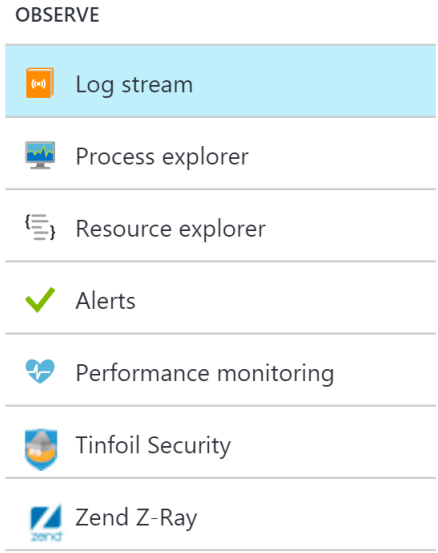
- Can enable IIS diagnostic logging in the app to get fine-grained details

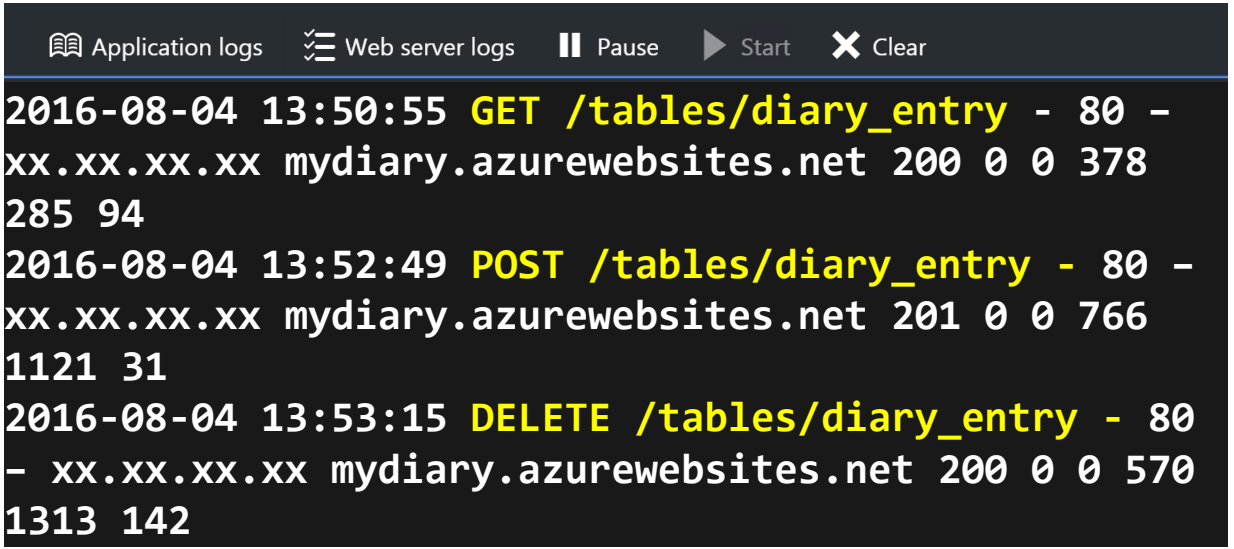




Watching the (almost) live stream

- Use the **Log Stream** feature to watch the app + IIS logs on the portal





Collecting other details

- Several other tools available in the portal to collect a variety of performance, runtime and historical data about your app

- App can also be configured to collect data from Application Insights and New Relic

Summary

1. Decide the proper type of database to add
2. Create the database + connection
3. Create one or more tables
4. Populate the database (optional)

Next Steps

- We've covered the basics of building an Azure App mobile service using either node or ASP.NET
- The next set of classes will focus on the client side and how to consume the service from a Xamarin application

Thank You!

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

Microsoft