This is a sample module from QuickLearn Training's course: Cloud-based Integration Using Azure App Service

Introduction to Custom API Apps

- Lesson 1: Anatomy of an API App
- Lesson 2: Fundamentals of Web API
- Lesson 3: Hello API App World
- Lesson 4: Debugging API Apps

QuickLearn TRAINING

Lesson: Anatomy of an API App

In this lesson, you will learn about…

- The components that make up an API App

Azure App Service

=

Outside the scope of this course

WEB APPS

Web apps that scale with your business

MOBILE APPS

Build Mobile apps for any device

LOGIC APPS

Automate business process across SaaS and on-premises

API APPS

Easily build and consume APIs in the cloud

QuickLearn TRAINING

What Are API Apps?

Implementation Agnostic Definition

- RESTful services that optionally expose tool-consumable metadata, which are hosted in API App containers in Azure

What Are API Apps?

.NET Based Implementation Definition

- Web API projects with a NuGet package reference to Swashbuckle for swagger metadata generation, which have an enhanced publish experience that can target an Azure API App container as a publish destination

What is Swagger?

Swagger

- Swagger is a standard metadata format (represented as a JSON object) for describing resources exposed by a RESTful API
- Plays a similar role for RESTful services to the role played by WSDL for SOAP-based services

What is Swashbuckle?

Swashbuckle

- Swashbuckle is a .NET library that generates swagger metadata by examining the final shape of a Web API

Requirements for Creating a Custom API App

Prerequisites

- Visual Studio 2013 or later
  - Community Edition is sufficient
- Azure SDK
  - Adds the Azure API App Template
  - Adds the API App publish target

Creating a Custom API App

1. Create a new Web Application in Visual Studio
   - Select the API App sub-template
2. Write the implementation of your logic
3. Publish the API App to a new or existing Azure API App container

Examining the Azure API App Template

QuickLearn TRAINING

Contents of an Azure API App Project

Standard Web API Starter

- WebApiConfig.cs
- ValuesController.cs

Swashbuckle Config

- SwaggerConfig.cs

QuickLearn TRAINING

Package References of an Azure API App Project

Referenced Packages

- Swashbuckle
  - Swagger metadata generator for .NET
- Microsoft ASP.NET Web API
  - Web API Libraries
- Application Insights for Web Applications (Optional)
  - Application Insights client for providing application telemetry

Exploring the Swashbuckle Configuration

SwaggerConfig.cs

- Contains commented documentation and sample code inline to guide you in emitting appropriate swagger metadata

```
2 references
public class SwaggerConfig
{
    0 references
    public static void Register()
    {
        var thisAssembly = typeof(SwaggerConfig).Assembly;

        GlobalConfiguration.Configuration
            .EnableSwagger(c =>
                {
                    // By default, the service root url is inferred from the request used to access the docs.
                    // However, there may be situations (e.g. proxy and load-balanced environments) where this does not
                    // resolve correctly. You can workaround this by providing your own code to determine the root URL.
                    //
                    //c.RootUrl(req => GetRootUrlFromAppConfig());

                    // If schemes are not explicitly provided in a Swagger 2.0 document, then the scheme used to access
                    // the docs is taken as the default. If your API supports multiple schemes and you want to be explicit
                    // about them, you can use the "Schemes" option as shown below.
                    //
                    //c.Schemes(new[] { "http", "https" });

                    // Use "SingleApiVersion" to describe a single version API. Swagger 2.0 includes an "Info" object to
                    // hold additional metadata for an API. Version and title are required but you can also provide
                    // additional fields by chaining methods off SingleApiVersion.
                    //
                    c.SingleApiVersion("v1", "WebApplication4");

                    // If your API has multiple versions, use "MultipleApiVersions" instead of "SingleApiVersion".
                    // In this case, you must provide a lambda that tells Swashbuckle which actions should be
                    // included in the docs for a given API version. Like "SingleApiVersion", each call to "Version"
```

Publishing an Azure API App

API App Publish Target

- Used to copy the files for your API App to a new or existing API App Container in Azure
- Accessible through the **Publish…** project context menu option

Publishing an Azure API App

QuickLearn TRAINING

Lesson 2: Fundamentals of Web API

In this lesson, you will learn about…

- How the key components of a Web API application come together to build out C# based HTTP APIs

History of Web API

Released with ASP.NET MVC 4.0

- Borrowed the model/controller components of MVC
  - Web API controllers had their own base class
- Borrowed convention-based development style
- Not explicitly bound to MVC 4.0

Key Components of a Web API Application

Routes

- Map inbound HTTP requests to methods that handle those requests
- Defined through class/method-level attributes

Controllers

- Classes implementing API logic within methods designed to handle incoming HTTP requests
- May derive from ApiController*

Models

- Classes that model the shape of API inputs/outputs

Creating a Controller (Code Example)

```
public class ValuesController
                    : ApiController
{

}
```

`ApiController` is the base class of API Controllers

QuickLearn
TRAINING

Configuring Routes (Code Example)

```
[RoutePrefix("api/values")]
public class ValuesController : ApiController
{
    [Route("{id}")]
    public IHttpActionResult GetValue(int id)
    {
    }
}
```

RoutePrefix attribute sets prefix for all routes in controller.

Route attribute specifies the template for an action's relative route

QuickLearn TRAINING

Returning Results

Options for Returning Web API Results

- Return POCO directly
  - Pro: Easy to set up and understand
  - Con: `HttpResponseException` must be used to signal non-200 responses
- Return `HttpResponseMessage`
  - Pro: Can return any value / status code using `Request.CreateResponse` extension method
  - Con: Response type metadata cannot be auto-generated

Options for Returning Web API Results

- Return `IHttpActionResult`
  - Pro: Provides asynchronous factory method for creating `HttpResponseMessage` instance, generated through helper methods on `ApiController`
  - Con: Response type metadata cannot be auto-generated, not generated by Azure App Service SDK extension methods

ApiController Extension Methods for Results

`ApiController` **has been extended to return results as** `IHttpActionResult` **instances through these extension methods:**

- BadRequest
- Conflict
- Content
- Created
- CreatedAtRoute
- InternalServerError
- Json
- NotFound
- Ok
- Redirect
- RedirectToRoute
- StatusCode
- Unauthorized

Implementing Actions (Code Example)

```
[RoutePrefix("api/values")]
public class ValuesController : ApiController
{
        [Route("{id}")]
        public IHttpActionResult GetValue(int id)
        {
                var values = new string[] { "value1", "value2" };

                if (id < 0 || id > values.GetUpperBound(0))
                        return NotFound();

                return Ok<string>(values[id]);
        }
}
```

IHttpActionResult is the return type

This extension method on ApiController returns IHttpActionResult corresponding to HTTP Status code 404

Some extension methods provide generics to return typed payload

QuickLearn TRAINING

Testing HTTP GET Requests Against an API

HTTP GET Requests

- GET is the verb web browsers use by default when typing a URL
- For RESTful services it would be used to return a given resource without side effects (allowing safe caching of result)

Examining HTTP Requests

- Requests can be examined by using F12 tools in browser
- Requests can be examined by using Network/HTTP traffic monitoring tool (e.g., Fiddler, Wireshark)

What Does Web API Do With My Request?

The following process occurs within the application:

1. Routes defined on controllers and actions are loaded on application startup
2. Incoming requests are evaluated against the route table
3. The first method with a matching route is executed
   - Incoming parameters are read from request query string and/or body and mapped to an instance of a model class
4. The method result is returned to the caller
   - May undergo translation to acceptable format (e.g., JSON, XML)

Calling an Action – Raw HTTP Request

```
GET /api/values/1 HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-US,en;q=0.7,el;q=0.3
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64;
Trident/7.0; Touch; rv:11.0) like Gecko
Accept-Encoding: gzip, deflate
Host: localhost:1710
DNT: 1
Connection: Keep-Alive
Cache-Control: no-cache
```

QuickLearn
TRAINING

Calling the Action – Raw HTTP Response

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/10.0
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET
Date: Thu, 16 Apr 2015 22:40:49 GMT
Content-Length: 8

"value2"
```

QuickLearn
TRAINING

Is This API RESTful?

REST

- **RE**presentational **S**tate **T**ransfer is an approach to building an HTTP API
  - All RESTful APIs are pure HTTP APIs
  - Not all pure HTTP APIs are RESTful
- Web API can be used to build RESTful APIs
- Not all APIs built using Web API are RESTful by design

Designing RESTful API Apps

RESTful APIs align themselves to HTTP

- URLs – Refers to a ***resource*** with which the API can interact
- HTTP Verbs – Define the desired action against the resource
- HTTP Status Code – Define the result of the action taken against the resource

Comparing RESTful and Non-RESTful Approach

Scenario: As a developer, I need a way to update the balance on an account

```
POST /Account/Update

{
    "account": "1234",
    "balance": 123.45
}
```

URL is the thing we want to do, HTTP verb is arbitrary

Body includes information that describes the target resource

```
PATCH /Account/1234

{
    "balance" : 123.45
}
```

Exact resource and action is known within the first line of the request

Body describes only the nature of the update

QuickLearn TRAINING

Supporting Multiple HTTP Verbs for RESTful APIs

Method-level Attributes Control Matching HTTP Verbs

- HttpDelete
- HttpGet
- HttpHead
- HttpOptions
- HttpPatch
- HttpPost
- HttpPut

Supporting Multiple HTTP Verbs

```
[RoutePrefix("api/values")]
public class ValuesController : ApiController
{
        [Route("{id}")]
        [HttpGet]
        public IHttpActionResult GetValue(int id)
        {
                /* Method Body Here */
        }

        [Route("{id}")]
        [HttpPut]
        public IHttpActionResult UpdateValue(int id, [FromBody]string value)
        {
                /* Method Body Here */
        }
}
```

HttpGet attribute ensures this action is invoked for GET requests

Here the HttpPut attribute is used. This is appropriate for a full update of an existing resource.

FromBody indicates that the parameter should be deserialized out of the HTTP request body.

QuickLearn TRAINING

Lesson 3: Hello API App World

In this lesson, you will learn about…

- What it takes to build a Hello World API App

Building Hello World API App

Building Hello World involves:

1. Setting up the Project
2. Creating the Model Classes
3. Creating a Controller
4. Defining the Actions
5. Publishing the API App

Setting Up the Project

Create the Project

1. Create a new **ASP.NET Web Application** project
2. Select the **Azure API App** template

QuickLearn TRAINING

Exposing Functionality to Logic Apps

Logic App designer can natively consume Web APIs

- Cards will show all possible operations
- Operations and parameters may have unfriendly names
- Operations and parameters will lack meaningful descriptions

Metadata can be added to API Apps to make them more friendly in the Logic Apps designer

- The *T-Rex Metadata Library* NuGet package can help

Introducing the T-Rex Metadata Library

T-Rex Provides .NET Attributes

- Can be used to decorate methods/parameters with designer names and descriptions
- Can be used to indicate triggers

T-Rex Provides Swashbuckle Filters

- Swashbuckle filters change how swagger metadata is generated so that it includes data from .NET attributes

Source Code / Documentation

- https://github.com/nihaue/trex

NuGet Package

- http://www.nuget.org/packages/trex

```
GlobalConfiguration.Configuration
    .EnableSwagger(c =>
        {
            c.SingleApiVersion("v1", "HelloWorld");
            c.ReleaseTheTRex();
        });
```

QuickLearn TRAINING

Setting Up the Project

Enable Metadata Generation

3. Add the **TRex** Nuget Package
4. In **SwaggerConfig.cs** add `using` directive for `TRex.Metadata`
5. In **SwaggerConfig.cs** add `c.ReleaseTheTRex()` inside configure action parameter of `EnableSwagger` method call

```
using HelloWorld;
using Swashbuckle.Application;
using System.Web.Http;
using TRex.Metadata;
using WebActivatorEx;

[assembly: PreApplicationStartMethod(typeof(SwaggerConfig), "Register")]
namespace HelloWorld
{
    2 references
    public class SwaggerConfig
    {
        0 references
        public static void Register()
        {
            var thisAssembly = typeof(SwaggerConfig).Assembly;

            GlobalConfiguration.Configuration
                .EnableSwagger(c =>
                    {
                        c.SingleApiVersion("v1", "HelloWorld");
                        c.ReleaseTheTRex();
                    });
        }
    }
}
```

Setting Up the Project

Remove Placeholder Content

6. Delete the placeholder **ValuesController.cs**

QuickLearn TRAINING

Building Hello World API App

Building Hello World involves:

1. Setting up the Project
2. Creating the Model Classes
3. Creating a Controller
4. Defining the Actions
5. Publishing the API App

Why Create the Models First?

RESTful APIs revolve around the data

- In Web API, this data is our models
- Defining models first helps get you into a RESTful mindset rather than RPC mindset

Creating the Model Classes

Define the Model

1. Add a class to the **Models** folder
2. Add relevant properties in the class

```
using System;

namespace HelloWorld.Models
{
    3 references
    public class Greeting
    {
        1 reference
        public Greeting()
        {
            Text = "Hello";
            GreetedThing = "World";
        }

        1 reference
        public string Text { get; set; }

        1 reference
        public string GreetedThing { get; set; }
    }
}
```

QuickLearn TRAINING

Creating the Model Classes

Decorate the Model Class

3. Add a `using` directive for `TRex.Metadata`
4. Decorate the properties with the `Metadata` attribute *(Optional)*

```
using System;
using TRex.Metadata;

namespace HelloWorld.Models
{
    3 references
    public class Greeting
    {
        1 reference
        public Greeting()
        {
            Text = "Hello";
            GreetedThing = "World";
        }

        [Metadata("Greeting", "Greeting given to the thing")]
        1 reference
        public string Text { get; set; }

        [Metadata("Greeted Thing", "The thing being greeted")]
        1 reference
        public string GreetedThing { get; set; }
    }
}
```

QuickLearn TRAINING

Building Hello World API App

Building Hello World involves:

1. Setting up the Project
2. Creating the Model Classes
3. Creating a Controller
4. Defining the Actions
5. Publishing the API App

Creating a Controller

Create the Controller Class

1. Add an **Empty Web API 2 Controller** to **Controllers** folder
2. Add a `RoutePrefix` attribute to define the base route for any actions defined inside

```
using System.Web.Http;

namespace HelloWorld.Controllers
{
    [RoutePrefix("api/greetings")]
    public class GreetingController : ApiController
    {


    }
}
```

QuickLearn TRAINING

Building Hello World API App

Building Hello World involves:

1. Setting up the Project
2. Creating the Model Classes
3. Creating a Controller
4. Defining the Actions
5. Publishing the API App

Defining the Actions

Define an Action

1. Add a `using` directive for the `Models` namespace
2. Define a new method that returns one of the model classes
3. Decorate the method with the `Route` attribute and `HttpGet` attribute

```
using HelloWorld.Models;
using System.Web.Http;

namespace HelloWorld.Controllers
{
    [RoutePrefix("api/greetings")]
    0 references
    public class GreetingController : ApiController
    {

        // GET /api/greetings/default
        [HttpGet]
        [Route("default")]
        public Greeting GetDefault()
        {

        }

    }
}
```

QuickLearn TRAINING

Defining the Actions

Add Action Metadata

4. Add a `using` directive for `TRex.Metadata`
5. Decorate the action with the `Metadata` attribute

```
using HelloWorld.Models;
using System.Web.Http;
using TRex.Metadata;

namespace HelloWorld.Controllers
{
    [RoutePrefix("api/greetings")]
    0 references
    public class GreetingController : ApiController
    {

        // GET /api/greetings/default
        [HttpGet]
        [Route("default")]
        [Metadata("Get Default Greeting",
            "Returns a greeting saying hello to the world.")]
        public Greeting GetDefault()
        {

        }

    }
}
```

QuickLearn TRAINING

Defining the Actions

Implement the Action

6. Write the code necessary to generate the appropriate model instance

```
using HelloWorld.Models;
using System.Web.Http;
using TRex.Metadata;

namespace HelloWorld.Controllers
{
    [RoutePrefix("api/greetings")]
    0 references
    public class GreetingController : ApiController
    {

        // GET /api/greetings/default
        [HttpGet]
        [Route("default")]
        [Metadata("Get Default Greeting",
            "Returns a greeting saying hello to the world.")]
        0 references
        public Greeting GetDefault()
        {
            return new Greeting();
        }

    }
}
```

Building Hello World API App

Building Hello World involves:

1. Setting up the Project
2. Creating the Model Classes
3. Creating a Controller
4. Defining the Actions
5. Publishing the API App

“Gotchas” When Publishing API Apps to App Service

Web Deploy Can Publish API Apps Directly into an Azure Hosting Environment

- Web Deploy does not know it is an API App
  - Destination must be an API App created in Azure
- Debugging must be explicitly enabled, if desired

Publishing the API App

Publish the API App

1. Right-click the project, and then click **Publish...**
2. Choose the **Microsoft Azure API Apps** publish target

QuickLearn TRAINING

Publishing the API App

Publish the API App

3. **Select** or **create** API App hosting container

Publishing the API App

Publish the API App

4. On the **Settings** tab, choose the **Debug** configuration, if desired
5. Click **Publish**

QuickLearn TRAINING

What Does My API App Look Like in the Designer?

QuickLearn TRAINING

What Is the Raw Result of Execution?

```
{
    "Text": "Hello",
    "GreetedThing": "World"
}
```

QuickLearn
TRAINING

Lesson 4: Debugging API Apps

In this lesson, you will learn about…

- Testing your API App at development time by simulating inbound requests, intercepting outbound requests, and debugging logic within your API App

Verifying Deployed API App Behavior

API Apps Deal in HTTP Requests and Responses

- To verify behavior of a deployed API App, you must use HTTP requests as test inputs

QuickLearn
TRAINING

Tools for Replicating HTTP Requests

Once your API App is accessible externally, you can simulate requests using:

- Postman
  - Used primarily to generate HTTP requests by hand
  - https://www.getpostman.com/
- Fiddler Tool
  - Used primarily to monitor outgoing HTTP traffic, but can also generate HTTP requests by hand
  - http://www.telerik.com/fiddler

Inspecting HTTP Calls to External Resources

Some API Apps are Connectors to External HTTP Resources

- Tools previously identified cannot inspect those calls
- When testing, you may wish to reconfigure your API App to call a fake HTTP resource that allows inspection of calls

QuickLearn TRAINING

Tools for Inspecting Inbound Requests

You can use the following tools to fake an external HTTP resource

- Requestb.in
  - Provides randomly generated endpoint on demand
  - Logs all requests to the endpoint for inspection
  - Allows all HTTP verbs, and responds 200 OK for every inbound request
    - *Not a good choice if your API App relies on the response*
- HttpResponder.com
  - Provides vanity endpoint on demand
    - *Not a good choice if your requests/responses are private*
  - Logs all requests to the endpoint for inspection
  - Allows all HTTP verbs, and responds 200 OK for every inbound request
  - Allows customization of JSON and XML responses

Using Requestb.in to Examine Requests

Using Requestb.in to Examine Requests

1. Browse to http://requestb.in/
2. Click **Create a RequestBin**
3. Issue HTTP requests to the provided Bin URL
4. Inspect requests by appending ?inspect to the Bin URL

Using Requestbi.in to Examine API App Outputs

You might also use requestb.in to examine API App outputs

1. Include API App in a Logic App
2. Add HTTP action
3. Configure HTTP action to POST the outputs to a Requestb.in

QuickLearn TRAINING

Demo: Generating and Examining HTTP Requests

In this demonstration, you will see how to…

- Create a Requestb.in
- Generate HTTP Requests using Postman
- Inspect requests in a Requestb.in

QuickLearn
TRAINING

What About Inspecting Execution Inside an API App?

You May Need to Attach a Debugger

- Azure SDK provides this capability through the Server Explorer interface inside of Visual Studio
- Remote debugging experience mirrors the local debugging experience

Attaching a Debugger to an API App

Debugging a Deployed API App

1. Open the Visual Studio Solution containing the API App
2. In Server Explorer, expand **Azure > App Service**
3. Expand the *Resource Group* containing the deployed API App
4. Right-click the API App, and then click **Attach Debugger**
5. Set breakpoints and debug the API App as usual

QuickLearn TRAINING

Reviewing Testing and Debugging Concepts

QuickLearn TRAINING

Demo: Building and Deploying Hello World

In this demonstration, you will see how to…

- Create a new API App project
- Define model classes
- Create a controller class
- Define routes
- Implement actions
- Publish and Test an API App

QuickLearn
TRAINING

Lab 13A: Building Custom Action API Apps [75 min]

In this lab, you will…

- Create a custom action API App
- Install the TRex NuGet package
- Use the T-Rex Metadata Library to generate Swagger metadata for API Apps
- Publish API Apps to Azure
- Build Logic Apps that use custom API Apps

QuickLearn
TRAINING