LEARNING asp.net-web-api2

#asp.net-web-api2

Table of Contents

About

Chapter 1: Getting started with asp.net-web-api2

Remarks

This section provides an overview of what asp.net-web-api2 is, and why a developer might want to use it.

It should also mention any large subjects within asp.net-web-api2, and link out to the related topics. Since the Documentation for asp.net-web-api2 is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

Detailed instructions on getting asp.net-web-api 2 set up or installed.

What and why Asp.Net Web API2?

What and Why ?

Asp.Net's Web API2 is the latest version of Web API. It is an easy way to implement a RESTful web service using all of the goodness that the Asp.Net framework provides. Once you understand the basic principles of REST, then a Asp.net Web API2 will be very easy to implement. Web API2 is built on Asp.Net's modular, pluggable pipeline model. This means that when a server hosting a web API2 receives a request, it passes through Asp.Nets request pipeline first. This enables you to easily add your own modules if you find that the default capabilities are not enough for your needs. With the recent announcements on `ASP.net vNext` this also means you can potentially host your Web API2 outside of Windows Server which opens up a whole range of usage cases. See here for detail.

How works ?

Web API2 uses the Controller and Action concepts from MVC. Resources are mapped directly to controllers; you would typically have a different controller for each of your main data entities (Product, Person, Order etc). Web API2 uses the Asp.Net routing engine to map URLs to controllers. Typically, APIs are held within a `/api/` route which helps to distinguish API controllers from other non-API in the same website.

Actions are used to map to specific HTTP verbs, for example you would typically have a GET action which returns all of the entities. This action would respond to `/api/Products` (where 'products' is your controller) and would look something like this:

```
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}
```

You may also have a `GET` action which accepts a specific `ID` and returns a specific entity. It would respond to `/api/Products/81` and would look something like this:

```
public string Get(int id)
 {
    return "value";
 }
```

There are many great hidden benefits to using Web API which you may not realise but actually save you a lot of work.

Web API2 is part of the ‘One Asp.Net’

Web API2 is part of the ‘One Asp.Net’ family which means that it natively supports all of the great shared features you may currently use with MVC or web forms, this includes (these are just a few examples):

- Entity Framework
- Authorisation and identity
- Scaffolding
- Routing

Serialization and Model Binding

Web API2 is setup by default to provide responses in either XML or JSON (JSON is default). However, as a developer you do not need to do any conversion or parsing – you simply return a strongly typed object and Web API2 will convert it to XML or JSON and return it to the calling client, this is a process called Content Negotiation. This is an example of a `GET` action which returns a strongly typed Product object.

```
public Product GetProduct(int id)
{
   var product = _products.FirstOrDefault(p => p.ID == id);
   if (product == null)
   {
      throw new HttpResponseException(HttpStatusCode.NotFound);
   }
   return Request.CreateResponse(HttpStatusCode.OK, product);
}
```

This also works for incoming requests using a feature called Model Validation. With Model Validation, Web API2 is able to validate and parse incoming response body data to a strongly typed object for you to work with in your code. This is an example of model binding:

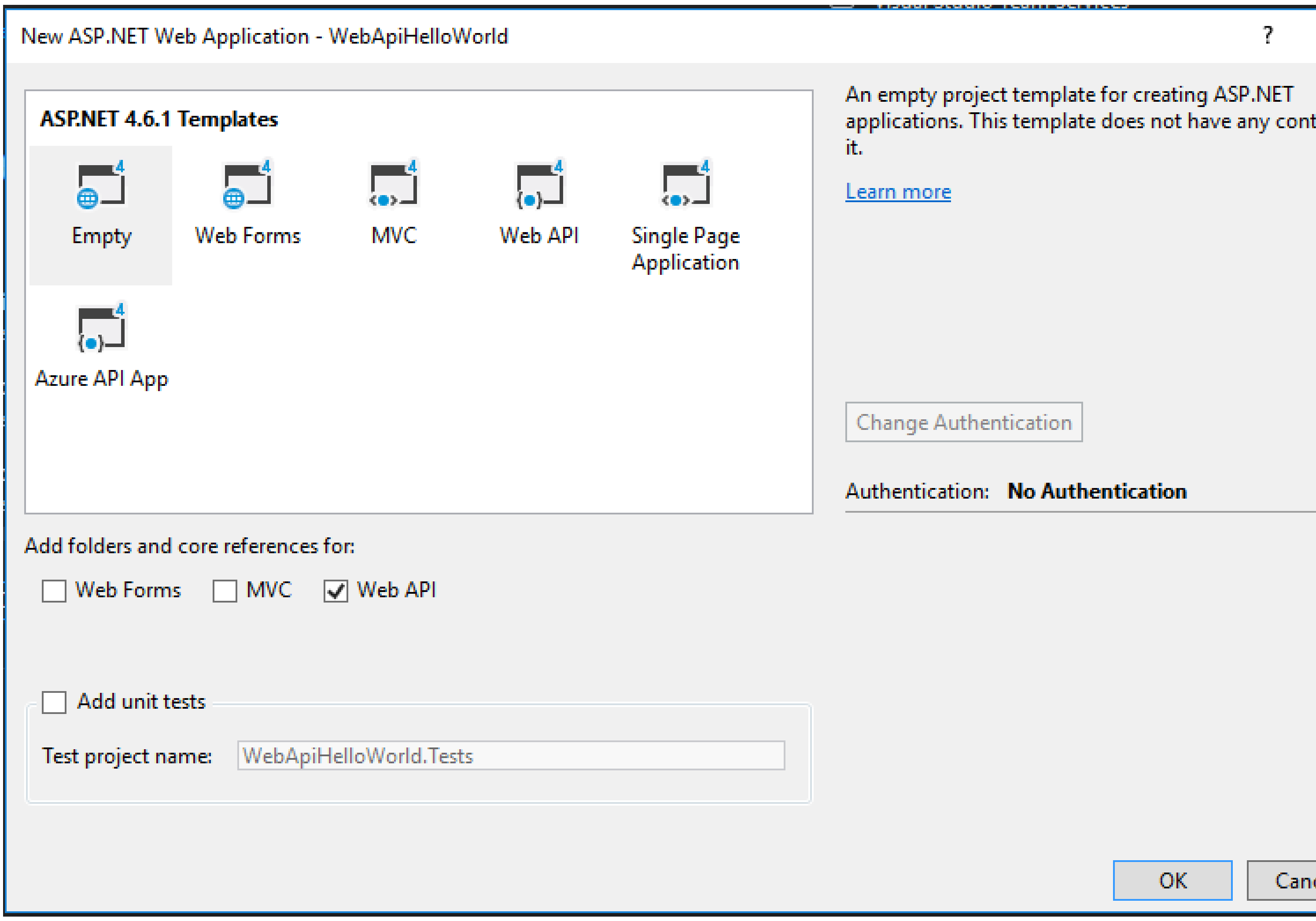
```
public HttpResponseMessage Post(Product product)
{
    if (ModelState.IsValid)
    {
        // Do something with the product (not shown).

        return new HttpResponseMessage(HttpStatusCode.OK);
    }
    else
    {
        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
    }
}
```

Hello Web Api

Web Api 2 - Hello World example

We are going to create a new Web Api simple application which return to us Json with message and user name.
Lets start! First create new Web Api project using Visual Studio and select Empty Template. Be sure to check "Web Api" folder:


New ASP.NET Web Application - WebApiHelloWorld

ASP.NET 4.6.1 Templates

An empty project template for creating ASP.NET applications. This template does not have any cont it.

Authentication: No Authentication

Add folders and core references for:

[ ] Web Forms [ ] MVC [✓ Web API]

[ ] Add unit tests

Test project name: WebApiHelloWorld.Tests

NOTE I didn't choose "Web Api" template because it adds reference to ASP.NET MVC to provide API Help Page. And in such basic application we don't really need it.

Adding a model

A model is an C# class that represents some data in our app. ASP.NET Web API is able to automatically serialize model to JSON, XML, or some other format (depends on configuration).

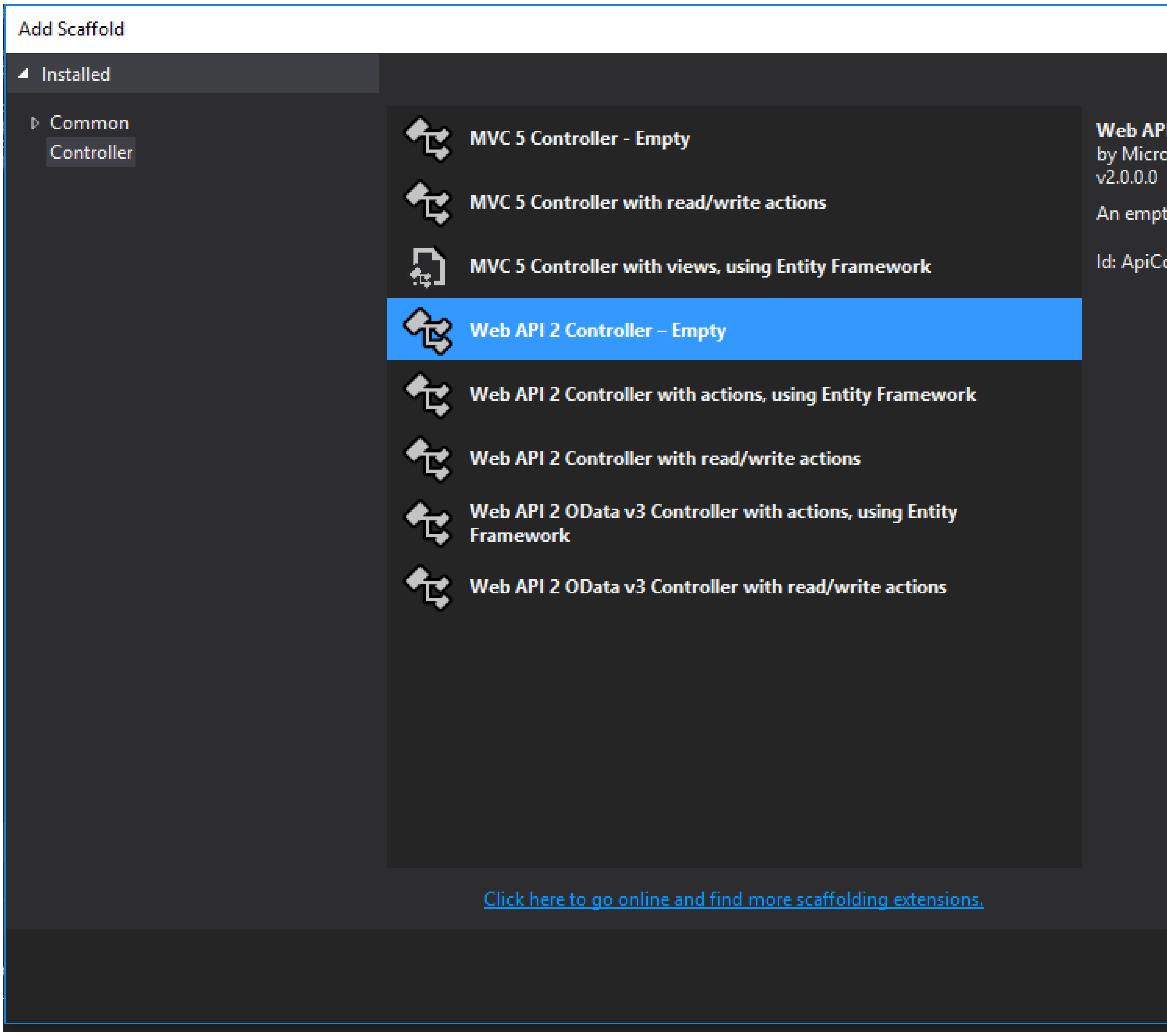
In our application we will create only one model, but real-world apps usually have a lot of them.

In Solution Explorer, right-click the **Models folder**. Next select **Add** and then select **Class**. Name the class "HelloMessage". Our model needs two properties: *MessageText* and *UserName*:

```
namespace WebApiHelloWorld.Models
{
    public class HelloMessage
    {
        public string MessageText { get; set; }
        public string UserName { get; set; }
    }
}
```

Adding a controller

A controller handles HTTP requests. Our app needs only one controller that returns Json with Hello message and user name (which we will pass in URL).
In Solution Explorer, right-click the **Controllers folder**. Next select **Add** and then select **Controller**. In the opened window, select **Web API Controller - Empty** and click **Add**.


Add Scaffold
Installed
Common
Controller
MVC 5 Controller - Empty
MVC 5 Controller with read/write actions
MVC 5 Controller with views, using Entity Framework
Web API 2 Controller – Empty
Web API 2 Controller with actions, using Entity Framework
Web API 2 Controller with read/write actions
Web API 2 OData v3 Controller with actions, using Entity Framework
Web API 2 OData v3 Controller with read/write actions
Web AP
by Micro
v2.0.0.0
An empt
Id: ApiC
Click here to go online and find more scaffolding extensions.

Set controller name as "HelloController". Next edit code of created controller. We need to add method which returns Hello message.

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

namespace WebApiHelloWorld.Controllers
{
    public class HelloController : ApiController
    {
        public HelloMessage GetMessage(string name)
        {
            HelloMessage message = new HelloMessage
            {
                MessageText = "Hello my Dear!",
                UserName = name
            };
            return message;
        }
    }
}
```

NOTE Be sure to add `using WebApiHelloWorld.Models`. Without it your controller won't find HelloMessage class.

Finish

That's all! Now you only need to build and start your application. Simply hit **Ctrl + F5** or just **F5** (to start without debugging). Visual studio will launch web browser. You need to call your controller. To do that add at the end of the URL "/api/hello?name=John". The result should be:

```
{
    "MessageText": "Hello my Dear!",
    "UserName": "John"
}
```

Read Getting started with asp.net-web-api2 online: https://riptutorial.com/asp-net-web-api2/topic/3323/getting-started-with-asp-net-web-api2

Chapter 2: Attribute Routing in ASP.NET Web API 2

Introduction

As the name suggests, this uses attributes to route. This gives the user more control over the URI's in the WebAPI. For example, you can describe hierarchies of the resource. However, the earlier 'Conventional Routing' is fully supported. Users can have a mixture of both too.

Syntax

- [RoutePrefix("api/books")] - for controller class
- [Route("getById")] - for actions
- [Route("~/api/authors/{authorId:int}/books")] - for overriding route prefix

Parameters

Parameter Name	Details
RoutePrefix	attribute to the controller class. all common url prefixes in actions are clubbed here. takes string as input
Route	attribute to the controller actions. each action will have route assosciated with(not necessarily)
Route("~/api/")	this overrides the Route Prefix

Remarks

Currently, Attribute Routes doesn't have Controller specific Message Handlers. As there is no way to specify Which handler to execute for which route at the time of declaration. This is possible in Conventional Routing.

Examples

Basic Attribute Routing

Simply add an attribute to the controller action

```
[Route("product/{productId}/customer")]
public IQueryable<Product> GetProductsByCustomer(int productId)
```

```
{
    //action code goes here
}
```

this will be queried as /product/1/customer and productId=1 will be sent to the controller action.

Make sure the one within '{ }' and the action parameter are same. productId in this case.

before using this, you have to specify that you are using Attribute Routing by:

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.MapHttpAttributeRoutes();
    }
}
```

Route prefixes

Usually, the routes in a controller have the same prefix connected somehow with functionality of this controller. For example:

```
public class ProductsController : ApiController
{
    [Route("api/products")]
    public IEnumerable<Product> GetProducts() { ... }

    [Route("api/products/{id:int}")]
    public Product GetProduct(int id) { ... }

    [Route("api/products")]
    [HttpPost]
    public HttpResponseMessage CreateProduct(Product product) { ... }
}
```

In such scenario we can set common prefix for whole controller. To do so we use the `[RoutePrefix]` attribute:

```
[RoutePrefix("api/products")]
public class ProductsController : ApiController
{
    // GET api/products
    [Route("")]
    public IEnumerable<Product> GetProducts() { ... }

    // GET api/products/5
    [Route("{id:int}")]
    public Product GetProduct(int id) { ... }

    //POST api/products
    [Route("")]
    [HttpPost]
    public HttpResponseMessage CreateProduct(Product product) { ... }
}
```

Overriding route prefix

If we want to override the route prefix we can use a tilde **(~)** in the routing attribute of the method:

```
[RoutePrefix("api/products")]
public class ProductsController : ApiController
{
    // GET api/owners/products
    [Route("~/api/owners/products")]
    public IEnumerable<Product> GetProducts() { ... }

    //...
}
```

Read Attribute Routing in ASP.NET Web API 2 online: https://riptutorial.com/asp-net-web-api2/topic/9559/attribute-routing-in-asp-net-web-api-2

Chapter 3: OAuth 2.0 in ASP.NET Web API

Remarks

Registering in an Android Application

These are the steps I've taken to log in / register using an Android app:

- Have a login activity which queries the ExternalLogins route, getting the available providers. This activity should have the NoHistory flag enabled and launch as a single instance.
- On a user's button press, launch a Custom Chrome Tab with the provider's URL. The user should be logged in and redirected back to your published site at the given return URL. Don't use a WebView.
- Have this page redirect the user again, using a custom URI scheme to launch a post-login activity within your application with the access token, expiry date and user account details added as additional data. This will need to be done in JavaScript on the web page, as the server's controllers can't access the URL parameters.
- Store the user's details and token in a local MySQL database. On each login, check to see if the token is still in date.
- Any calls to the API can now be authorized using the Authorization HTTP header, with your stored token added as so: “Bearer {token}”

Examples

Configuring an OAuth Provider

You need to get some details from your OAuth provider of choice. We'll be looking at Google, but ASP.NET is also set up to allow out-the-box use of Twitter, Facebook and Microsoft (obviously).

You'll want to go to the Google developer console (https://console.developers.google.com/) and create a project, enable the Google+ API (for getting the user's profile info, such as their name and avatar) and create a new OAuth 2 Client ID in the “Credentials” section. The authorized JavaScript origins should be your project's root URL (e.g. https://yourapi.azurewebsites.net) and the redirect URIs need to include ASP's built-in Google callback endpoint (https://yourapi.azurewebsites.net/signin-google) as well as your callback route of choice (https://yourapi.azurewebsites.net/callback). Getting these wrong will result in Google having a hissy fit.

Back in your Visual Studio project, open App_Start > Startup.Auth.cs. Replace the commented Google section at the bottom with the code below, adding the ID and Secret from the Google Developers Console:

```
var googleAuthOptions = new GoogleOAuth2AuthenticationOptions()
{
      ClientId = "YOUR ID",
      ClientSecret = "YOUR SECRET",
            Provider = new GoogleOAuth2AuthenticationProvider()
            {
                  OnAuthenticated = (context) =>
                  {
                        context.Identity.AddClaim(new Claim("urn:google:name",
context.Identity.FindFirstValue(ClaimTypes.Name)));
                        context.Identity.AddClaim(new Claim("urn:google:email",
context.Identity.FindFirstValue(ClaimTypes.Email)));
                        //This following line is need to retrieve the profile image
                        context.Identity.AddClaim(new Claim("urn:google:accesstoken",
context.AccessToken, ClaimValueTypes.String, "Google"));
                        return System.Threading.Tasks.Task.FromResult(0);
                  }
            }
      };
app.UseGoogleAuthentication(googleAuthOptions);
```

These additional claims allow you to query Google for the user's profile information, such as their name and avatar URL.

Storing OAuth User Profiles

When someone registers with your application, a new ApplicationUser object will be stored in the database. By default the class is very barebones, but it can be customised - you can find it in Models > IdentityModels.cs. This is mine:

```
public class ApplicationUser : IdentityUser
{
      public string ImageUrl { get; set; }
      public DateTime DateCreated { get; set; }
      public string FirstName { get; set; }
      public string AuthProvider { get; set; }
      public string Surname { get; set; }

      public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser>
manager, string authenticationType)
      {
            // Note the authenticationType must match the one defined in
CookieAuthenticationOptions.AuthenticationType
            var userIdentity = await manager.CreateIdentityAsync(this, authenticationType);
            // Add custom user claims here
            return userIdentity;
      }
}
```

For reference, the user profile returned from Google with the Google+ API enabled takes the following JSON structure:

```
{{
    "id": "1****************6",
    email": "dan********@gmail.com",
    "verified_email": true,
    "name": "Dan Richardson",
    "given_name": "Dan",
    "family_name": "Richardson",
    "link": "https://plus.google.com/+DanRichardson",
    "picture": "https://lh4.googleusercontent.com/photo.jpg",
    "gender": "male",
    "locale": "en"
}}
```

Allowing Redirect URLs Other Than Site Root

Go to Providers > ApplicationOAuthProvider.cs and edit the ValidateClientRedirectUri function. This was a big gotcha to me, as if you don't do this there'll be a fantastically unhelpful error message. By default, this code will make any callbacks to your site invalid unless they're to the site's root. You likely want to be able to handle the callbacks in a controller, so you'll need to change it to something like this:

```
public override Task ValidateClientRedirectUri(OAuthValidateClientRedirectUriContext context)
{
      if (context.ClientId == _publicClientId)
      {
            Uri expectedRootUri = new Uri(context.Request.Uri, "/");
            Uri expectedCallbackUri = new Uri(context.Request.Uri, "/callback");

            if (expectedRootUri.AbsoluteUri == context.RedirectUri ||
                  expectedCallbackUri.AbsoluteUri == context.RedirectUri)
            {
                  context.Validated();
            }
      }
      return Task.FromResult<object>(null);
}
```

Registration Flow

Here is the default flow of registering a user in Web API. All of these routes can be found in the AccountController:

- The user requests a list of the login providers using the GetExternalLogins route, passing a return URL as a parameter. This returns an array of provider objects, containing the provider's name and the route that should be requested in order to log in with it (each configured to use the given return url).

  e.g. GET: /api/Account/ExternalLogins?returnUrl=/callback&generateState=true, where the requested return URL is /callback

- The user calls one of these returned URLs in a browser, where they're redirected to the provider's login page. Once logged in, the provider passes a cookie back to ASP, which handles the creation of an external user account.

- The user will be redirected to the return URL they passed in the first step. An external access token is passed back to the user, appended to the URL in a # param. This token can only be used on select routes, such as RegisterExternal.

- The user now sends a POST request to RegisterExternal, using the new access token as a

Bearer key. ASP then creates a new ApplicationUser and returns a proper access token which can be used on any route.

Storing OAuth Profile Information

I have found that the Web API template is broken - the default implementation relies on cookies in the final step, which you probably don't want to be using in a Rest API. Without a cookie, GetExternalLoginInfoAsync in RegisterExternal always returns null. I removed RegisterExternal entirely, instead creating the final user account in GetExternalLogin - called on return from the OAuth provider (in this case, Google):

```
[OverrideAuthentication]
[HostAuthentication(DefaultAuthenticationTypes.ExternalCookie)]
[AllowAnonymous]
[Route("ExternalLogin", Name = "ExternalLogin")]
public async Task<IHttpActionResult> GetExternalLogin(string provider, string error = null)
{
      if (error != null)
      {
            return Redirect(Url.Content("~/") + "#error=" + Uri.EscapeDataString(error));
      }

      if (!User.Identity.IsAuthenticated)
      {
            return new ChallengeResult(provider, this);
      }

      ExternalLoginData externalLogin = ExternalLoginData.FromIdentity(User.Identity as
ClaimsIdentity);

      if (externalLogin == null)
      {
            return InternalServerError();
      }

      if (externalLogin.LoginProvider != provider)
      {
            Authentication.SignOut(DefaultAuthenticationTypes.ExternalCookie);
            return new ChallengeResult(provider, this);
      }

      ApplicationUser user = await UserManager.FindAsync(new
UserLoginInfo(externalLogin.LoginProvider,
                externalLogin.ProviderKey));

      bool hasRegistered = user != null;

      if (hasRegistered)
      {
            Authentication.SignOut(DefaultAuthenticationTypes.ExternalCookie);

            ClaimsIdentity oAuthIdentity = await user.GenerateUserIdentityAsync(UserManager,
                    OAuthDefaults.AuthenticationType);
            ClaimsIdentity cookieIdentity = await user.GenerateUserIdentityAsync(UserManager,
            CookieAuthenticationDefaults.AuthenticationType);

            AuthenticationProperties properties =
ApplicationOAuthProvider.CreateProperties(user.UserName);
```

```
                Authentication.SignIn(properties, oAuthIdentity, cookieIdentity);
      }
      else
      {
            var accessToken = Authentication.User.Claims.Where(c =>
c.Type.Equals("urn:google:accesstoken")).Select(c => c.Value).FirstOrDefault();
            Uri apiRequestUri = new
Uri("https://www.googleapis.com/oauth2/v2/userinfo?access_token=" + accessToken);

            RegisterExternalBindingModel model = new RegisterExternalBindingModel();

            using (var webClient = new System.Net.WebClient())
            {
                  var json = webClient.DownloadString(apiRequestUri);
                  dynamic jsonResult = JsonConvert.DeserializeObject(json);
                  model.Email = jsonResult.email;
                  model.Picture = jsonResult.picture;
                  model.Family_name = jsonResult.family_name;
                  model.Given_name = jsonResult.given_name;
            }

            var appUser = new ApplicationUser
            {
                  UserName = model.Email,
                  Email = model.Email,
                  ImageUrl = model.Picture,
                  FirstName = model.Given_name,
                  Surname = model.Family_name,
                  DateCreated = DateTime.Now
            };

            var loginInfo = await Authentication.GetExternalLoginInfoAsync();

            IdentityResult result = await UserManager.CreateAsync(appUser);
            if (!result.Succeeded)
            {
                  return GetErrorResult(result);
            }

            result = await UserManager.AddLoginAsync(appUser.Id, loginInfo.Login);
            if (!result.Succeeded)
            {
                  return GetErrorResult(result);
            }

            ClaimsIdentity oAuthIdentity = await
appUser.GenerateUserIdentityAsync(UserManager,
                   OAuthDefaults.AuthenticationType);
            ClaimsIdentity cookieIdentity = await
appUser.GenerateUserIdentityAsync(UserManager,
                    CookieAuthenticationDefaults.AuthenticationType);

            AuthenticationProperties properties =
ApplicationOAuthProvider.CreateProperties(appUser.UserName);
            Authentication.SignIn(properties, oAuthIdentity, cookieIdentity);
      }

      return Ok();
}
```

This removes the need for the client application to perform a needless POST request after receiving an external access token.

Read OAuth 2.0 in ASP.NET Web API online: https://riptutorial.com/asp-net-web-api2/topic/7603/oauth-2-0-in-asp-net-web-api

Credits

S. No	Chapters	Contributors
1	Getting started with asp.net-web-api2	Archil Labadze, Community, Krzyserious, Mostafiz
2	Attribute Routing in ASP.NET Web API 2	Ajay Aradhya, Krzyserious
3	OAuth 2.0 in ASP.NET Web API	GS_Dan