**Conventional Routing vs Attribute Routing –**

* **Conventional Routing –**
* The routes are defined using *route templates*.
* At least one route should be defined in route collection.
* E.g.,

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

// Add default route using convention-based routing

config.Routes.MapHttpRoute(

name: "DefaultApi",

***routeTemplate: "api/{controller}/{id}",***

defaults: new { id = RouteParameter.Optional }

);

}

}

* We need to mention the most specific routes first, then the general routes.
* When more routes are defined, it becomes difficult to manage it.
* **Attribute Routing –**
* The routes are defined using *[Route()]* attribute.
* The [Route()] attribute can be applied to any controller OR action method.
* For using Attribute routing, we must use config.MapHttpAttributeRoutes() in WebApiConfig.cs
* E.g.,

**WebApiConfig.cs -**

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

// Enable attribute routing

config.MapHttpAttributeRoutes();

}

}

**Use attribute in Controller / Action method –**

public class StudentController : ApiController

{

***[Route("api/student/names")]***

public IEnumerable<string> Get()

{

return new string[] { "student1", "student2" };

}

}

* With attribute routing, route definitions stay near to their action methods. This makes the route definitions more manageable.

**Parameter Binding –**

* Web API binds action method parameter either from URL’s query string or from request body.
  + By default, if the parameter is a primitive type then it gets value from the query string.
  + By default, if the parameter is a complex type then it gets value from the request body.
* The query string parameter name must match with the action method parameter, but it is not case-sensitive. Also, they can be in different order.
* The request body’s type properties must match with the properties of the action method type, but it is not case-sensitive.
* To change the default behavior of parameter binding, use [FromUri] & [FromBody] attribute.
* **Use [FromUri],** to force web API to get value of a complex type from query string.
  + E.g.,

**Controller –**

public class StudentController : ApiController

{

public Student Get(**[FromUri]** Student stud)

{

}

}

<http://localhost:xxxx/api/student?id=1&name=steve>

if we request above URL, the web API will create a ***student*** type with ***id*** & ***name*** as properties with values from query string.

* **Use [FromBody],** to force web API to get values of primitive type from request body.
  + E.g.,

**Controller –**

public class StudentController : ApiController

{

public Student Post**([FromBody]** string name)

{

}

}

**Request body –**

“steve”

* + [FromBody] attribute can be applied to only one parameter name of an action method, i.e., you cannot apply it for multiple parameter.

To summarize the parameter binding rules, refer below figure,

**Action Parameter Type** **Binding Source**

Query String

Request Body

Request Body

Query String

[FromUri] Complex Type

[FromBody] Primitive Type

Complex Type

Primitive Type

**Action method return types –**

* The web API action methods can have following return types –

1. Void
2. Primitive / Complex Type
3. HttpResponseMessage
4. IHttpActionResult

* It is not necessary that all action methods should return a result, it can return void.
* The advantage of ***HttpResponseMessage*** is, we can configure the response as per the requirement. We can set status code, content or error message.
* **IHttpActionResult** was introduced in Web API 2.
* We can use various methods (like Ok(), NotFound()) that returns an object that implement IHttpActionResult.
* We can create a custom class which will implement IHttpActionResult and we can use that class as a custom result type.
  + E.g.,

**Custom class -**

Public Class **TextResult** : IHttpActionResult

{

// custom logic..

}

**Action method –**

public IHttpActionResult GetName(int id)

{

string name = GetStudentName(id);

if (String.IsNullOrEmpty(name))

{

return NotFound();

}

return new **TextResult**(name, Request);

}

**Data Formats –**

* In Http Request, the MIME type is specified in the request header using **Accept** and **content-type** attribute.
* **Accept –** 
  + It specifies the format in which client wants data.
  + For e.g., if we specify the header with ***accept: application/json***, then client expects the response in JSON format.
* **Content-type –** 
  + It specifies the format in which client sends data.
  + For e.g., if we specify the header with ***content-type: application/json***, then client sends request data in JSON format.
* By default, Web API handles both JSON & XML data base on Accept & content-type headers.

For e.g., if we send a POST request with ***accept: application/json*** & ***content-type: application/xml***, then we will get response in JSON format.

And, if we send a POST request with ***accept: application/xml*** & ***content-type: application/json***, then we will get response in XML format.

**Media Type Formatters –**

* Web API includes built-in media formatters –

1. JsonMediaTypeFormatter
2. XmlMediaTypeFormatter
3. FormUrlEncodedMediaTypeFormatter
4. JQueryMvcFormUrlEncodedFormatter

* We can customize the built-in formatters.
  + For E.g., To write JSON property names in camelCase notation, we can set CamelCasePropertyNamesContractorResolver on the serializer settings.

*public static class WebApiConfig*

*{*

*public static void Register(HttpConfiguration config)*

*{*

*// configure json formatter*

*JsonMediaTypeFormatter jsonFormatter = config.Formatters.JsonFormatter;*

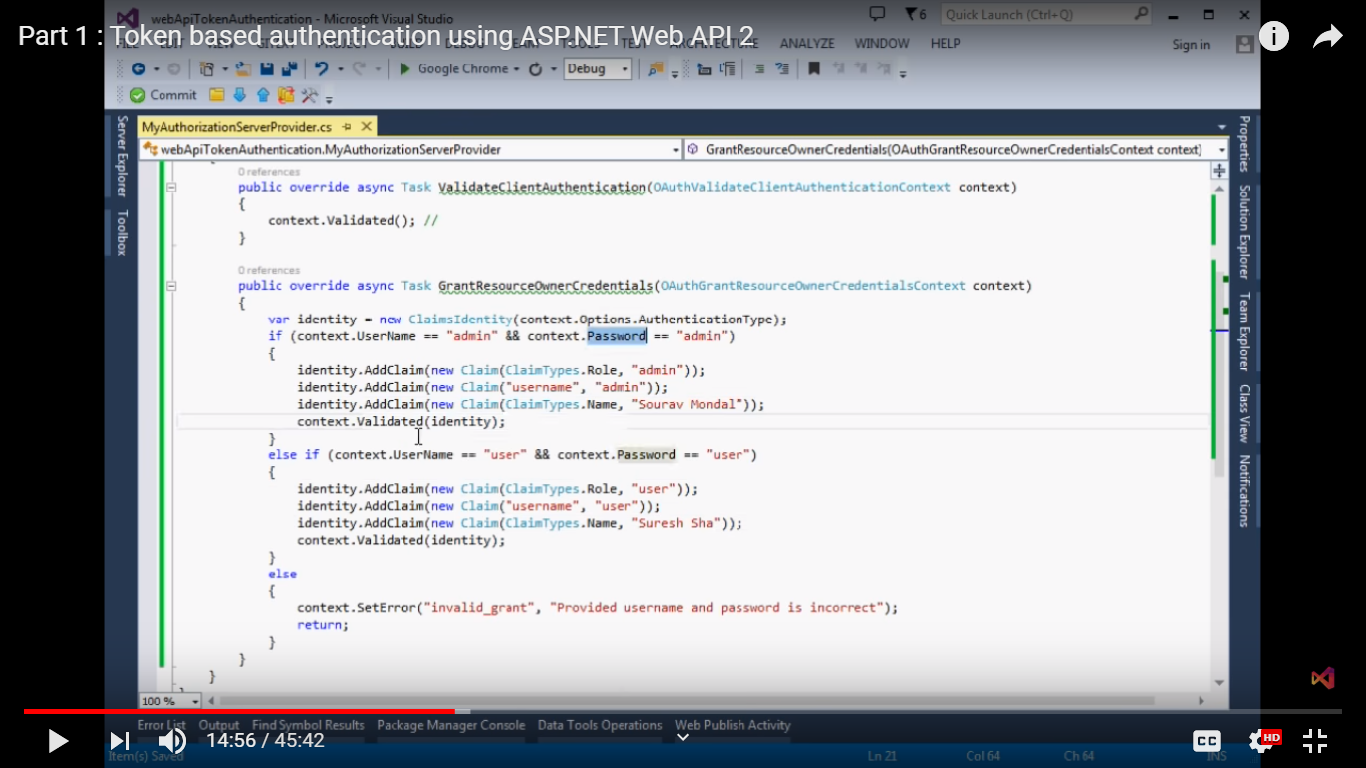
*jsonFormatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();*

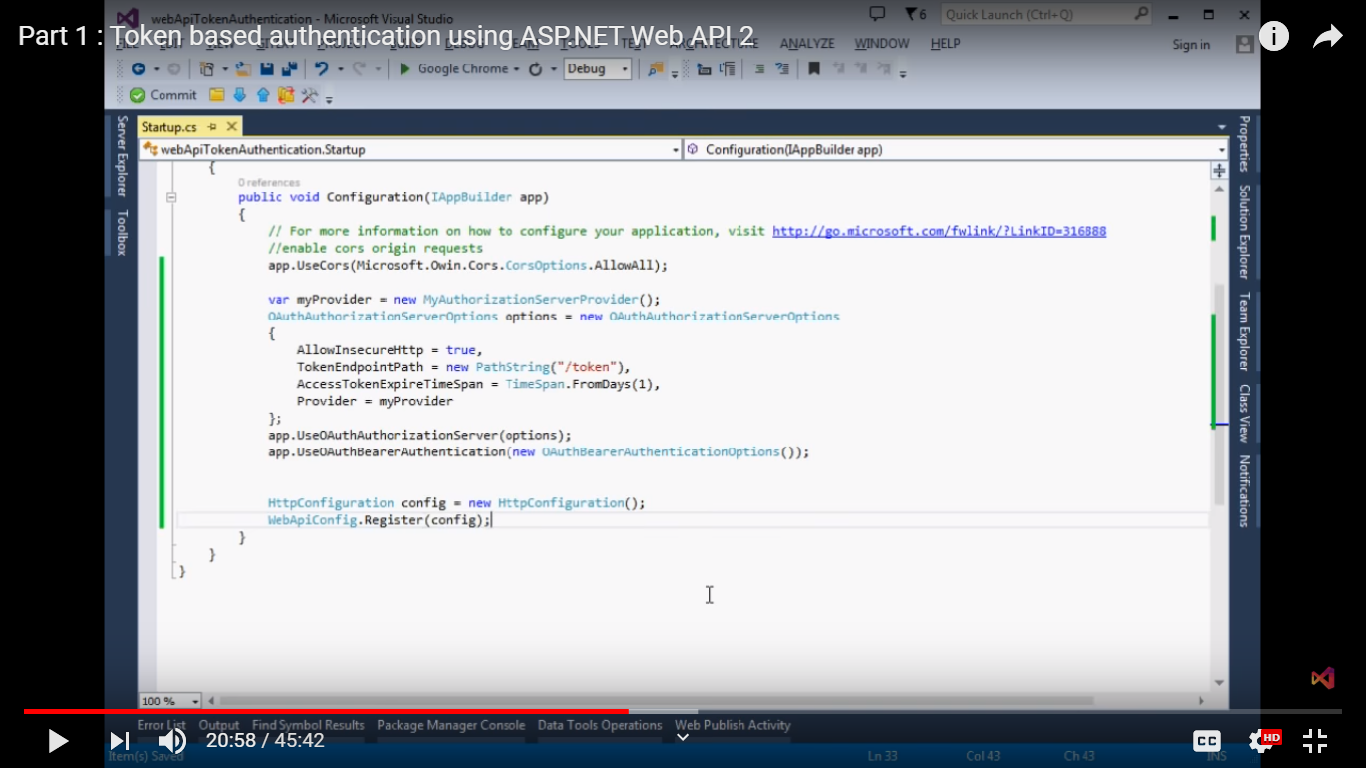
*}*

*}*

**Token Based Authentication –**

* We can use OWIN for implementing token-based authentication.
* Steps –
  1. Install the OWIN nuget packages to the project/solution.
     + Microsoft.Owin.Host.SystemWeb
     + Microsoft.Owin.Security.OAuth
     + Microsoft.Owin.Cors (required for implementation of CORS)
  2. Create a class which will inherit ***OAuthAuthorizationServerProvider*** class
     + Override ***ValidateClientAuthentication*** method
     + Override ***GrantResourceOwnerCredentials*** method
     + Sample code –



* 1. Create an OWIN startup class to configure OAuth server options.
     + In this class we need to configure, OAuth server options like TokenEndPointPath, AccessTokenExpirationTimeSpan, Provider(the class created in step 2 above)
     + Sample code – 
* To create token, we need to send a HTTP POST request to OAuth server endpoint ***“/Token”*** with ***Username, Password & Grant\_Type***. Then a token is issued by the OAuth server.
* Then store the token at client side, may be in session, and use it for accessing web API methods. For every request, you need to append below header in HTTP Request –
  + Authorization: “Bearer {token}”
* The token has the expiration time. So, when it expires, you need to again issue it.

**JWT Token –**

* JSON Web Token is an open standard that defines a compact & self-contained way for securely transmitting data between the parties as a JSON object.
* This is digitally signed, so it can be trusted.
* JWTs can be signed using a secret like HMAC algorithm, OR public/private key pair using RSA or ECDSA.
* JWT token structure –
  1. **Header –**
* Header contains type of token & signing algorithm.
* E.g. ***{ “alg” : “HS256”, “typ” : “JWT”}***
* Then this JSON is encoded to Base64URL string.
  1. **Payload –**
* This contains the Claim information like, user, and additional data.
* This data is then Base64Url encoded to form the 2nd part of the token.
* Do not put secret information in payload unless it is encrypted.
  1. **Signature –**
* To create Signature part, you need to take encoded Header, encoded Payload, a secret, and the algorithm mentioned in header & sign that.
* For e.g., If HMAC SHA256 algorithm is used then Signature will be created in following way –

***HMACSHA256(***

***Base64UrlEncode(header) + “.” +***

***Base64UrlEncode(payload),***

***secret***

***)***

* Refer <https://jwt.io/introduction/> for more information.
* Sample JWT token –

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.S9R\_7glkciZg-gwz4GbsrL-2YknG0mGTLPpH\_qHLY\_0

**CORS –**

* **Same-Origin Policy -** Browser security prevents a web page from making AJAX requests to another domain. This restriction is called the ***same-origin policy***, and prevents a malicious site from reading sensitive data from another site.
* CORS is a W3C standard that allows a server to relax the same-origin policy.
* Using CORS, a server can explicitly allow some cross-origin requests while rejecting others.
* CORS is safer and more flexible than earlier techniques such as JSONP.
* **Implementation –**
  + Install nuget package - Microsoft.AspNet.WebApi.Cors
  + WebApiConfig.cs, add below code –

using System.Web.Http;

namespace WebService

{

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

// New code

config.EnableCors();

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

}

}

}

* + Add [EnableCors] attribute to the Controller –

using System.Net.Http;

using System.Web.Http;

using System.Web.Http.Cors;

namespace WebService.Controllers

{

[EnableCors(origins: "http://mywebclient.azurewebsites.net", headers: "\*", methods: "\*")]

public class TestController : ApiController

{

// Controller methods not shown...

}

}

**Dependency Injection in Web API –**

Read it from Microsoft docs - <https://docs.microsoft.com/en-us/aspnet/web-api/overview/advanced/dependency-injection>