**===ASP.NET Web API**

The ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers, mobile devices & IOT Things. The ASP.NET Web API is an ideal platform for building Restful applications on the .NET Framework.

**===Why to use the Web API**

Currently most mobile devices, browsers and tablets are the medium for accessing most of the internet and in this also people are using mobile apps the most and to provide data to apps we are now going to use the Microsoft new technology called Web API.

If you want to expose the data/information of your application to your clients and other people then that other people can use your data and interact with the data/information you expose to them.

For example, a mobile application requires a service.

HTML 5 requires a service.

Desktop PC and tablets require services.

Currently most device apps require Web API services.

**ValuesController is WebAPI controller which inherits from API Controller. Which is under system.web.http namespace.**

**===Content Negotiation**

IMP & Always go through it

<http://www.c-sharpcorner.com/article/asp-net-web-api-content-negotiation/>

Using Content-Type the client can specify the format in which they send data from the server (i.e. request sends from server).

Content-Type: application/json

Using this acceptheader the client can specify the format which they want from the server (i.e. response needed from server).

e.g. Accept:application/xml

Depending upon the accepted header value in the request, the server sends the response. This is called as Content Negotiation.

Now, if don’t set any Accept header, then by default Web API returns JSON data

**===What is MediaTypeFormatter**

MediaTypeFormatter is an abstract class from which JSONMediaTypeFormatter & XMLMediaTypeFormatter inherites.

**===How to return only JSON from Asp.net Web API** Service.config.Formatters.Remove(config.Formatters.XmlFormatter);

**===Post Method in WebAPI**

If method return type is void status code 204 no content is return.

Change method return type to HttpResponseMessage when new item created we return 201 status code item created.

**CreateResponse of web API method example**

Request.CreateResponse(HttpStatusCode.Created, empdat);

Request.CreateResponse(HttpStatusCode.Ok);

**===Custom Method Names in WebAPI**

Like if we start method with get then it automatically take as get request

HTTPVerb mapped to Get() or GetEmployees() or GetSomething()

Same for post, put & delete.

Attributes that are used to map custom named methods in a controller class to get, post, put, delete http verbs

HttpGet = Get

HttpPost = Post

HttpPut = Put

HttpDelete = Delete

**===WebAPI Query String Parameters**

Request need to made as below

<http://localhost:1033/api/employees?Name=alok>

Function need to create like this

[HttpGet]

public HttpResponseMessage LoadAllEmployees(string Name="All")

{

}

**===fromuri frombody attribute**

When the ASP.NET Web API calls a method on a controller, it must set values for the parameters, a process called parameter binding.

By default, Web API uses the following rules to bind parameters:

If the parameter is a "simple" type, Web API tries to get the value from the URI. Simple types include the .NET primitive types (int, bool, double, and so forth), plus TimeSpan, DateTime, Guid, decimal, and string, plus any type with a type converter that can convert from a string.

For complex types (like customer, employee objects), Web API tries to read the value from the message body, using a media-type formatter.

So, if you want to override the above default behaviour and force Web API to read a complex type from the URI, add the [FromUri] attribute to the parameter. To force Web API to read a simple type from the request body, add the [FromBody] attribute to the parameter.

So, to answer your question, the need of the [FromBody] and [FromUri] attributes in Web API is simply to override, if necessary, the default behaviour as described above.

To change the default parameter binding process use fromuri frombody attribute

public HttpResponseMessage PutEmployee([FromBody] int id,[FromUri] Empdata emp)

And request sample

<http://localhost:1033/api/employees?Name=DigVijay&Salary=50000>

use [FromBody] attribute to force web api to get simple types from the request body

use [FromURI] attribute to force web api to get complex type from the URI.

**===Ajax Call from Web API**

Example

$(document).ready(function () {

var ulemplist= $('#unorderedlist');

$("#btn").click(function () {

$.ajax({

type: 'GET',

url: 'api/employees',

datatype: 'json',

success: function (data) {

ulemplist.empty();

$.each(data, function (index, val) {

var FullName = val.Name;

ulemplist.append('<li>' + FullName + '</li>');

});

}

});

});

**===Cross Domain Ajax Call From Web API**

For security reasons browsers doesn't allow cross domain ajax request

we can get around this problem by using 2 ways

Using JSONP(json with padding)

Enable CORS(cross origin resource sharing)

We can call callback like this in JSONP as below()

<http://localhost:1033/api/employees?callback=abc>

to get result wrap in function while doing cross page domain request.

/\*\*/ typeof abc === 'function' && abc({"EmpNo":11,"Name":"DigVijay","Salary":9999});

**===by using CORS**

When the Request comes from some other domain (domain1.com), it throws an error. It means, the browser has a property called Access-Control-Allow-Origin which restricts the requests from different domains for security purposes. So, we need to enable CORS to accomplish the request

CORS stands for Cross-Origin Resource Sharing. It is a mechanism that allows restricted resources on a web page to be requested from another domain, outside the domain from which the resource originated.

To use Microsoft CORS package, you need to install from NuGet package.

Go to Tools Menu-> Library Package Manager -> Package Manager Console -> execute the below command.

Install-Package Microsoft.AspNet.WebApi.Cors

**You can configure CORS support for Web API at three levels.**

Global level

Controller level

Action level

**Then in WebAPIConfig.cs register method**

//Configure Cors for cross page request of ajax for all domain

EnableCorsAttribute cors = new EnableCorsAttribute("\*", "\*", "\*");

config.EnableCors(cors);

or //Configure Cors for cross page request of ajax for Specific level

EnableCorsAttribute cors = new EnableCorsAttribute("http://localhost:80/DemoApp/WebForm1.aspx", "\*", "GET,POST");

config.EnableCors(cors);

**We can allow by using Web Config. Too**

<httpProtocol>

<customHeaders>

<!-- Adding the following custom HttpHeader

will help prevent CORS from stopping the Request-->

<add name="Access-Control-Allow-Origin" value="\*" />

</customHeaders>

</httpProtocol>

**At controller level**

[EnableCors(origins: "\*", headers: "\*", methods: "\*", exposedHeaders: "X-My-Header")]

public class TestController : ApiController

{

}

**===Basic Authentication in WebAPI**

<http://www.c-sharpcorner.com/blogs/basic-authentication-in-webapi>

Nowadays, WebAPI is a trending technology. As we are exposing our WebAPI to the outside world, we should maintain security in WebAPI. It means a valid user can only access WebAPI, or else it will throw un-authorization error.

add a class, which will used as Authorization filter. The class BasicAuthenticationAttribute inherits from BasicAuthenticationAttribute abstract class.

Our authorization filter is ready and we need to register it. You can register at global, controller or action level. Here, we have added for controller level.

You can add the filter in either WebApiConfig or FilterConfig class file.

In WebApiConfig.cs

config.Filters.Add(new BasicAuthenticationAttribute());

In FilterConfig.cs

filters.Add(new BasicAuthenticationAttribute());

[BasicAuthentication]

public class BlogController : ApiController

{

}

**===AJAX request to call WebAPI with Basic authentication**

$.ajax({

type: 'GET',

url: 'api/employees',

datatype: 'json',

//This is For Basic Authentication

headers: { 'Authorization': 'Basic ' + btoa(username + ':' + password) },

success: function (data) {

ulemplist.empty();

$.each(data, function (index, val) {

var FullName = val.Name;

var Gender = val.Gender;

ulemplist.append('<li>' + FullName + ' ('+Gender+') </li>');

});

},

complete: function (jqXHR) {

if(jqXHR.status=='401')

{

ulemplist.empty();

ulemplist.append('<li style="color:red">' + jqXHR.status + ' : ' + jqXHR.statusText + '</li>');

}

}

**===Token Based Authentication Using ASP.Net Web API, OWIN and Identity with Entity Framework**

The Open Web Interface for .NET (OWIN) defines a standard interface between .NET web servers and web applications. The OAuth authorization framework enables a third-party application to obtain limited access to a HTTP service.

The standard way to authenticate via Web API is to use token-based authentication. We pass the username and password across in the request. If authentication is successful, the server passes a token back in the response. We then include that token in later requests. If the token is not present, the server issues a 401 unauthorised response.

**===what you need to do is follow these steps**

Create a Web API project with Individual User Accounts Authentication.

Now, you will have all ready to use API's for Register, change password, as well as API endpoint to generate token for a user.

Create another project but this time it is MVC with No Authentication in same solution.

**===we can post following request to generate token**

<http://localhost:3376/token>

and Request body contains username=alok&password=alok123&grant\_type=password

which generate access token. Code to generate access token is inside App\_Start\Startup.Auth.cs file in web api

$('#btnLogin').click(function () {

$.ajax({

// Post username, password & the grant type to /token

url: '/token',

method: 'POST',

contentType: 'application/json',

data: {

username: $('#txtUsername').val(),

password: $('#txtPassword').val(),

grant\_type: 'password'

},

// When the request completes successfully, save the

// access token in the browser session storage and

// redirect the user to Data.html page. We do not have

// this page yet. So please add it to the

// EmployeeService project before running it

success: function (response) {

sessionStorage.setItem("accessToken", response.access\_token);

window.location.href = "Data.html";

},

// Display errors if any in the Bootstrap alert <div>

error: function (jqXHR) {

$('#divErrorText').text(jqXHR.responseText);

$('#divError').show('fade');

}

});

});

We can set it in session as follows

1. sessionStorage data is lost when the browser window is closed.

2. To store an item in the browser session storage use setItem() method

Example : sessionStorage.setItem("accessToken", response.access\_token)

3. To retrieve an item from the browser session storage use getItem() method

Example : sessionStorage.getItem("accessToken")

4. To remove an item from the browser session storage use removeItem() method

Example : sessionStorage.removeItem('accessToken')

I think there is some confusion about the difference between MVC and Web Api. In short, for MVC you can use a login form and create a session using cookies. For Web Api there is no session. That's why you want to use the token.

**===There are 2 ways for the user to Log Off**

1. By closing the browser window. Since we are storing the access token in browser session storage, the access token will be lost when we close the browser window.

2. By clicking the "Log Off" button, which explicitly removes the access token from the browser session storage.

If you do not want to loose the access token, when the browser is closed store the access token in browser local storage instead of session storage. The way you store, retrieve and remove items from local storage is exactly the same as storing, retrieving and removing items from session storage, except that you use localStorage object instead of sessionStorage object.

localStorage.setItem('accessToken');

localStorage.getItem('accessToken');

localStorage.removeItem('accessToken');

**===Attribute routing in ASP NET Web API**

In Web API 1, we had **convention-based routing** that define routes using route templates. When we create a new Web API project using Visual Studio, a default route is created in **WebApiConfig.cs** file. The default route is shown below

config.Routes.MapHttpRoute(

    name: "DefaultApi",

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

    defaults: new { id = RouteParameter.Optional }

);

So with the above default route and **StudentsController** in place   
/api/students is mapped to Get() method in StudentsController  
  
When we navigate to /api/students/1 we get the following exception message  
Multiple actions were found that match the request:   
   Get on type WebAPI.Controllers.StudentsController   
   GetStudentCourses on type WebAPI.Controllers.StudentsController  
  
This is because the framework does not know which of the 2 following action methods to map to the URI /api/students/1  
Get(int id)  
GetStudentCourses(int id)  
  
This can be very easily resolved by using **Attribute Routing**. Here is what we want the framework to do  
1. URI /api/students/1 should be mapped to **Get(int id)**. This method returns the student by id.  
2. URI /api/students/1/courses should be mapped to **GetStudentCourses(int id)**. This method returns the student courses by student id.  
  
To achieve the above, simply decorate GetStudentCourses() action method with the below [Route] attribute

[Route("api/students/{id}/courses")]

At this point build the solution and navigate to /api/students/1. Notice you now get student details whose id=1. When you navigate to /api/students/1/courses you get all the courses into which student with id=1 is enrolled.  
  
**What is Attribute Routing**  
Using the [Route] attribute to define routes is called Attribute Routing  
  
**What are the advantages of using Attribute Routing**  
Attribute routing gives us more control over the URIs than convention-based routing. Creating URI patterns like hierarchies of resources (For example, students have courses, Departments have employees) is very difficult with convention-based routing. With attribute routing all you have to do is use the [Route] attribute as shown below.  
  
[Route("api/students/{id}/courses")]  
  
**How to enable Attribute Routing**  
In ASP.NET Web API 2, Attribute Routing is enabled by default. The following line of code in WebApiConfig.cs file enables Attribute Routing.  
  
config.MapHttpAttributeRoutes();  
  
**Can we use both Attribute Routing and Convention-based routing in a single Web API project**  
Yes, both the routing mechanisms can be combined in a single Web API project. The controller action methods that have the [Route] attribute uses Attribute Routing, and the others without [Route] attribute uses Convention-based routing.

**===RoutePrefix attribute in Web API**

The common prefix "api/students" can be specified for the entire controller using the [RoutePrefix] attribute as shown below. This eliminates the need to repeat the common prefix "api/students" on every controller action method.

[RoutePrefix("api/students")]

public class StudentsController : ApiController

{

    [Route("api/students")]

    public IEnumerable<Student> Get()

    [Route("api/students/{id}")]

    public Student Get(int id)

    [Route("api/students/{id}/courses")]

    public IEnumerable<string> GetStudentCourses(int id)

}

If we add GetTeachers() Method in student controller and we need to navigate using /api/teachers then

there is definitely a need to override the RoutePrefix used on the StudentsController. To override the RoutePrefix use **~** as shown below

[Route("~/api/teachers")]

public IEnumerable<Teacher> GetTeachers()

{

    List<Teacher> teachers = new List<Teacher>()

    {

        new Teacher() { Id = 1, Name = "Rob" },

        new Teacher() { Id = 2, Name = "Mike" },

        new Teacher() { Id = 3, Name = "Mary" }

    };

    return teachers;

}

**What is the use of RoutePrefix attribute**  
RoutePrefix attribute is used to specify the common route prefix at the controller level to eliminate the need to repeat that common route prefix on every controller action method  
  
**How to override the route prefix**  
Use ~ character to override the route prefix

**===Web API attribute routing constraints**

* If an integer is specified in the URI (/api/students/1), then we want the Get(int id) method that has integer parameter invoked
* If a string is specified in the URI (/api/students/Sam), then we want the Get(string name) method that has string parameter invoked

This can be very easily achieved using Route Constraints as shown below. To specify route constraint, the syntax is "{parameter:constraint}". With these constraints in place, if the parameter segment in the URI is an integer, then Get(int id) method with integer parameter is invoked, if it is a string then Get(string name) method with string parameter is invoked.  
  
Please note that **"alpha"** stands for uppercase or lowercase alphabet characters. Along with int and alpha, we also have constraints like decimal, double, float, long, bool etc. Please check MSDN for the full list of available constraints.

[Route("{id:int}")]

public Student Get(int id)

{

    return students.FirstOrDefault(s => s.Id == id);

}

[Route("{name:alpha}")]

public Student Get(string name)

{

    return students.FirstOrDefault(s => s.Name.ToLower() == name.ToLower());

}

Some of the constraints take arguments. To specify arguments use parentheses as shown below.

|  |  |  |
| --- | --- | --- |
| **Constraint** | **Description** | **Example** |
| min | Matches an integer with a minimum value | {x:min(0)} |
| max | Matches an integer with a maximum value | {x:max(100)} |
| length | Matches a string with the specified length or within a specified range of lengths | {x:length(3)} {x:length(1,10)} |
| minlength | Matches a string with a minimum length | {x:minlength(1)} |
| maxlength | Matches a string with a maximum length | {x:maxlength(100)} |
| range | Matches an integer within a range of values | {x:range(1,100)} |

**Example  :** If you want Get(int id) method to be mapped to URI /api/students/{id}, only if id is a number greater than ZERO, then use the "min" constraint as shown below.

[Route("{id:int:min(1)}")]

public Student Get(int id)

{

    return students.FirstOrDefault(s => s.Id == id);

}

With the above change, if you specify a positive number like 1 in the URI, then it will be mapped to Get(int id) method as expected  
/api/students/1  
  
However, if you specify 0 or a negative number less than ZERO, you will get an error. For example if you specify 0 as the value for id in the URI, you will get  
No HTTP resource was found that matches the request URI 'http://localhost:65116/api/students/0'  
  
Along with the "min" constraint you can also specify "max" constraint as shown below. For example if you want the id value in the URI to be between 1 and 3 inclusive, then you can specify both "min" and "max" constraints as shown below.

[Route("{id:int:min(1):max(3)}")]

public Student Get(int id)

{

    return students.FirstOrDefault(s => s.Id == id);

}

The above example can also be achieved using just the "range" attribute as shown below

[Route("{id:int:range(1,3)}")]

public Student Get(int id)

{

    return students.FirstOrDefault(s => s.Id == id);

}

**===IHttpActionResult vs HttpResponseMessage**

If we remember the HTTP response creation of Web API 1.0 we used to use write 3 to 4 lines of code to create one full fledge HTTP response by setting the status code and media type with an appropriate message. The style is something like this.

This code snippet will return one Unauthorized HTTP response (haha.. not an unauthorized response, but HTTP response of unauthorized type) asynchronously. Now, if we want to change the response type then obviously we must change the status code.  
  
Fine and simple but it is simpler in the Web API 2. We can create the same kind of response with a single line of code.

In Web API 1, we have HttpResponseMessage type that a controller action method returns. A new type called "IHttpActionResult" is introduced in Web API 2 that can be returned from a controller action method. Instead of returning HttpResponseMessage from a controller action, we can now return IHttpActionResult. There are 2 main advantages of using the IHttpActionResult interface.

1. The code is cleaner and easier to read
2. Unit testing controller action methods is much simpler. We will discuss,

**In addition to Ok() and NotFound() helper methods**, we have the following methods that we can use depending on what we want to return from our controller action method. All these methods return a type, that implements IHttpActionResult interface.

* BadRequest(),Conflict(),Created(),InternalServerError(),Redirect()
* Unauthorized()

**======Web API versioning**

**Why is versioning required in Web API?**

* Once a Web API service is made public, different client applications start using your Web API services.
* As the business grows and requirements change, we may have to change the services as well, but the changes to the services should be done in way that does not break any existing client applications.
* This is when Web API versioning helps. We keep the existing services as is, so we are not breaking the existing client applications, and develop a new version of the service that new client applications can start using.

**Different options available to version Web API services :** Versioning can be implemented using  
1. URI's  
2. Query String  
3. Version Header  
4. Accept Header  
5. Media Type

**1. With URI versioning**

There are two types

**a. convention-based routing**

In **WebApiConfig.cs** file we have the following route.

config.Routes.MapHttpRoute(

    name: "Version1",

    routeTemplate: "api/v1/Students/{id}",

    defaults: new { id = RouteParameter.Optional, controller = "StudentsV1" }

);

**b. Attribute Routing**

public class StudentsV1Controller : ApiController

{

    [Route("api/v1/students")]

    public IEnumerable<StudentV1> Get() {...}

    [Route("api/v1/students/{id}")]

    public StudentV1 Get(int id) {...}

}

**2. With QueryString versioning**

In Web API, there is a class called DefaultHttpControllerSelector. This class has a method called SelectController() that selects the controller based on the information it has in the URI.

So from the URI, the SelectController() method takes the name of the controller in this case "Students" and finds "StudentsController" and returns it. This is the default implementation that we get out of the box.

**Step 1 :** Since the default controller selector implementation provided by Web API does not work for us, we have to provide our own custom controller selector implementation. To do this  
**1.** Add a folder to the web api project. Name it "Custom"  
**2.** Add a class file to the folder. Name it "CustomControllerSelector"

& override below method with custom controller selection.

public override HttpControllerDescriptor

            SelectController(HttpRequestMessage request)

The next thing that we need to do is, replace the default controller selector with our custom controller selector. This is done in WebApiConfig.cs file. Notice we are replacing IHttpControllerSelector, with our CustomControllerSelector. DefaultHttpControllerSelector implements IHttpControllerSelector, so that is the reason we are replacing IHttpControllerSelector.

config.Services.Replace(typeof(IHttpControllerSelector),

    new CustomControllerSelector(config));

**3. With custom header versioning**

// Get the version number from Custom version header

            string customHeader = "X-StudentService-Version";

            if (request.Headers.Contains(customHeader))

            {

                versionNumber = request.Headers.GetValues(customHeader).FirstOrDefault();

            }