**ASP.NET Web API**

**What is ASP.NET Web API ?**  
The term API stands for ‘Application Programming Interface’. ASP.NET Web API is a framework for building Web API’s, i.e. HTTP based services on top of the .NET Framework. The most common use case for using Web API is for building RESTful services. These services can then be consumed by a broad range of clients like  
1. Browsers  
2. Mobile applications  
3. Desktop applications  
4. IOTs

**What are RESTful services**  
REST stands for Representational State Transfer. REST was first introduced in the year 2000 by Roy Fielding as part of his doctoral dissertation. REST is an architectural pattern for creating an API that uses HTTP as its underlying communication method. The REST architectural pattern specifies a set of constraints that a system should adhere to. Here are the REST constraints.

**Client Server constraint** - This is the first constraint. Client sends a request and the server sends a response. This separation of concerns supports the independent evolution of the client-side logic and server-side logic.

**Stateless constraint** - The next constraint is the stateless constraint. The communication between the client and the server must be stateless between requests. This means we should not be storing anything on the server related to the client. The request from the client should contain all the necessary information for the server to process that request. This ensures that each request can be treated independently by the server.  
  
**Cacheable constraint** - Some data provided by the server like list of products, or list of departments in a company does not change that often. This constraint says that let the client know how long this data is good for, so that the client does not have to come back to the server for that data over and over again.  
  
**Uniform Interface**- The uniform interface constraint defines the interface between the client and the server. To understand the uniform interface constraint, we need to understand what a resource is and the HTTP verbs - GET, PUT, POST & DELETE. In the context of a REST API, resources typically represent data entities. Product, Employee, Customer etc are all resources. The HTTP verb (GET, PUT, POST, DELETE) that is sent with each request tells the API what to do with the resource. Each resource is identified by a specific URI (Uniform Resource Identifier). The following table shows some typical requests that you see in an API

| **Resource** | **Verb** | **Outcome** |
| --- | --- | --- |
| /Employees | GET | Gets list of employees |
| /Employee/1 | GET | Gets employee with Id = 1 |
| /Employees | POST | Creates a new employee |
| /Employee/1 | PUT | Updates employee with Id = 1 |
| /Employee/1 | DELETE | Deletes employee with Id = 1 |

Another concept related to Uniform Interface is **HATEOAS**. HATEOAS stands for Hypermedia as the Engine of Application State. All this means is that in each request there will be set of hyperlinks that let's you know what other actions can be performed on the resource.

There are 2 other constraints as well  
Layered System  
Code on Demand (optional)

WCF (Windows Communication Foundation) - One of the choices available in .NET for creating RESTful services is WCF. The problem with WCF is that, a lot of configuration is required to turn a WCF service into a RESTful service. The more natural choice for creating RESTful services is ASP.NET Web API, which is specifically created for this purpose.

WCF is more suited for building services that are transport/protocol independent.

There is nothing wrong to use WCF to create RESTful services. It's just that it's a bit more complex and configuration can be a headache. If you are stuck with .NET 3.5 or you have an existing SOAP service you must support but want to add REST to reach more clients, then use WCF.   
  
If you don't have the limitation of .NET 3.5 and you want to a create brand new restful service then use ASP.NET Web API.

**Controller:**

Notice with in the **Controllers** folder we have **ValuesController** which inherits from ApiController class that is present in System.Web.Http namespace. This is different from the MVC controller. The MVC Controller class inherits from the Controller class that is present in System.Web.Mvc namespace. The HomeController class which is an MVC controller inherits from the Controller class.  
  
2. Notice in the **ValuesController** class we have methods (Get, Put, Post & Delete) that map to the HTTP verbs (GET, PUT, POST, DELETE) respectively. We have 2 overloaded versions of Get() method - One without any parameters and the other with id parameter. Both of these methods respond to the **GET** http verb depending on whether the id parameter is specified in the URI or not.

3. Now let's look at the default route that is in place for our Web API project. We have the Application\_Start() event handler In Global.asax file. This event is raised when the application starts. In the Application\_Start() event handler method we have configuration for Filters, Bundles etc. The one that we are interested in is the configuration for our Web API project, which is in WebApiConfig.Register() method. Right click on WebApiConfig.Register and select "Go To Definition" from the context menu. This will take you to the Register() method in the WebApiConfig class. This class is in App\_Start folder.  
  
4. In the Register() method we have the default route configured for our Web API project. Web API routes are different from the MVC routes. You can find the MVC routes in RouteConfig.cs file in App\_Start folder.

5. The default Web API route starts with the word api and then / and then the name of the controller and another / and an optiontion id parameter.  
"api/{controller}/{id}"  
  
6. At this point if we use the following URI in the browser, we get an error - Authorization has been denied for this request.  
http://localhost/api/values  
  
7. To get rid of this error, comment Authorize attribute on the **ValuesController** class. This is related to security which we will discuss in a later video.  
  
8. Now if you visit, http://localhost/api/values, you should see the following XML as the result

<ArrayOfstring xmlns:i="http://www.w3.org/2001/XMLSchema-instance"

               xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">

<string>value1</string>

<string>value2</string>

</ArrayOfstring>

9. Let us understand what is going on here. The name of the controller is values. So if we use a URI http://localhost:portnumber/api/values, then the web api is going to look for a controller with name **Values + the word controller** in your project. So if you have specified values in the URI it is going to look for **ValuesController**, if you specify **Products**, then it is going to look for **ProductsController**. This is all by convention and works this way out of the box.  
  
10. In a real world application this might be the domain name, for example  
http://pragimtech.com/api/values  
  
11. The browser is issuing a GET request which maps to the Get() method in the **ValuesController** class. The GET() in the values controller is returning value1 and value2 which is what we see in the browser.  
  
12. We have another overload of GET() method which takes Id parameter. Remember with the default route, the id parameter is optional. That is the reason we are able to call the GET method with or without the Id parameter. If the id parameter is specified in the URI, then the Get() method with the parameter in values controller is called

13. If a controller with the specified name is not found you will get an error. For example, in your project if you comment the ValuesController class in your project and then use the URI /api/values you will get the following error

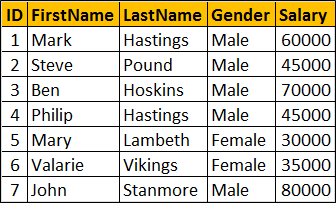
### HTTP GET PUT POST DELETE

C - Create a row  
R - Read a row  
U - Update a row  
D - Delete a row  
  
In the context of an ASP.NET Web API resource these 4 actions correspond to GET, POST, PUT and DELETE as shown in the table below

| **CRUD** | **HTTP Verb** |
| --- | --- |
| Create | POST |
| Read | GET |
| Update | PUT |
| Delete | DELETE |

**Request Verbs :** These HTTP verbs (GET, POST, PUT & DELETE) describe what should be done with the resource.

**Request Header :** When a client sends request to the server, the request contains a header and a body. The request header contains additional information such as what type of response is required. For example, do you want the response to be in XML or JSON.  
  
**Request Body :** Request Body contains the data to send to the server. For example, a POST request contains the data for the new item that you want to create. The data format may be in XML or JSON.  
  
**Response Body :** The Response Body contains the data sent as response from the server. For example, if the request is for a specific product, the response body includes product details either in XML or JSON format.  
  
**Response Status codes :** These are the HTTP status codes, that give the client details on the status of the request. Some of the common status codes are 200/OK, 404/Not Found, 204/No Content.

The Web API **EmployeeService**that we will be building will retrieve and return the data from the following SQL Server Employees table. We will be using Entity Framework to retrieve data from the database. You can use any technology of your choice to retrieve data from the database. For example you can even use raw ADO.NET.  


Create Database EmployeeDB

Go

Use EmployeeDB

Go

Create table Employees

(

     ID int primary key identity,

     FirstName nvarchar(50),

     LastName nvarchar(50),

     Gender nvarchar(50),

     Salary int

)

Go

Insert into Employees values ('Mark', 'Hastings', 'Male', 60000)

Insert into Employees values ('Steve', 'Pound', 'Male', 45000)

Insert into Employees values ('Ben', 'Hoskins', 'Male', 70000)

Insert into Employees values ('Philip', 'Hastings', 'Male', 45000)

Insert into Employees values ('Mary', 'Lambeth', 'Female', 30000)

Insert into Employees values ('Valarie', 'Vikings', 'Female', 35000)

Insert into Employees values ('John', 'Stanmore', 'Male', 80000)

Go

**Content Negotiation in Web API**.

One of the standards of the RESTful service is that, the client should have the ability to decide in which format they want the response - XML, JSON etc. A request that is sent to the server includes an Accept header. Using the Accept header the client can specify the format for the response. For example  
  
Accept: application/xml returns XML  
Accept: application/json returns JSON

Depending on the Accept header value in the request, the server sends the response. This is called Content Negotiation.

**So what does the Web API do when we request for data in a specific format**  
The Web API controller generates the data that we want to send to the client. For example, if you have asked for list of employees. The controller generates the list of employees, and hands the data to the Web API pipeline which then looks at the Accept header and depending on the format that the client has requested, Web API will choose the appropriate formatter. For example, if the client has requested for XML data, Web API uses XML formatter. If the client has requested for JSON data, Web API uses JSON formatter. These formatters are called Media type formatters.

ASP.NET Web API is greatly extensible. This means we can also plugin our own formatters, for custom formatting the data.  
  
Multiple values can also be specified for the Accept header.

If you don't specify the Accept header, by default the Web API returns JSON data.

**What is MediaTypeFormatter**  
MediaTypeFormatter is an abstract class from which JsonMediaTypeFormatter and XmlMediaTypeFormatter classes inherit from. JsonMediaTypeFormatter handles JSON and XmlMediaTypeFormatter handles XML.

**How to return only JSON from ASP.NET Web API Service irrespective of the Accept header value**  
Include the following line in **Register**() method of WebApiConfig.cs file in **App\_Start**folder. This line of code completely removes XmlFormatter which forces ASP.NET Web API to always return JSON irrespective of the Accept header value in the client request. Use this technique when you want your service to support only JSON and not XML.   
  
With this change, irrespective of the Accept header value (application/xml or application/json), the Web API service is always going to return JSON.

config.Formatters.Remove(config.Formatters.XmlFormatter);

We always get xml in browser whenever we issue a request.

**How to return JSON instead of XML from ASP.NET Web API Service when a request is made from the browser.**

**Approach 1 :**Include the following line in Register() method of WebApiConfig.cs file in App\_Start folder. This tells ASP.NET Web API to use JsonFormatter when a request is made for text/html which is the default for most browsers. The problem with this approach is that Content-Type header of the response is set to text/html which is misleading.

config.Formatters.JsonFormatter.SupportedMediaTypes

    .Add(new MediaTypeHeaderValue("text/html"));

**Approach 2 :** Include the following class in WebApiConfig.cs file in App\_Start folder. 

public class CustomJsonFormatter : JsonMediaTypeFormatter

{

    public CustomJsonFormatter()

    {

        this.SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/html"));

    }

    public override void SetDefaultContentHeaders(Type type, HttpContentHeaders headers, MediaTypeHeaderValue mediaType)

    {

        base.SetDefaultContentHeaders(type, headers, mediaType);

        headers.ContentType = new MediaTypeHeaderValue("application/json");

    }

}

**Register the formatter:** Place the following line in Register() method of WebApiConfig.cs file in App\_Start folder  
config.Formatters.Add(new CustomJsonFormatter());

### Implementing post method in ASP.NET Web API

We want to add a new Employee to the Employees table. Include the following Post() method in EmployeesController. Notice the Employee object is being passed as parameter to the Post method. The Employee parameter is decorated with [FromBody] attribute. This tells Web API to get employee data from the request body.

public void Post([FromBody] Employee employee)

{

    using (EmployeeDBEntities entities = new EmployeeDBEntities())

    {

        entities.Employees.Add(employee);

        entities.SaveChanges();

    }

}

This works fine and adds the employee to the database as expected. The problem here is that since the return type of the Post() method is void, we get status code of 204 No Content. When a new item is created, we should actually be returning status code 201 Item Created. With 201 status code we should also include the location i.e URI of the newly created item. To achieve this, change the implementation of the Post() method as shown below.

public HttpResponseMessage Post([FromBody] Employee employee)

{

    try

    {

        using (EmployeeDBEntities entities = new EmployeeDBEntities())

        {

            entities.Employees.Add(employee);

            entities.SaveChanges();

            var message = Request.CreateResponse(HttpStatusCode.Created, employee);

            message.Headers.Location = new Uri(Request.RequestUri +

                employee.ID.ToString());

            return message;

        }

    }

    catch (Exception ex)

    {

        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ex);

    }

}

public HttpResponseMessage Get(int id)

{

    using (EmployeeDBEntities entities = new EmployeeDBEntities())

    {

        var entity = entities.Employees.FirstOrDefault(e => e.ID == id);

        if (entity != null)

        {

            return Request.CreateResponse(HttpStatusCode.OK, entity);

        }

        else

        {

            return Request.CreateErrorResponse(HttpStatusCode.NotFound,

                "Employee with Id " + id.ToString() + " not found");

        }

    }

}

**Here are the important points to remember**

* If a method return type is void, by default status code 204 No Content is returned.
* When a new item is created, we should be returning status code 201 Item Created.
* With 201 status code we should also include the location i.e URI of the newly created item.
* When an item is not found, instead of returning NULL and status code 200 OK, return 404 Not Found status code along with a meaningful message such as "Employee with Id = 101 not found"

### Implementing Delete method in ASP.NET Web API

public void Delete(int id)

{

    using (EmployeeDBEntities entities = new EmployeeDBEntities())

    {

        entities.Employees.Remove(entities.Employees.FirstOrDefault(e => e.ID == id));

        entities.SaveChanges();

    }

}

public HttpResponseMessage Delete(int id)

{

    try

    {

        using (EmployeeDBEntities entities = new EmployeeDBEntities())

        {

            var entity = entities.Employees.FirstOrDefault(e => e.ID == id);

            if (entity == null)

            {

                return Request.CreateErrorResponse(HttpStatusCode.NotFound,

                    "Employee with Id = " + id.ToString() + " not found to delete");

            }

            else

            {

                entities.Employees.Remove(entity);

                entities.SaveChanges();

                return Request.CreateResponse(HttpStatusCode.OK);

            }

        }

    }

    catch (Exception ex)

    {

        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ex);

    }

}

### Implementing PUT method in ASP.NET Web API

public void Put(int id, [FromBody]Employee employee)

{

    using (EmployeeDBEntities entities = new EmployeeDBEntities())

    {

        var entity = entities.Employees.FirstOrDefault(e => e.ID == id);

        entity.FirstName = employee.FirstName;

        entity.LastName = employee.LastName;

        entity.Gender = employee.Gender;

        entity.Salary = employee.Salary;

        entities.SaveChanges();

    }

}

public HttpResponseMessage Put(int id, [FromBody]Employee employee)

{

    try

    {

        using (EmployeeDBEntities entities = new EmployeeDBEntities())

        {

            var entity = entities.Employees.FirstOrDefault(e => e.ID == id);

            if (entity == null)

            {

                return Request.CreateErrorResponse(HttpStatusCode.NotFound,

                    "Employee with Id " + id.ToString() + " not found to update");

            }

            else

            {

                entity.FirstName = employee.FirstName;

                entity.LastName = employee.LastName;

                entity.Gender = employee.Gender;

                entity.Salary = employee.Salary;

                entities.SaveChanges();

                return Request.CreateResponse(HttpStatusCode.OK, entity);

            }

        }

    }

    catch (Exception ex)

    {

        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ex);

    }

}

### Custom method names in ASP.NET Web API

By default, the HTTP verb GET is mapped to a method in a controller that has the name Get() or starts with the word Get. In the following EmployeesController, the method is named Get() so by convention this is mapped to the HTTP verb GET. Even if you rename it to GetEmployees() or GetSomething() it will still be mapped to the HTTP verb GET as long as the name of the method is prefixed with the word Get. The word Get is case-insensitive. It can be lowercase, uppercase or a mix of both.

If the method is not named Get or if it does not start with the word get then Web API does not know the method name to which the GET request must be mapped and the request fails with an error message stating The requested resource does not support http method 'GET' with the status code 405 Method Not Allowed

Attributes that are used to map your custom named methods in the controller class to GET, POST, PUT and DELETE http verbs.

| **Attribute** | **Maps to http verb** |
| --- | --- |
| [HttpGet] | GET |
| [HttpPost] | POST |
| [HttpPut] | PUT |
| [HttpDelete] | DELETE |

### ASP.NET Web API query string parameters

| **Query String** | **Data** |
| --- | --- |
| http://localhost/api/employees?gender=All | All Employees |
| http://localhost/api/employees?gender=Male | Only Male Employees |
| http://localhost/api/employees?gender=Female | Only Female Employees |

public HttpResponseMessage Get(string gender = "All")

{

    using (EmployeeDBEntities entities = new EmployeeDBEntities())

    {

        switch (gender.ToLower())

        {

            case "all":

                return Request.CreateResponse(HttpStatusCode.OK,

                    entities.Employees.ToList());

            case "male":

                return Request.CreateResponse(HttpStatusCode.OK,

                    entities.Employees.Where(e => e.Gender.ToLower() == "male").ToList());

            case "female":

                return Request.CreateResponse(HttpStatusCode.OK,

                    entities.Employees.Where(e => e.Gender.ToLower() == "female").ToList());

            default:

                return Request.CreateErrorResponse(HttpStatusCode.BadRequest,

                    "Value for gender must be Male, Female or All. " + gender + " is invalid.");

        }

    }

}

### FromBody and FromUri in Web API

### Web API maps the data in the request to the PUT method parameters in the EmployeesController. This process is called Parameter Binding.

Now let us understand the default convention used by Web API for binding parameters.

1. If the parameter is a simple type like int, bool, double, etc., Web API tries to get the value from the URI (Either from route data or Query String)
2. If the parameter is a complex type like Customer, Employee etc., Web API tries to get the value from the request body

So in our case, the id parameter is a simple type, so Web API tries to get the value from the request URI. The employee parameter is a complex type, so Web API gets the value from the request body.  
  
We can change this default **parameter binding** process by using [FromBody] and [FromUri] attributes. Notice in the example below

1. We have decorated id parameter with [FromBody] attribute, this forces Web API to get it from the request body
2. We have decorated employee parameter with [FromUri] attribute, this forces Web API to get employee data from the URI (i.e Route data or Query String)

public HttpResponseMessage Put([FromBody]int id, [FromUri]Employee employee)

{

    try

    {

        using (EmployeeDBEntities entities = new EmployeeDBEntities())

        {

            var entity = entities.Employees.FirstOrDefault(e => e.ID == id);

            if (entity == null)

            {

                return Request.CreateErrorResponse(HttpStatusCode.NotFound,

                    "Employee with Id " + id.ToString() + " not found to update");

            }

            else

            {

                entity.FirstName = employee.FirstName;

                entity.LastName = employee.LastName;

                entity.Gender = employee.Gender;

                entity.Salary = employee.Salary;

                entities.SaveChanges();

                return Request.CreateResponse(HttpStatusCode.OK, entity);

            }

        }

    }

    catch (Exception ex)

    {

        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ex);

    }

}

### Here is the request from Fiddler 1. Employee data is specified in the URI using query string parameters  2. The id is specified in the request body  frombody attribute example

### Call ASP.NET Web API from jQuery

<!DOCTYPE html>

<html>

<head>

    <title></title>

    <meta charset="utf-8" />

    <script src="Scripts/jquery-1.10.2.js"></script>

    <script type="text/javascript">

        $(document).ready(function () {

            var ulEmployees = $('#ulEmployees');

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

                $.ajax({

                    type: 'GET',

                    url: "api/employees/",

                    dataType: 'json',

                    success: function (data) {

                        ulEmployees.empty();

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

                            var fullName = val.FirstName + ' ' + val.LastName;

                            ulEmployees.append('<li>' + fullName + '</li>');

                        });

                    }

                });

            });

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

                ulEmployees.empty();

            });

        });

    </script>

</head>

<body>

    <div>

        <input id="btn" type="button" value="Get All Employees" />

        <input id="btnClear" type="button" value="Clear" />

        <ul id="ulEmployees" />

    </div>

</body>

</html>

### Calling ASP.NET Web API service in a cross domain using jQuery ajax

### What is same origin policy Browsers allow a web page to make AJAX requests only with in the same domain. Browser security prevents a web page from making AJAX requests to another domain. This is called same origin policy.

### The following 2 URLs have the same origin http://localhost:1234/api/employees http://localhost:1234/Employees.html The following 2 URLs have different origins, because they have different port numbers (1234 v/s 5678) http://localhost:1234/api/employees http://localhost:5678/Employees.html The following 2 URLs have different origins, because they have different domains (.com v/s .net) http://pragimtech.com/api/employees http://pragimtech.net/Employees.html The following 2 URLs have different origins, because they have different schemes (http v/s https) https://pragimtech.com/api/employees <http://pragimtech.net/Employees.html>

There are 2 ways to get around this problem

* Using JSONP (JSON with Padding)
* Enabling CORS (Cross Origin Resource Sharing)

### So what is JSONP and what does it do? JSONP stands for JSON with Padding. All JSONP does is wraps the data in a function. So for example, if you have the following JSON object {     "FirstName" : "Mark",     "LastName"  : "Hastings",     "Gender"    : "Male", } JSONP will wrap the data in a function as shown below CallbackFunction({     "FirstName" : "Mark",     "LastName"  : "Hastings",     "Gender"    : "Male", })

### Browsers allow to consume JavaScript that is present in a different domain but not data. Since the data is wrapped in a JavaScript function, this can be consumed by a web page that is present in a different domain.

### Cross-origin resource sharing ASP.NET Web API

**enable CORS** (Cross Origin Resource Sharing) which allows cross domain ajax calls. CORS support is released with ASP.NET Web API 2.

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

config.EnableCors();

**Parameters of EnableCorsAttribute**

| **Parameter** | **Description** |
| --- | --- |
| origins | Comma-separated list of origins that are allowed to access the resource. For example "http://www.pragimtech.com,http://www.mywebsite.com" will only allow ajax calls from these 2 websites. All the others will be blocked. Use "\*" to allow all |
| headers | Comma-separated list of headers that are supported by the resource. For example "accept,content-type,origin" will only allow these 3 headers. Use "\*" to allow all. Use null or empty string to allow none |
| methods | Comma-separated list of methods that are supported by the resource. For example "GET,POST" only allows Get and Post and blocks the rest of the methods. Use "\*" to allow all. Use null or empty string to allow none |

1. There is no need to create an instance of EnableCorsAttribute in Register() method of WebApiConfig.cs file. Call the EnableCors() method without any parameter values.  
  
**config.EnableCors();**  
  
2. Apply the  EnableCorsAttribute on the controller class

[EnableCorsAttribute("\*", "\*", "\*")]

public class EmployeesController : ApiController

{

}

In the same manner, you can also apply it at a method level if you wish to do so.  
  
To disable CORS for a specific action apply [DisableCors] on that specific action  
  
When CORS is enabled, the browser sets the **origin**header of the request to the domain of the site making the request. The server sets **Access-Control-Allow-Origin** header in the response to either \* or the origin that made the request. \* indicates any site is allowed to make the request.

### ASP.NET Web API enable HTTPS

using System;

using System.Net;

using System.Net.Http;

using System.Text;

using System.Web.Http.Controllers;

using System.Web.Http.Filters;

namespace EmployeeService

{

    public class RequireHttpsAttribute : AuthorizationFilterAttribute

    {

        public override void OnAuthorization(HttpActionContext actionContext)

        {

            if (actionContext.Request.RequestUri.Scheme != Uri.UriSchemeHttps)

            {

                actionContext.Response = actionContext.Request

                    .CreateResponse(HttpStatusCode.Found);

                actionContext.Response.Content = new StringContent

                    ("<p>Use https instead of http</p>", Encoding.UTF8, "text/html");

                UriBuilder uriBuilder = new UriBuilder(actionContext.Request.RequestUri);

                uriBuilder.Scheme = Uri.UriSchemeHttps;

                uriBuilder.Port = 44337;

                actionContext.Response.Headers.Location = uriBuilder.Uri;

            }

            else

            {

                base.OnAuthorization(actionContext);

            }

        }

    }

}

**Step 2 :** Include the following line of code in **Register()** method of **WebApiConfig class** in **WebApiConfig.cs** file in **App\_Start**folder. This adds **RequireHttpsAttribute**as a filter to the filters collection. So for every request the code in this filter is executed. If the request is issued using HTTP, it will be automatically redirected to HTTPS.

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

**Please note :** If you don't want to enable HTTPS for the entire application then don't add RequireHttpsAttribute to the filters collection on the config object in the register method. Simply decorate the controller class or the action method with RequireHttpsAttribute for which you want HTTPS to be enabled. For the rest of the controllers and action methods HTTPS will not be enabled.

**Enable basic authentication**  
1. The BasicAuthenticationAttribute can be applied on a specific controller, specific action, or globally on all Web API controllers.  
  
2. To enable basic authentication across the entire Web API application, register BasicAuthenticationAttribute as a filter using the Register() method in WebApiConfig class

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

3. You can also apply the attribute on a specific controller, to enable basic authentication for all the methods in that controller  
  
4. In our case let's just enable basic authentication for Get() method in EmployeesController. Also modify the implementation of the Get() method as shown below.

[BasicAuthentication]

public HttpResponseMessage Get(string gender = "All")

{

    string username = Thread.CurrentPrincipal.Identity.Name;

    using (EmployeeDBEntities entities = new EmployeeDBEntities())

    {

        switch (username.ToLower())

        {

            case "male":

                return Request.CreateResponse(HttpStatusCode.OK,

                    entities.Employees.Where(e => e.Gender.ToLower() == "male").ToList());

            case "female":

                return Request.CreateResponse(HttpStatusCode.OK,

                    entities.Employees.Where(e => e.Gender.ToLower() == "female").ToList());

            default:

                return Request.CreateResponse(HttpStatusCode.BadRequest);

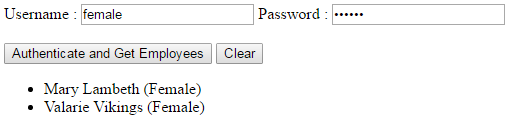
        }

    }

}

**Testing basic authentication using fiddler**  
1. The username and password need to be colon (:) separated and base64 encoded.   
  
2. Just google with the string - base64 encode. The first web site that you get is <https://www.base64encode.org/>

### Call web api service with basic authentication using jquery ajax

Depending on the credentials provided the web api service should authenticate and return the correct results. If female username and password are used only female employees should be returned. If male username and password are used only male employees should be returned.   


<!DOCTYPE html>

<html>

<head>

    <title></title>

    <meta charset="utf-8" />

    <script src="Scripts/jquery-1.10.2.js"></script>

    <script type="text/javascript">

        $(document).ready(function () {

            var ulEmployees = $('#ulEmployees');

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

                // Get the username & password from textboxes

                var username = $('#txtUsername').val();

                var password = $('#txtPassword').val();

                $.ajax({

                    type: 'GET',

                    // Make sure to change the port number to

                    // where you have the employee service

                    // running on your local machine

                    url: 'http://localhost:35171/api/Employees',

                    dataType: 'json',

                    // Specify the authentication header

                    // btoa() method encodes a string to Base64

                    headers: {

                        'Authorization': 'Basic ' + btoa(username + ':' + password)

                    },

                    success: function (data) {

                        ulEmployees.empty();

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

                            var fullName = val.FirstName + ' ' + val.LastName;

                            ulEmployees.append('<li>' + fullName + ' (' + val.Gender + ')</li>')

                        });

                    },

                    complete: function (jqXHR) {

                        if (jqXHR.status == '401') {

                            ulEmployees.empty();

                            ulEmployees.append('<li style="color:red">'

                                + jqXHR.status + ' : ' + jqXHR.statusText + '</li>')

                        }

                    }

                });

            });

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

                ulEmployees.empty();

            });

        });

    </script>

</head>

<body>

    Username : <input type="text" id="txtUsername" />

    Password : <input type="password" id="txtPassword" />

    <br /><br />

    <input id="btn" type="button" value="Authenticate and Get Employees" />

    <input id="btnClear" type="button" value="Clear" />

    <ul id="ulEmployees"></ul>

</body>

</html>