ASP.NET WEB API

* Application Programmable Interface:
* As the name suggests, it’s going to act as a programmable interface between applications, so they can connect each other for data sharing and other tasks.
* In simple terms, it consists of functions/sub routines which can be accessed by programmers to access specific feature of an application/data/operating system or other services.
* Web API:
* As the name suggests, an API over web, which can be accessed via **HTTP** protocol.
* It is a concept but not a technology, can be implemented in .Net, Java.
* ASP.NET WEB API:
* A framework used to build API’s to connect with other applications.
* It works similar to ASP.NET MVC except that it sends the data in a response instead of html view.
* It is like Web Service or WCF except it supports only HTTP protocol.

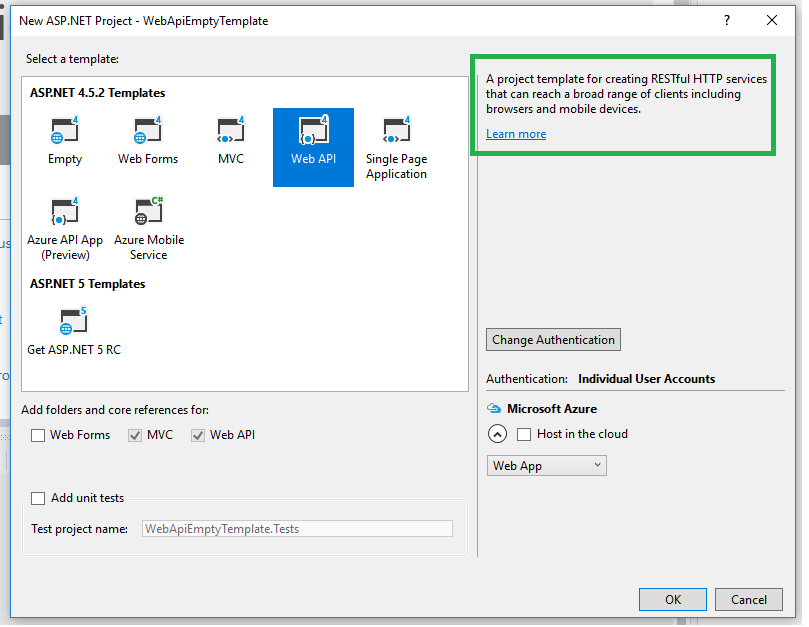
|  |
| --- |
| <https://www.tutorialsteacher.com/webapi/what-is-web-api>  https://www.tutorialsteacher.com/Content/images/webapi/webapi-overview.png |

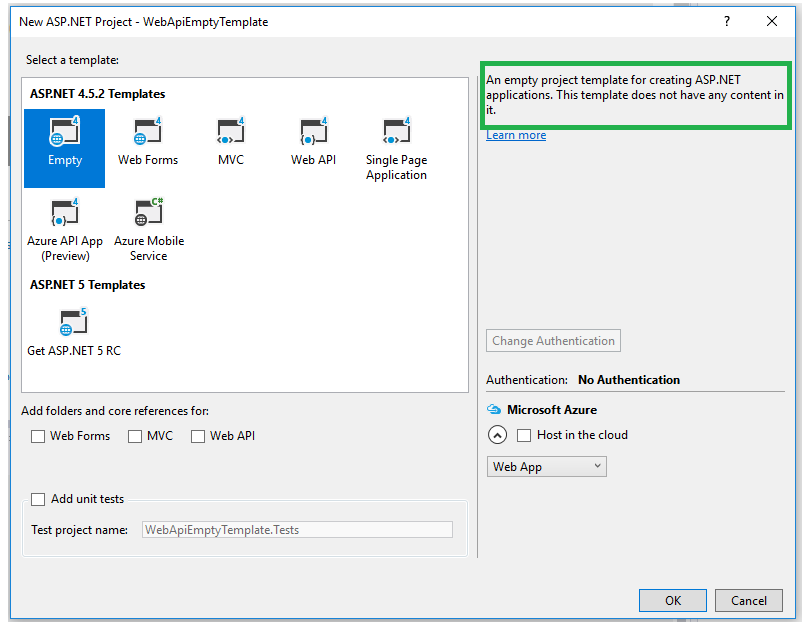
* Create Web API Project:

File 🡪 New 🡪 Project 🡪 Visual C# 🡪 Web 🡪 *Select* ASP.NET Web Application 🡪 *Select* Web API *template* 🡪 Create

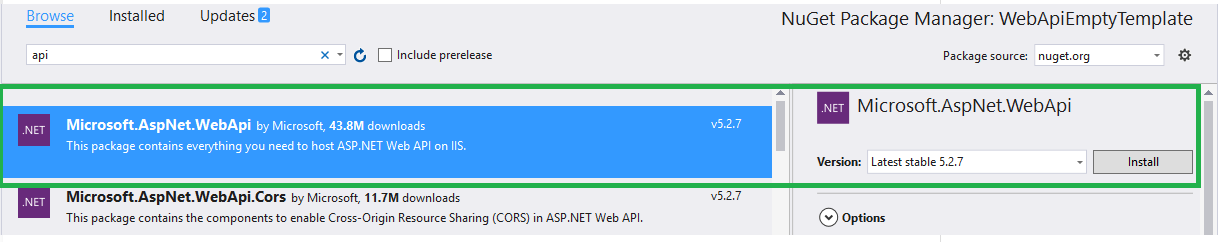
We can create Web API Project in two ways.

1. Web API with MVC project. 🡪 Above steps (Creates MVC controller as well API controller files)



1. Stand-alone Web API project. 🡪 (Need to create empty template and need to install Nuget package **Microsoft.AspNet.WebApi** (screenshot below))

Nuget package **Microsoft.AspNet.WebApi:**



* Then add manually controller cs and Web API config file with the routing configurations.
* In Global.asax.cs, call the register of WebAPIconfig.cs file.
* Test Web API:

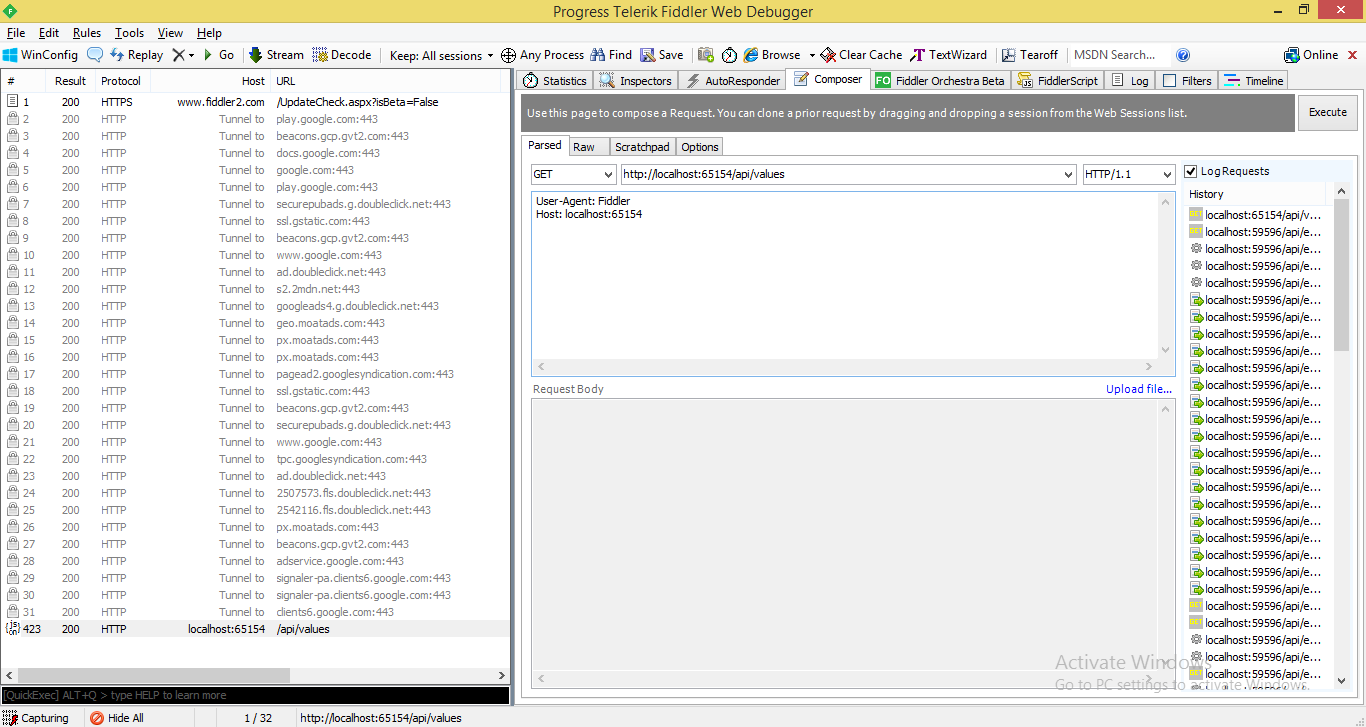
We can use below tools to test API.

1) Fiddler

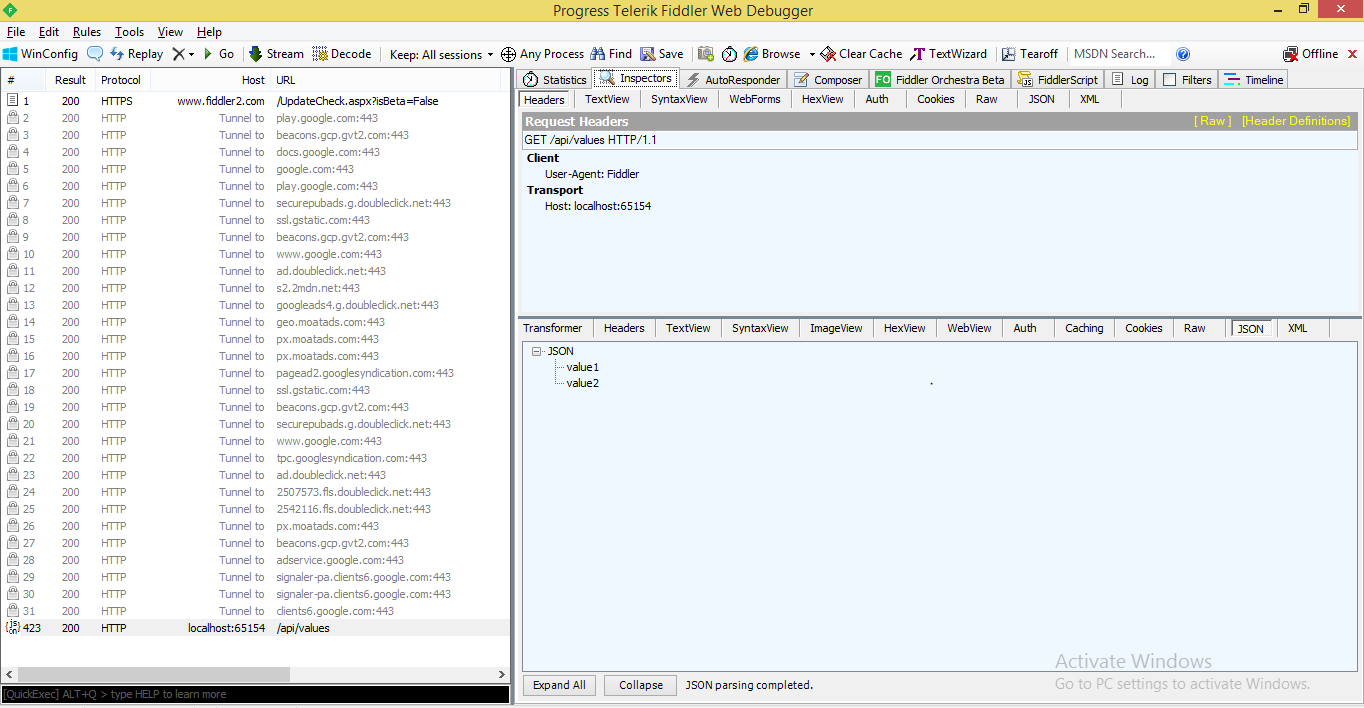
2) Postman

* We can compose the http request using the composer tab and by clicking execute, resource (API method) will be executed/called and the response will be shown in left pane.
* Double clicking on that will open Inspectors tab, shows the request details like request headers (http method GET/POST/PUT/DELETE) and response details like response headers (response code 200 OK or 404 Not Found), result in json/xml format. (screenshots below)

**Composer**



**Inspectors**

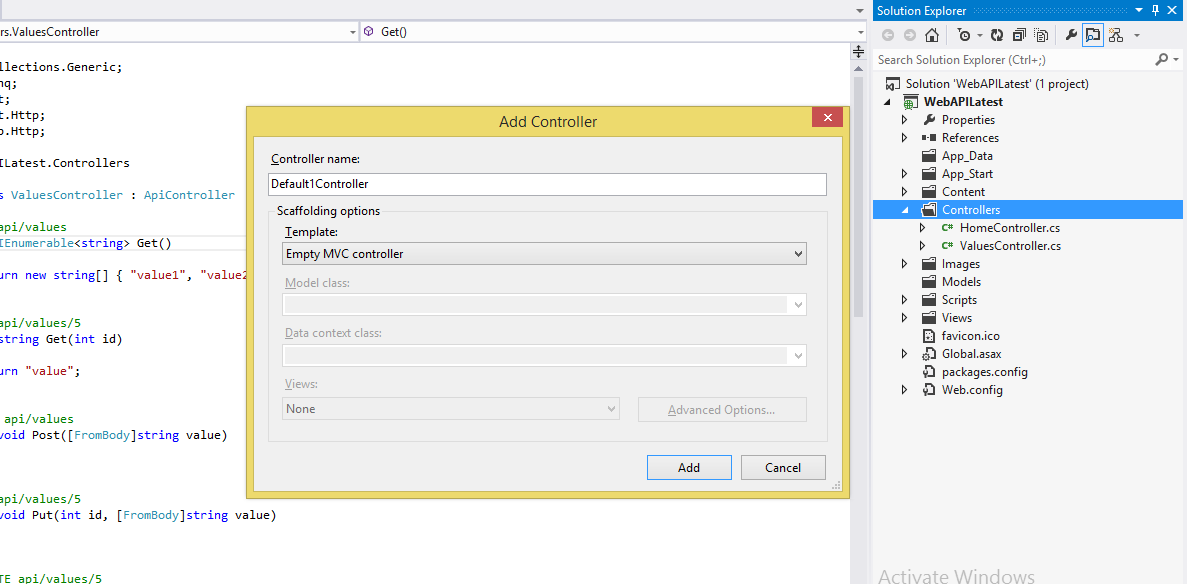


**POSTMAN** – similar to Fiddler debugging proxy tool, compose a request, execute with particular http methods (GET, POST, PUT, DELETE ..,).

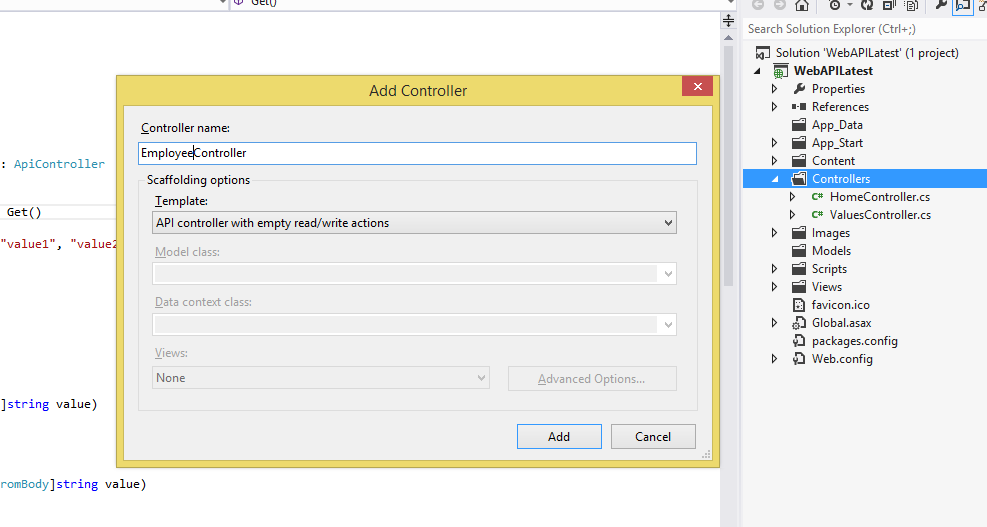
* Web API Controller:
* Like in MVC, Controllers are the handlers which handle the incoming http requests and output the response.
* Controllers are classes can be created under folder named “**Controllers”** or any other **“folder”,** or in the root folder of the project.
* It must be derived from “System.Web.Http.**ApiController”.**

Right Click on the Controllers folder 🡪 Add 🡪 Controller 🡪 Give appropriate name 🡪 Select template 🡪 Empty API controller /API controller with empty read write actions (below screenshots).

**#1**



#2



|  |
| --- |
| public class EmployeeController : ApiController |
| { |
| // GET api/employee |
| [HttpGet] |
| public IEnumerable<string> Emp() |
| { |
| return new string[] { "value1", "value2" }; |
| } |
|  |
| // GET api/employee/5 |
| public string Get(int id) |
| { |
| return "value"; |
| } |
|  |
| // POST api/employee |
| public void Post([FromUri]string value) |
| { |
|  |
| } |
|  |
| // PUT api/employee/5 |
| public void Put(int id, [FromBody]string value) |
| { |
| } |
|  |
| // DELETE api/employee/5 |
| public void Delete(int id) |
| { |
| } |
|  |
| public IEnumerable<string> PostEmp() |
| { |
| return new string[] { "Value3", "Value4" }; |
| } |
| } |

* It works based on the incoming URI and the type of HTTP request (GET/POST/PUT/PATCH/DELETE).
* Based on which the controller redirects to appropriate action method.

**GET – retrieves the data, POST – inserts a new record, PUT – updates the existing record,**

**PATCH – updates record partially, DELETE – deletes a record.**

For example, in the above **e.g.;**

* **Post** request type without any parameter redirects to **PostEmp ()** method, same with “**value**” parameter, redirects to “void Post([FromUri]string value)”.

**Action Method Naming Convention:**

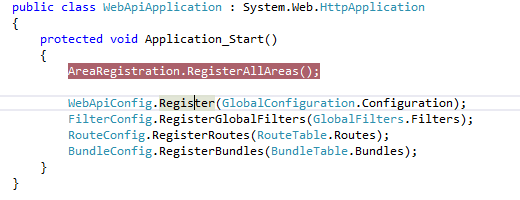
* Action method name can be HTTP method’s name(GET/POST/PUT etc.,) or can be given different name but with the HTTP verb attribute([HTTPGET/HTTPPOST) applied to it or any name suffixed with HTTP verb (GetEmployee).

**Difference between Web API and MVC controller**

| Web API Controller | MVC Controller |
| --- | --- |
| Derives from System.Web.Http.ApiController class | Derives from System.Web.Mvc.Controller class. |
| Method name must start with Http verbs otherwise apply http verbs attribute. | Must apply appropriate Http verbs attribute. |
| Specialized in returning data. | Specialized in rendering view. |
| Return data automatically formatted based on Accept-Type header attribute. Default to json or xml. | Returns ActionResult or any derived type. |
| Requires .NET 4.0 or above | Requires .NET 3.5 or above |

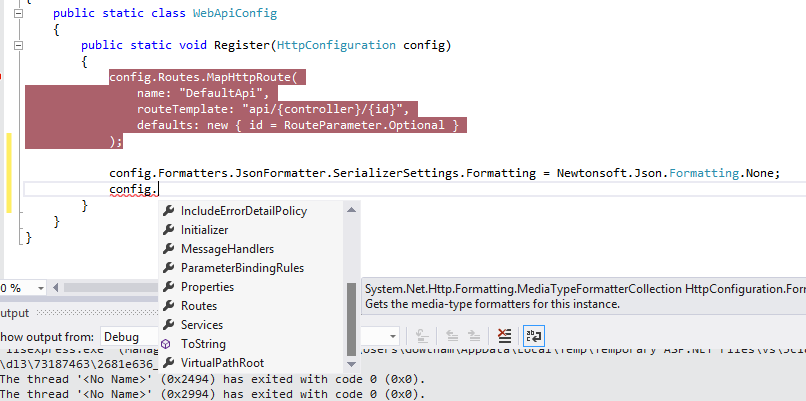
* Configure Web API:
* Likewise in MVC, it supports code based configuration. It can be configured in “Register**”** method of **“WebAPIconfig.cs”** file.
* Register method is a static one and it will be called under Application\_Start from **Global.asax** file with the parameter GlobalConfiguration.

**Global.asax**



* In the Register method, it is received as **HttpConfiguration** object and which helps us to customize API’s properties/behaviors to our needs like Routing, Formatting, Dependency injection resolver etc.., See below are the properties that can be overridden or customized to our application needs.

**WebApiConfig.cs**

****

* Web API Routing:

Routing in Web API works in the similar way as in ASP.NET MVC. It handles the incoming http request and routes to appropriate controller/action method.

Two types of Routing;

1. Conventional based Routing
2. Attribute based Routing (WebAPI 2.0 i.e., MVC 5.0)

**Conventional Based Routing:**

* It works based on the routeTemplate defined in the Register method of **Web API config**, by using HttpConfiguration object. “**Config.Routes**” is a route table, there must be at least one route table to be present for Conventional based routing to work.

|  |
| --- |
| //Manually Creating routes -- internal function of MapHttpRoute method |
| System.Web.Http.Routing.IHttpRoute employeeroute = config.Routes.CreateRoute( |
| routeTemplate: "api/employeedetails/{id}", |
| defaults: new { controller = "Employee", id = RouteParameter.Optional }, |
| constraints:null |
| ); |
|  |
| config.Routes.Add("EmployeeRoute", employeeroute); |
|  |
| config.Routes.MapHttpRoute( |
| name: "EmployeeRoute", |
| routeTemplate: "api/employeedetails/{id}", |
| defaults: new { controller = "Employee", id = RouteParameter.Optional } |
| ); |
| config.Routes.MapHttpRoute( |
| name: "DefaultApi", |
| routeTemplate: "api/{controller}/{id}", |
| defaults: new { id = RouteParameter.Optional } |
| ); |

|  |
| --- |
|  |

**Config.Routes.MapHttpRoute** is an extension method which helps to create and add Route to the Route collection.

Basically, based on the incoming URI and the http method (GET/POST/PUT/DELETE/PATCH), the controller/action methods gets called and executed.

Above table has multiple routes configured in which default route to be given at the last, so that if no route is matched with incoming request, the default one gets executed. If you see in the above e.g.,

|  |
| --- |
| routeTemplate: "api/employeedetails/{id}", |
| defaults: new { controller = "Employee", id = RouteParameter.Optional } |

Whenever a request with Uri “**http://localhost:65154/api/employeedetails**” comes, employee controller will be called. By this way our name will not be exposed.

**Attribute Based Routing:** (Supported only in WebAPI 2.0 i.e., MVC 5.0)

As the name suggests, here the incoming URI will be routed to controller/action method based on the attribute applied on the action/controller level.

* Attribute used is; **[Route (“”)]**

Screenshot below;

|  |
| --- |
|  |

* Here, if you see **Route** attribute is applied on the action method level, so the **Get** method of the **Values** controller must be called like **“**[**http://localhost:51820/api/value/get**](http://localhost:51820/api/value/get)**”.**
* But, as a **prerequisite** -Attribute based Route to work; we need to configure it in the Register method of **WebAPIconfig.cs** file. See below;
* **Config.MapHttpAttributeRoutes ();**

|  |
| --- |
|  |

**Findings*****(while practicing)***

*Whenever we use multiple routes, always place the specific/unique routes at the top and the generic i.e., default routes at the bottom then only things are working as expected.*

*See below screenshot;*

|  |
| --- |
|  |

*In the above eg.* ***DefaultApi*** *and* ***WebAPIRoute2*** *are specific routes, but* ***WebAPIRoute2*** *is placed below the generic route* ***WebAPIRoute1*** *(in fact default one, just renamed). Here, the URI with* ***“api/apivalues”***[*http://localhost:51820/api/apivalues*](http://localhost:51820/api/apivalues) *does work as expected but the URI* [*http://localhost:51820/api/test*](http://localhost:51820/api/test) *does not work as expected, shows error;*

|  |
| --- |
| *<Error>* |
| *<Message>* |
| *No HTTP resource was found that matches the request URI 'http://localhost:51820/api/test'.* |
| *</Message>* |
| *<MessageDetail>* |
| *No type was found that matches the controller named 'test'.* |
| *</MessageDetail>* |
| *</Error>* |

*To make* ***WebAPIRoute2*** *to work,**we have to place it above the WebAPIRoute1.*

|  |
| --- |
|  |

*In the above, I’ve also added* ***action*** *attribute in defaults to “****pTestAPI”,*** *so whenever we call* [*http://localhost:51820/api/test*](http://localhost:51820/api/test) *with* ***HTTPGET*** *routes to Values controller and pTestAPI method will be called.*

* Parameters Binding:

Previous, we have seen how the incoming request URI gets routed to the appropriate controller/action methods. Here, we see how the parameters of action methods get bounded with the incoming http URI.

* The parameters of an action method can be either **primitive** or **complex** types.

**Primitive types:**

* *int, bool, string, DateTime, decimal, GUID or any other type which can be converted from string format.*

**Complex types:**

* *User defined classes.*

By default, Web API will try to extract the values for primitive type parameters from query string of incoming http request and for Complex types takes from request body. Below follows examples;

* The below method will try to assign the parameter values from the incoming URI query string. So if the incoming request is **http://localhost:51820/api/values?id=1200&s=abc** then **“id”** value will be **1200** and **“s”** value will be **“abc”.**

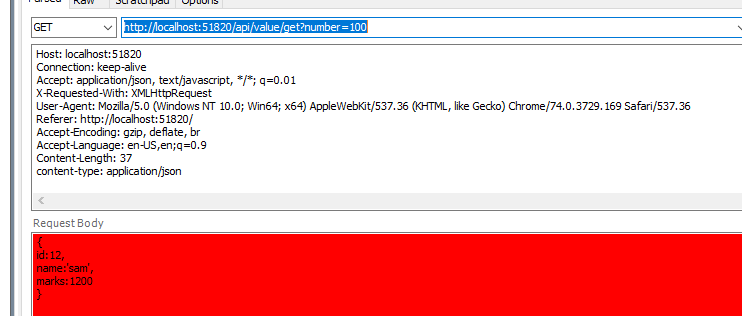
|  |
| --- |
|  |

* The below method has both primitive and complex type parameters, so “**number”** will be extracted from **query string** and “**student” object** will be extracted from **request body**.

|  |
| --- |
|  |

* Tested in fiddler with the below URL, http method **GET**, **student** object values provided in **Request Body**. See the red color because by default GET method takes values from query string, but here we pass the **complex type parameter**, so its values has to be given in request body.

That’s why there is red alert in the text area, **but works with no fault**.

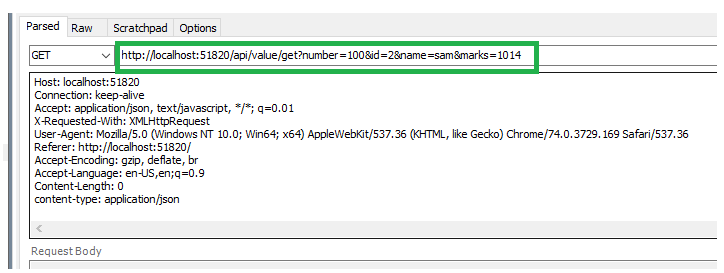


* But for any reason, if we want to change its behavior i.e., reading the primitive type from request body or reading the complex type from query string, we have to use **[FromUri]** and **[FromBody]** attributes.

In below example, Student object will be assigned from query string.

|  |
| --- |
|  |

Here you see the student object properties are assigned from query string.



|  |
| --- |
| ***Note:*** |
| ***FromBody*** *attribute can be applied on only one primitive parameter of an action method. It cannot be applied on multiple primitive parameters of the same action method.* |

Courtesy: <https://www.tutorialsteacher.com/webapi/parameter-binding-in-web-api>

|  |
| --- |
|  |

* Action Return type:

In earlier section, we’ve seen how the parameters bindings of the action method work in Web API. Here, we’ll see what the return types available for an action method are;

Return types;

1. Void
2. Primitive or Complex type
3. HttpResponseMessage
4. IHttpActionResult

**VOID:**

It is not necessary that always action has to return something. For example, consider the below action method returns nothing but just deletes a record from List/DB.

**Calling the Delete method**

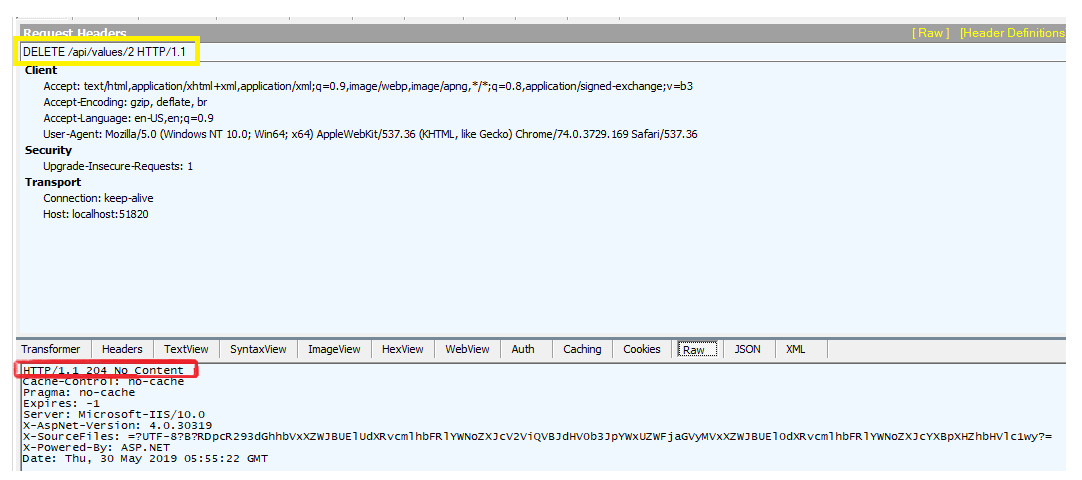
|  |
| --- |
|  |

**Method that is called returns nothing but just removes a record from Student List**

|  |
| --- |
|  |

**See the response output from fiddler’s RAW view;**

* You can see the Http Delete is called with URL (api/values/2) 🡪 2 is the id passed.
* Its response header shows Status Code – 204 No Content which is actually ­­misleading information.

****

**Primitive Type or Complex Type:**

The Return type can also be primary (int, string, decimal, double etc.) or complex types (List or user defined classes).

**Calling the PUT/update method:**

You can see below method returns string (primary type);

|  |
| --- |
|  |

**Below Called Update Student method:**

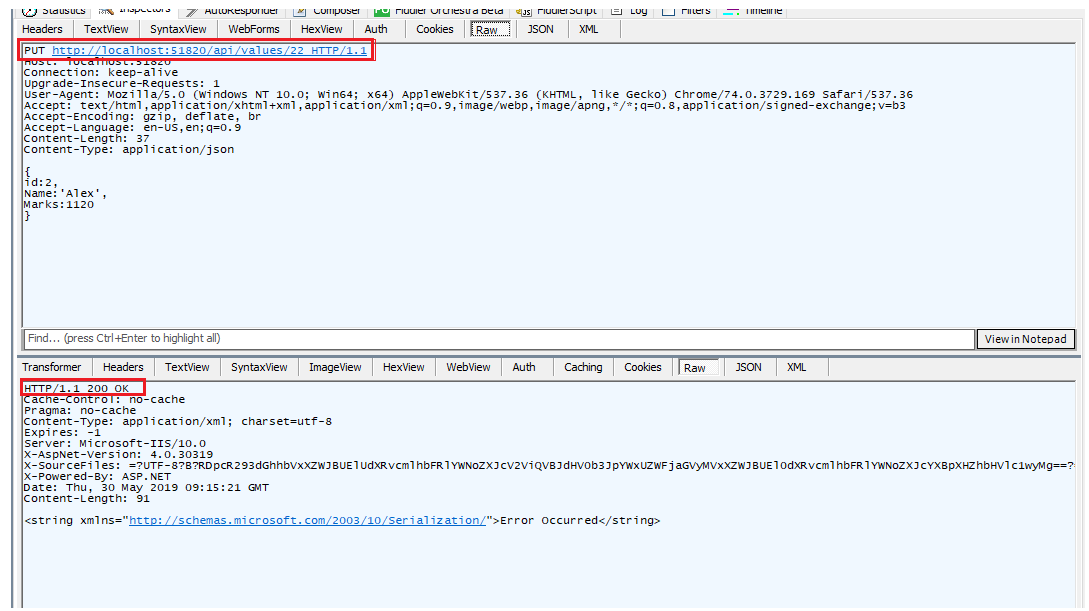
|  |
| --- |
|  |

**See the response output from fiddler’s RAW view;**

|  |
| --- |
|  |

**See the status code it returns 200 OK.**

* In case see for the below scenario, the Id parameter passed is “**22**” which is not there in the list to update, but as per the code it returns a string value as “**Error Occurred**” and the status code says **200 OK** as the action successfully executed, which is also misleading information.



**Complex Type:**

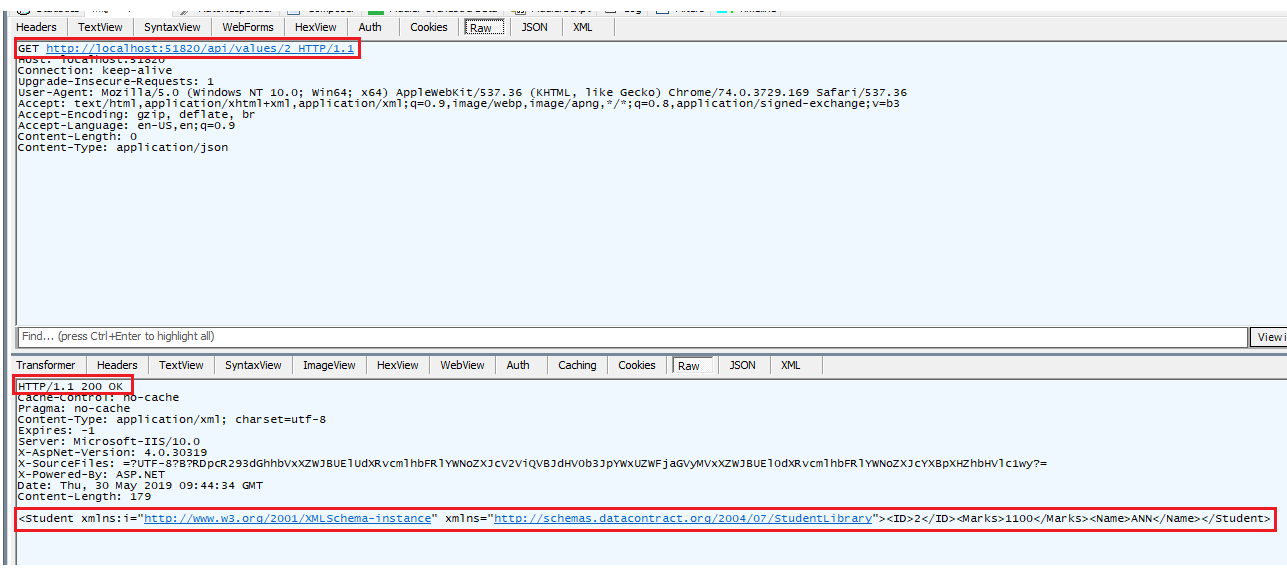
|  |
| --- |
|  |

**See the response output from fiddler’s RAW view;**

* Here, I tried passing the **id = 22** as parameter value, but there is no value as such in list, so it returns **nil value** in response (below marked), but since the action executed successfully, status code says **200 OK**.
* Here, what we infer the **response code is out of our control** as it gives misleading information.

|  |
| --- |
|  |

**For id=2 🡪 we get the proper result with proper status code 200 OK.**



**HttpResponseMessage:**

* The Web API controller always sends the Http Response Message as we’ve seen in the previous examples (returning the **status code**).
* So, we can also return **HttpResponseMessage** directly from the action method. By this way, we can configure the response to be returned based on our requirement.

See below example;

|  |
| --- |
|  |

It clearly enables us to return appropriate status code, if the student id is present, we get status code **200 OK**, and else we get **Not Found 404** as response.

**See the response output from fiddler’s RAW view;**

Id=22 🡪 which is not available in the list hence **404 Not Found** as we’ve configured it in code.

|  |
| --- |
|  |

Below For id = 2🡪 It returns with appropriate status code 200 Ok with the data highlighted as well since it has been configured in the code.

|  |
| --- |
|  |

**IHttpActionResult:**

* The **IHttpActionResult** is introduced in Web API 2.0. An action method can return an implementation of **IHttpActionResult** class which is more or less similar to **ActionResult** class in MVC.
* We can also create our own class implementing **IHttpActionResult** and return it as well or use various methods of **ApiController** class object that implement the **IHttpActionResult** class.

|  |
| --- |
|  |

Few methods available in the above **ApiController** class implementing **IHttpController** class. In fact

|  |
| --- |
|  |

**OkNegotiatedContentResult** class is an implementation class of **IHttpActionResult** class.

|  |
| --- |
|  |

See the below method returns IHttpActionResult class which returns the response as **“Ok” and “NotFound”** which have implemented the **IHttpActionResult** class.

|  |
| --- |
|  |

|  |
| --- |
|  |

**See the response output from fiddler’s RAW view;**

Id = 23 which is not available in the List, hence it returns **NotFound ()** as response.

|  |
| --- |
|  |

If Id = 2 is passed, the **“Ok ()”** method as response. See below snapshot.

|  |
| --- |
|  |

Below given are the methods of **ApiController** class that returns an object of a class that implements **IHttpActionResult** Interface.

|  |
| --- |
|  |

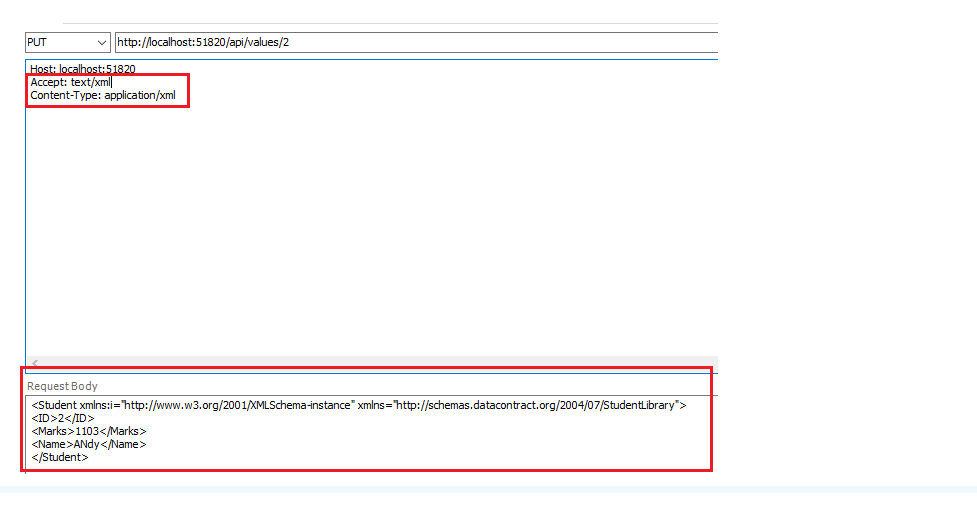
* Web API Request/Response Data Formats:

In earlier section, we’ve seen what all the return types of the action method are. Here, we’ll learn how the Web API handles the different formats/types of request/response data. (Xml, Json).

In HTTP request, the Media Type (aka MIME) is specified in the request header using **Accept** and **Content-Type** attributes.

* **Accept** header attribute specifies the format of the response data which the client expects.
* **Content-type** header attribute specifies the format of the data in the request body that is sent to the receiver to parse it accordingly to process and send the response to the client.

**Request Header**

****

* Media Type Formatters:

In previous section, we’ve seen we can configure different formats of data type can be passed to server (WebAPI) or can expect different response data from WebAPI using Accept/Content-Type headers in HTTP request.

Here, we’ll learn how the WebAPI does processes different formats of request data or how does it change the response data based on the Accept header value.

The Answer is; by using **Media Type Formatters**.

Media type formatters are the classes responsible for serializing request/response data so that WebAPI can understand the request data and send the response data in format in which the client expects.

Below are the built in Media type formatters;

| Media Type Formatter Class | MIME Type | Description |
| --- | --- | --- |
| JsonMediaTypeFormatter | application/json, text/json | Handles JSON format |
| XmlMediaTypeFormatter | application/xml, text/json | Handles XML format |
| FormUrlEncodedMediaTypeFormatter | application/x-www-form-urlencoded | Handles HTML form URL-encoded data |
| JQueryMvcFormUrlEncodedFormatter | application/x-www-form-urlencoded | Handles model-bound HTML form URL-encoded data |

Basically, when you request API resourcefromFiddler, the default format of response would be in JSON.

But when you fire the URL directly from browser, the response would be in XML format, that is because, if you inspect the request header, its Accept would be like;

**Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8**

But if any of the formatters you need to remove, for example you need always JSON response from API, you can also remove XML formatter from config (HttpConfiguration object).

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

Here, though we have accept header in **application/xml**, we will get only the JSON format output.

But, if we want both and in browser we need json format data, you need to include below code in Register method of WebAPI config.cs;

**System.Net.Http.Formatting.JsonMediaTypeFormatter jsonMediaTypeFormatter = config.Formatters.JsonFormatter;**

**jsonMediaTypeFormatter.SupportedMediaTypes.Add(new System.Net.Http.Headers.MediaTypeHeaderValue("text/html"));**

But even in the above scenario, the content-type of response header will still be as “text\html”. That is misleading in fact.

To change that we need to add a class and set the default content headers for “text/html”. And we need to register that custom formatter class in register method.

|  |
| --- |
| public class CustomJsonFormatter : System.Net.Http.Formatting.JsonMediaTypeFormatter |
| { |
| public CustomJsonFormatter() |
| { |
| this.SupportedMediaTypes.Add(new System.Net.Http.Headers.MediaTypeHeaderValue("text/html")); |
| this.SerializerSettings.Formatting = Newtonsoft.Json.Formatting.Indented;-- **indenting data (like Ctrl + K + D we do in code)** |
|  |
| } |
| /\* Type override then a space, shows the list of methods can be overridden, system generates the code for you\*/ |
| public override void SetDefaultContentHeaders(Type type, System.Net.Http.Headers.HttpContentHeaders headers, System.Net.Http.Headers.MediaTypeHeaderValue mediaType) |
| { |
| base.SetDefaultContentHeaders(type, headers, mediaType); |
| headers.ContentType = new System.Net.Http.Headers.MediaTypeHeaderValue("application/json"); |
|  |
| } |
| } |

**After indented:**

|  |
| --- |
|  |

Cross Origin Resource Sharing: (CORS)

As a security feature, browsers don’t allow cross origin sharing of resources like; “<http://localhost:34725/Employees.html>” is trying to access API of “<http://localhost:51820/api/employee>” or “<https://localhost:51820/api/employee>”.

***CORS can be enabled by below steps;***

* To install Nuget Package “**Microsoft ASP.NET Web API 2.2 Cross-Origin Support**” this contains the components to enable CORS in ASP.NET Web API.
* Enable CORS globally across all API controllers 🡪 Register it in **WebApiConfig.cs** Register method like below;



* Here, **EnableCorsAttribute** class is present in the **“System.Web.Http.Cors”** namespace. Its constructor has the three parameters (***string origins, string headers, string methods***) for including ***origins*** such as <http://localhost:34725> from which these API resources can be accessed. There can be comma separated values of origins as well, so only from those origins these API resources can be accessed, any other origin won’t be able to.

If any origin can access then use “**\***”.

***Headers*** parameter needs to explore more (not clear as of now)

***Methods*** parameter includes methods (such as GET, POST, PUT etc.,) can be accessible by given origin.

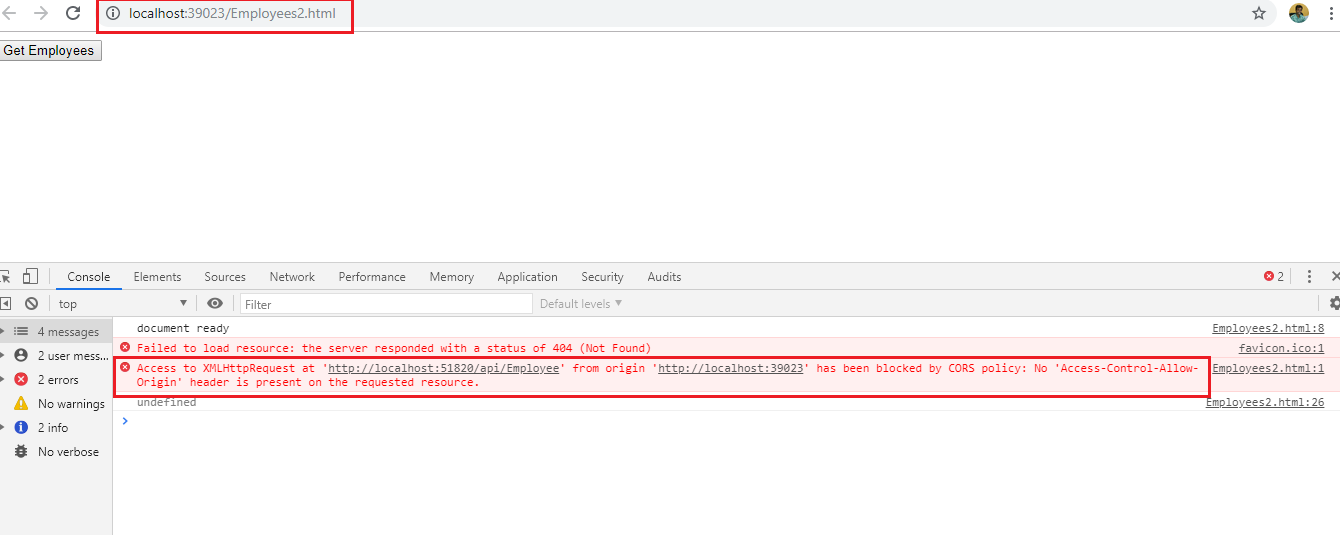
It can also be enabled at Controller/Action level by below;

So that CORS applies only to the Controller/Action method on which it is applied.

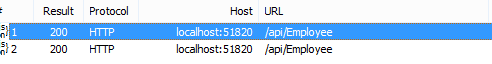
|  |
| --- |
|  |

Any other resource you try to access will show the below error in browser console;

Here look at the origin trying to access API, “<http://localhost:39023/>” and the above screenshot of origin allowed for CORS (“http://localhost:34725”) so we get the below “**Access-Control-Allow-Origin**” header error.



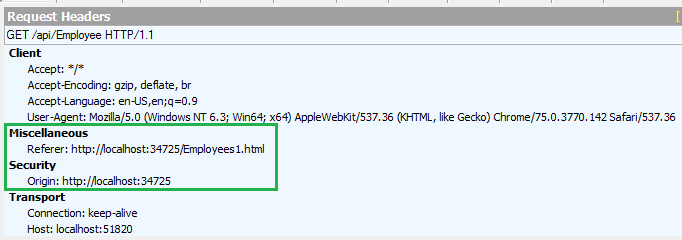
We can find out below what happens in fiddler when this is requested is given below;



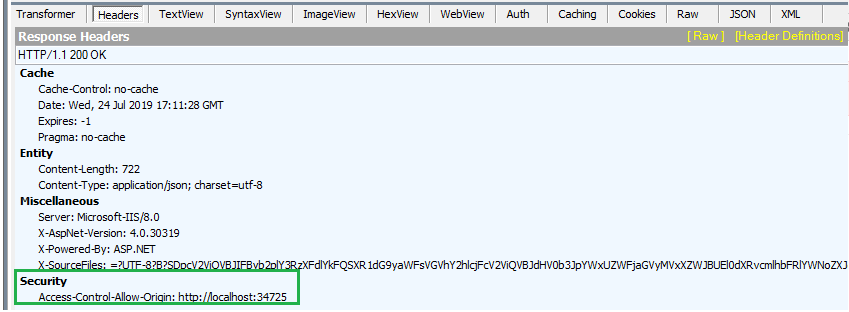
There are two requests one is from CORS enabled origin (<http://localhost:34725/Employees1.html>) and the other is from a different origin (<http://localhost:39023/Employees2.html>). See the Request/Response headers of each below screen shots;

**http://localhost:34725/Employees1.html:**

See below Security origin header shows the origin of the request.

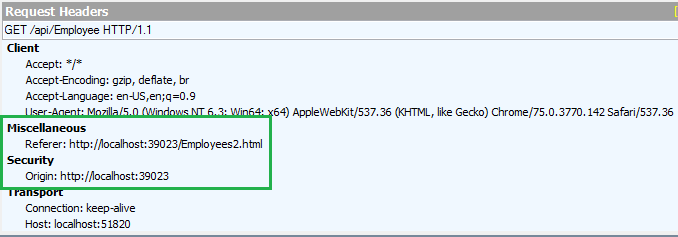
****

Below given the Response header which has a Security Access-Control-Allow-Origin header says the requested origin is allowed to access API. So this will be allowed by browser for displaying.

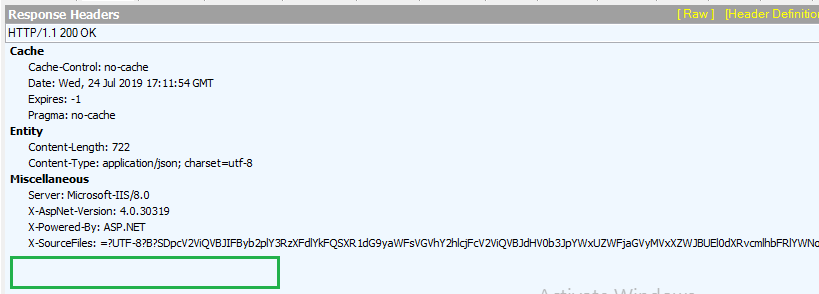


**http://localhost:39023/Employees2.html:**

See below Security origin header shows the origin of the request.

****

Below given the Response header which doesn’t have a Security Access-Control-Allow-Origin header. So this will be denied by browser for displaying.

****

* Filters: