Asp.net Web API

Christen Zarif

OutLine

- Web API
- Content- Negotiation
- Media Type Formatter
- Post – put – delete “HttptResponseMEssage”
- Method Name :[HttpGet]- [HttpPost]
- FromBody – FromURI “Default Parameter Binding”
- Attribute Routing

outline

- Representational state transfer Architecture
- Web Services
- Rest Web Services
- API
- Web API
- ASP.Net Web API

Web Service

- A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data

What is a RESTful?

- **REST**, or **RE**presentational **S**tate **T**ransfer, is an architectural style for providing standards between **computer systems** on the **web**, making it easier for systems to communicate with each other.
- The REST is actually an architectural pattern that is basically used for creating Web API's which uses HTTP as the communication method.
- This REST architectural pattern specifies a set of constraints and those constraints a system should follow to be considered as a Restful Service. The following are the REST constraints
- For More

RESTful Constraints?

- **Client-Server Constraint**
  - In the REST architectural style, the implementation of the **client** and the implementation of the **server** can be **done independently** without each knowing about the other.
  - This means that the **code** on the client side can be **changed** at any time **without affecting** the operation of the server, and the code on the server side can be changed without affecting the operation of the client.
  - As long as each side knows what **format of messages** to send to the other, they can be kept modular and separate.

RESTful Constraints?(cont.)

- **Client-Server 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.

RESTful Constraints?(cont.)

- **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 only **contain** the **necessary information** for the server to process that request.
  - This ensures that each **request** can be **treated independently** by the server.

RESTful Constraints?(cont.)

- **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.

RESTful Constraints?(cont.)

- **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 <u>what a resource</u> is and the <u>HTTP verbs</u> - GET, PUT, POST & DELETE.
    - 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.

RESTful Constraints?(cont.)

- **Uniform Interface**
-

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

RESTful Web Services

- Web services based on REST Architecture are known as RESTful Web Services.
- These web services use HTTP methods to implement the concept of REST architecture.
- A **RESTful web service** usually defines a URI (Uniform Resource Identifier), which is a service that provides resource representation such as JSON and a set of HTTP Methods

Http Methods

- **GET** – Provides a read only access to a resource.
- **PUT** – Used to update an existing resource or create a new resource.
- **DELETE** – Used to remove a resource.
- **POST** –Used to create a new resource.

```
app.get('/home', function (req, res) {
    res.sendFile( __dirname + "/" + "Express-PostMethod.html" );
})

app.post('/process_post', function (req, res) {

    res.end("<b>Welcome</b>: " + req.body.first_name + " " + req.body.last_name);
})
```

Http –Terms and Concepts

- **Request Verbs :**
  - (GET, POST, PUT & DELETE)
  - These verbs describe what should be done with the Recourse
- **Request Header :**
  - Contains additional information about the request ,
    Example : what type of response is required (Content-Type,Accept)
- **Request Body** :
  - Contains the data to send to the server
- **Response Body :**
  - Contains the data send as response from the server
- **Response Status Codes :**
  - Provide the client ,the status of the request

Http Requests and Responses

- Http Request

Http verb

URL

Http version

```
POST /api/v1/person HTTP/1.1
```

Http Header

```
host: example.com
Content-Type: x-www-form-urlencoded
```

Http Body

```
name=joe&surname=blogg
```

Http Requests and Responses (Con.)

- Http Response

Status Code

Description

```
201 Created
```

```
Content-Type: text/plain
Location: /api/v1/person/123
```

Http Header

```
Person was successfully created
```

Http Body

HTTP Header Fields

- HTTP Header Fields define parameters for the HTTP operation
- Around that line are Content-Type, Content-Length, and a few others. Each of them is a **so-called HTTP Header Field**.
- These fields exist in both requests and responses. Most of them differ for requests and responses, but some , such as content-type, exist on both sides of the HTTP protocol.

Use HTTP Response Codes to Indicate Status

- **200 OK :** General success status code. This is the most common code. Used to indicate success.
- **201 CREATED :** Successful creation occurred (via **either POST or PUT**). Set the Location header to contain a link to the newly-created resource (on POST). Response body content may or may not be present.
- **204 NO CONTENT :** Indicates success but nothing is in the response body, often used **for DELETE and PUT** operations.
- **400 BAD REQUEST :** General error for when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples.
- **401 UNAUTHORIZED :** Error code response for missing or invalid authentication token.
- **403 FORBIDDEN :** Error code for when the user is not authorized to perform the operation or the resource is unavailable for some reason (e.g. time constraints, etc.).
- **404 NOT FOUND :** Used when the requested resource is not found, whether it doesn't exist or if there was a 401 or 403 that, for security reasons, the service wants to mask.
- **405 METHOD NOT ALLOWED :** Used to indicate that the requested URL exists, but the requested HTTP method is not applicable. For example, POST /users/12345 where the API doesn't support creation of resources this way (with a provided ID). **The Allow HTTP header must be set when returning a 405 to indicate the HTTP methods that are supported**. In the previous case, the header would look like "Allow: GET, PUT, DELETE"
- **409 CONFLICT :** Whenever a resource conflict would be caused by fulfilling the request. Duplicate entries, such as trying to create two customers with the same information, and deleting root objects when cascade-delete is not supported are a couple of examples.
- **500 INTERNAL SERVER ERROR :** Never return this intentionally. The general catch-all error when the server-side throws an exception. Use this only for errors that the consumer cannot address from their end.

Internet Media Types

- "text/html" is an identifier for a file format on the Internet

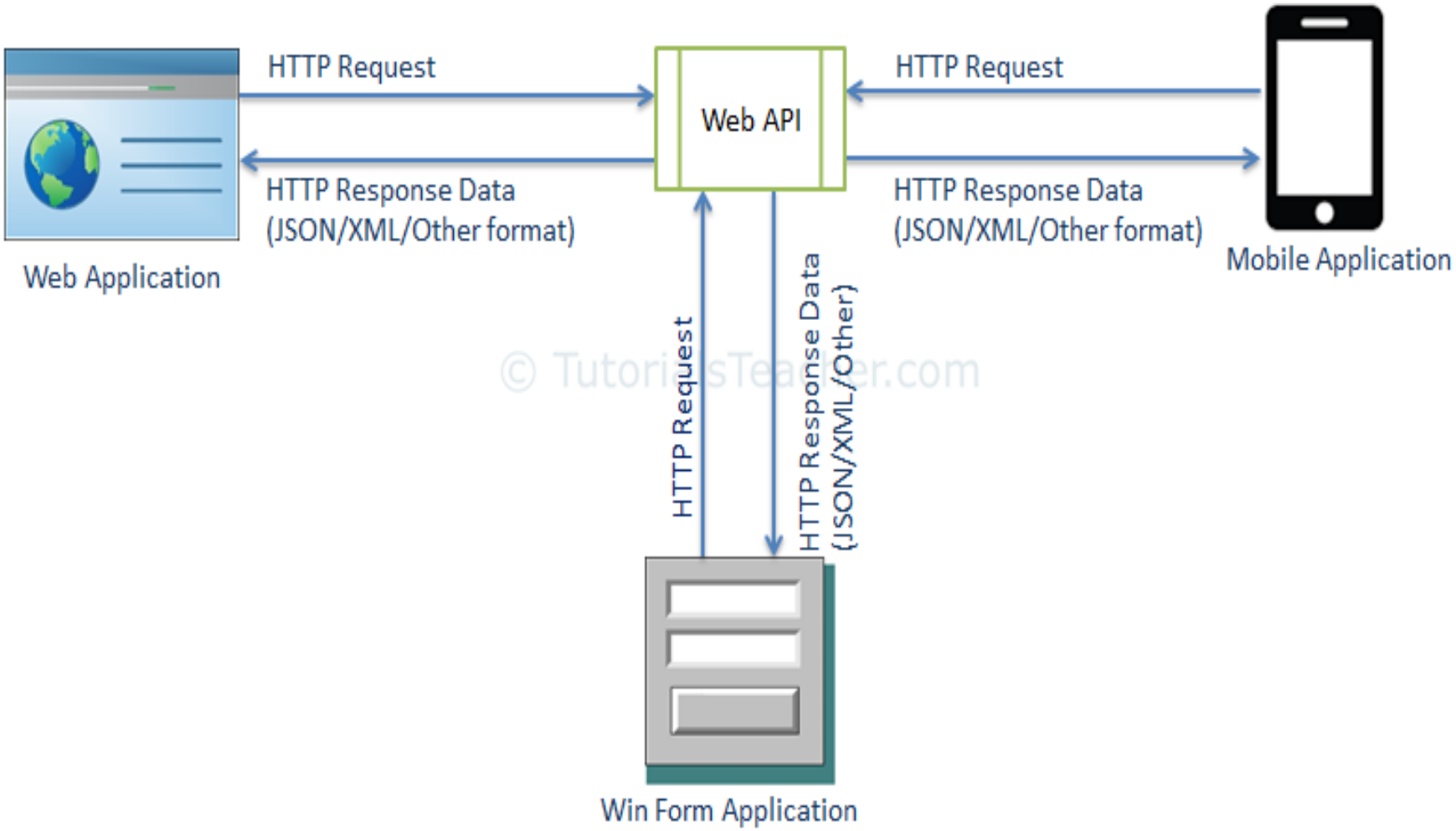
- text/html media type definition, it consists of (at least) two parts, where the **first is a typ**e and the **second is a subtype**. Either can be followed by one or more parameters.

```
content-type: text/html
```

ASP.NET Web API

- The ASP.NET Web API is an extensible framework for building **HTTP based services** that can be accessed in **different applications on different platforms** such as web, windows, mobile etc.
- The most common use for building RESTful services
- It works more or less the same way as ASP.NET MVC web application **except** that it sends data as a response instead of html view.
- It is like a webservice or WCF service but the **exception** is that it only supports HTTP protocol.

Web API



Mapping HTTP concepts in HTTP to ASP.NET Web API elements

Verb -> Action selection

URL -> Routing, Controller selection, parameter binding

Content negotiation-> MediaTypeFormatter

Content type -> MediaTypeFormatter

Encoding -> MediaTypeFormatter

```
POST http://localhost/PingYourPackage.API.Client.Web/Home/Create HTTP/1.1
Host: localhost
Connection: keep-alive
Content-Length: 360
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Origin: http://localhost
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/27.0.1453.116 Safari
Content-Type: application/x-www-form-urlencoded
Referer: http://localhost/PingYourPackage.API.Client.Web/Home/Create
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-GB,en;q=0.8,en-US;q=0.6
Cookie: glimpseState=On; glimpseOptions=%7B%22open%22%3Atrue%2C%22height%22%3A300%2C%22activeTab%22%3A%22Trace%22
__RequestVerificationToken=MuRJxi8h0Hw2Lg7tboaqTmyNcH3nJoDq62XVjQREgyvLK4FNGjOI4VEqh7N_0CVo5exKVVBYXnPevvf7vp_El8
```

Other including authentication, caching, etc -> custom delegating handlers, filters

Payload -> Action invocation, MediaTypeFormatter, parameter binding, model binding

ASP.NET Web API Characteristics

- is an ideal platform for building **RESTful services**.
- Built on **top of ASP.NET** and supports ASP.NET request/response pipeline
- Maps **HTTP verbs** to **method names**.
- Supports different formats of response data. Built-in support **for JSON, XML, BSON** format.
- Can be hosted in IIS, **Self-hosted** or other web server that supports .NET 4.0+.
- ASP.NET Web API framework includes new **HttpClient** to communicate with Web API server. HttpClient can be used in ASP.MVC server side, Windows Form application, Console application or other apps.

ASP.NET Web API Versions:

Web API Version	Supported .NET Framework	Coincides with	Supported in
Web API 1.0	.NET Framework 4.0	ASP.NET MVC 4	VS 2010
Web API 2 - Current	.NET Framework 4.5	ASP.NET MVC 5	VS 2012, 2013

Create Web API project:

- You can create a Web API project in two ways:
  - Web API with MVC Project
  - Stand-alone Web API Project

Stand-alone Web API Project

Create a new ASP.NET Web Application

Empty

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

Web Forms

A project template for creating ASP.NET Web Forms applications. ASP.NET Web Forms lets you build dynamic websites using a familiar drag-and-drop, event-driven model. A design surface and hundreds of controls and components let you rapidly build sophisticated, powerful UI-driven sites with data access.

MVC

A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Web API

A project template for creating RESTful HTTP services that can reach a broad range of clients including browsers and mobile devices.

Single Page Application

A project template for creating rich client side JavaScript driven HTML5 applications using ASP.NET Web API. Single Page Applications provide a rich user experience which includes client-side interactions using HTML5, CSS3, and JavaScript.

Authentication

No Authentication

Change

Add folders & core references

- [ ] Web Forms
- [ ] MVC
- [x] Web API

Advanced

- [ ] Configure for HTTPS
- [ ] Docker support

(Requires Docker Desktop)

- [ ] Also create a project for unit tests

WebApplication1.Tests

Back | Create

Stand-alone Web API Project (Con.)

- Open Manage NuGet Packages popup.
- Select **Microsoft ASP.NET Web API2.2** package and click on Install

Web API Configuration

- Web API is configured only using code based configuration using **GlobalConfiguration** class.
- The **Configure()** method requires a callback method where you have configured your Web API.

Global.asax

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        GlobalConfiguration.Configure(HelloWebAPIConfig.Register);
    }
}
```

Web API Configuration (Con.)

- We need to configure our Web API routes when application starts.
- So call HelloWebAPIConfig.Register() method in the Application_Start event in the Global.asax as shown below.

```
public static class HelloWebAPIConfig
{
    public static void Register(HttpConfiguration config)
    {
        // Web API routes
        config.MapHttpAttributeRoutes();

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}"
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

Demo

- First Web API Action:

Example: Web API Controller

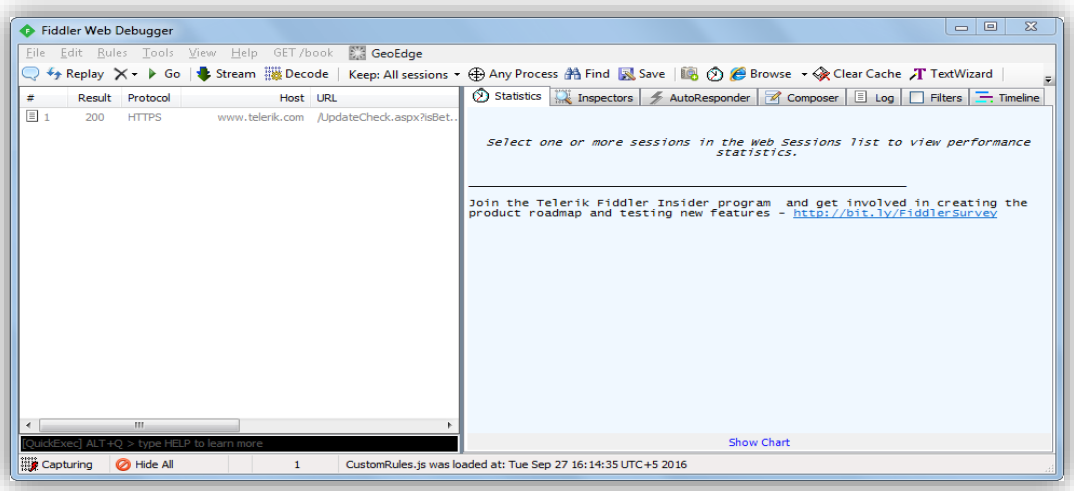
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace HelloWebAPI.Controller
{
    public class HelloController : ApiController
    {
        public string Get()
        {
            return "Hello World";
        }
    }
}
```

Test Web API

- To test Web API locally to check request & response during development.
- We can use the following third party tools for testing Web API.
  - Fiddler
  - Postman

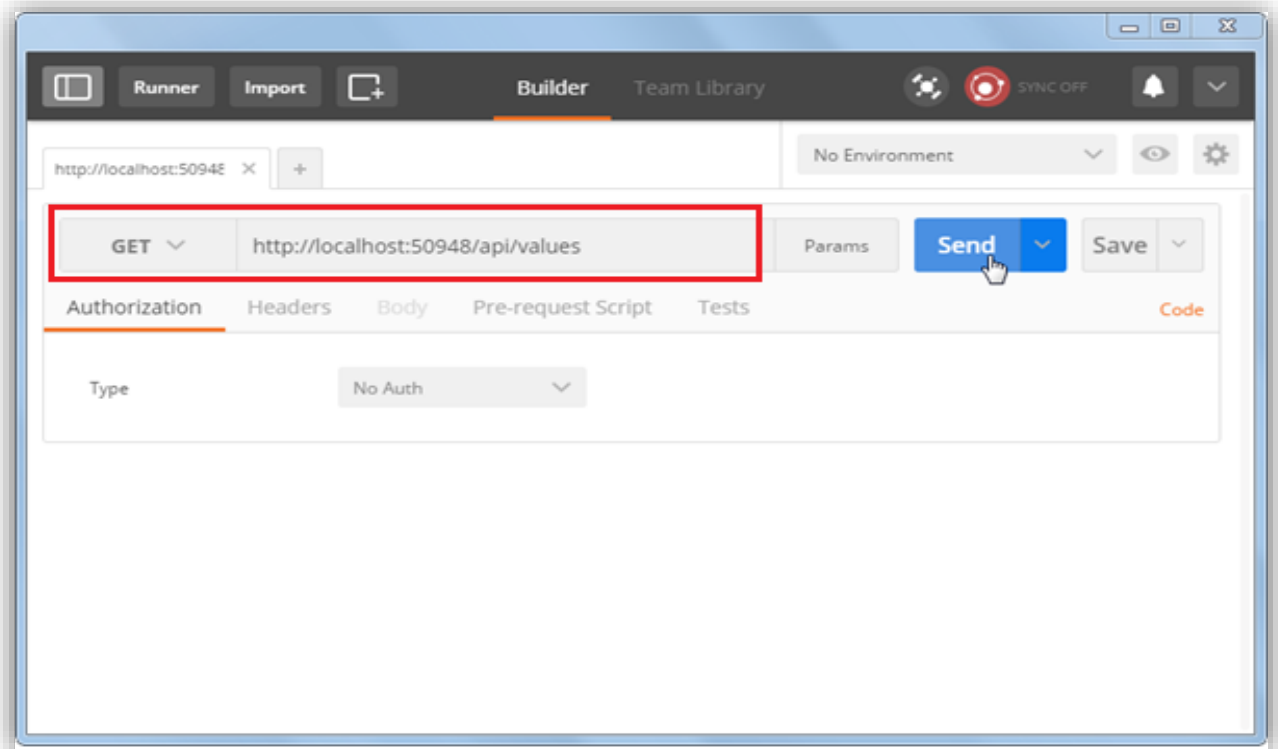
Fiddler

- Fiddler is a free debugging proxy for any browser.
- We can use it to compose and execute different HTTP requests to our Web API and check HTTP response.
- **Step 1:**
  - Download and install Fiddler from
    - https://www.telerik.com/download/fiddler
- **Step 2:**
  - Open fiddler



Postman

- Postman is a free API debugging tool. You can install it on your Chrome browser or Mac. Install it for Chrome from
- https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop



Web API Controller

- similar to ASP.NET MVC controller. It handles incoming HTTP requests and send response back to the caller.
- derived from **System.Web.Http.ApiController** class.
- All the public methods of the controller are called action methods.
- **Notes** : action methods names match with HTTP verbs like Get, Post, Put and Delete.

Web API Controller (Con.)

```
public class ValuesController : ApiController
{
    // GET api/values
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    // GET api/values/5
    public string Get(int id)
    {
        return "value";
    }

    // POST api/values
    public void Post([FromBody]string value)
    {
    }

    // PUT api/values/5
    public void Put(int id, [FromBody]string value)
    {
    }

    // DELETE api/values/5
    public void Delete(int id)
    {
    }
}
```

Web API controller Base class

Handles Http GET request

http://localhost:1234/api/values

Handles Http GET request with query string

http://localhost:1234/api/values?id=1

Handles Http POST request

http://localhost:1234/api/values

Handles Http Put request

http://localhost:1234/api/values?id=1

Handles Http DELETE request

http://localhost:1234/api/values?id=1

Web API Controller (Con.)

```
namespace MyWebAPI.Controllers
{
    public class ValuesController : ApiController
    {
        [HttpGet]
        public IEnumerable<string> Values()
        {
            return new string[] { "value1", "value2" };
        }

        [HttpGet]
        public string Value(int id)
        {
            return "value";
        }

        [HttpPost]
        public void SaveNewValue([FromBody]string value)
        {
        }

        [HttpPut]
        public void UpdateValue(int id, [FromBody]string value)
        {
        }

        [HttpDelete]
        public void RemoveValue(int id)
        {
        }
    }
}
```

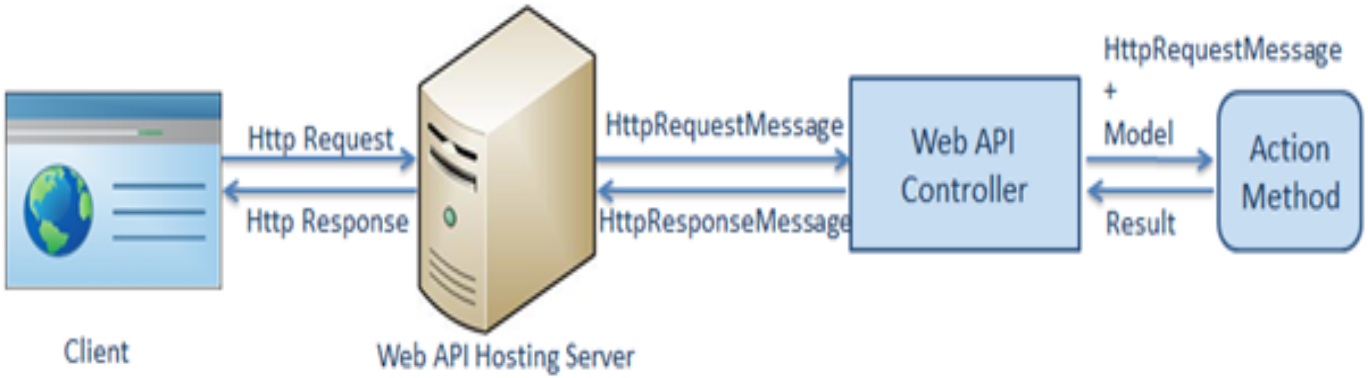
Methods that do not start with an HTTP verb then you can apply the appropriate **http verb attribute** on the method

Action Method Naming Conventions

HTTP Request Method	Possible Web API Action Method Name	Usage
GET	Get() | get() | GET() | GetAllStudent() *any name starting with Get *	Retrieves data.
POST	Post() | post() | POST() | PostNewStudent() *any name starting with Post*	Inserts new record.
PUT	Put() | put() | PUT() | PutStudent() *any name starting with Put*	Updates existing record.
PATCH	Patch() | patch() | PATCH() | PatchStudent() *any name starting with Patch*	Updates record partially.
DELETE	Delete() | delete() | DELETE() | DeleteStudent() *any name starting with Delete*	Deletes record.

Web API Controller(cont.)

- The following figure illustrates the overall request/response pipeline.



Web API Request Pipeline

MVC & Web API 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

Web API Routing

- Web API routing is similar to ASP.NET MVC Routing.
- It routes an incoming HTTP request to a particular action method on a Web API controller.
- Web API supports two types of routing:
  - Convention-based Routing
  - Attribute Routing

Web API Routing(cont.)

- **Convention-based Routing**
- In the convention-based routing, Web API uses **route templates** to determine which controller and action method to execute.
- At least one route template must be added into route table in order to handle various HTTP requests.
- When we created Web API project using WebAPI template in the Create Web API Project section, it also added WebApiConfig class in the App_Start folder with default route as shown below.

Web API Routing(cont.)

- **Convention-based Routing**

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // Enable attribute routing
        config.MapHttpAttributeRoutes();

        // Add default route using convention-based routing
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

Web API Routing(cont.)

- **Convention-based Routing (**Configure Multiple Routes)
-

```
        // school route
config.Routes.MapHttpRoute(
    name: "School",
    routeTemplate: "api/myschool/{id}",
    defaults: new { controller="school", id = RouteParameter.Optional }
    constraints: new { id ="/d+" }
);

        // default route
config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

Web API Routing(cont.)

- **Attribute Routing**
  - Attribute routing is supported in **Web API 2.**
  - As the name implies, attribute routing uses **[Route()]** attribute to define routes.
  - The Route attribute can be applied on any controller or action method.
  - In order to use attribute routing with Web API, it must be enabled in WebApiConfig by calling **config.MapHttpAttributeRoutes()** method.
  - Consider the following example of attribute routing.

Web API Routing(cont.)

- **Attribute Routing**
- Consider the following example
-

```
[HttpGet]
public string GetStudent(int id)
{
    return "student is " + id;
}
[HttpGet]
public string GetInstructor(int id)
{
    return "Instructor is "+id;
}
```

Web API Routing(cont.)

- **Attribute Routing**
- When you request the following url
- http://localhost:50630/api/home?id=1
- You will get the following error
-

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<Error>
  <Message>An error has occurred.</Message>
 ▼<ExceptionMessage>
   Multiple actions were found that match the request: GetStudent on type FIRSTAPI_PROJ.Controllers.HomeCont
  </ExceptionMessage>
  <ExceptionType>System.InvalidOperationException</ExceptionType>
 ▼<StackTrace>
   at System.Web.Http.Controllers.ApiControllerActionSelector.ActionSelectorCacheItem.SelectAction(HttpContro
   System.Web.Http.Controllers.ApiControllerActionSelector.SelectAction(HttpControllerContext controllerConte
   controllerContext, CancellationToken cancellationToken) at System.Web.Http.Dispatcher.HttpControllerDispat
  </StackTrace>
 </Error>
```

Web API Routing(cont.)

- **Attribute Routing**

We Can solve this by attribute route as the following

```
[HttpGet]
[Route("api/home/student")]
public string GetStudent(int id)
{
    return "student is " + id;
}
[HttpGet]
[Route("api/home/instructor")]
public string GetInstructor(int id)
{
    return "Instructor is "+id;
}
```

Web API Routing(cont.)

- **Attribute Routing**

We Can solve this by attribute route as the following

localhost:50630/api/home/student?id=1

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">student is 1</string>
```

localhost:50630/api/home/instructor?id=1

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">Instructor is 1</string>
```

Web API Routing(cont.)

- **Attribute Routing (Route prefix)**

route prefix can include parameters:

```
[RoutePrefix("customers/{customerId}")]
public class OrdersController : ApiController
{
    // GET customers/1/orders
    [Route("orders")]
    public IEnumerable<Order> Get(int customerId) { ... }
}
```

Override the route prefix

- Use a tilde (~) on the method attribute to override the route prefix

```
[RoutePrefix("api/books")]
public class BooksController : ApiController
{
    // GET /api/authors/1/books
    [Route("~/api/authors/{authorId:int}/books")]
    public IEnumerable<Book> GetByAuthor(int authorId) { ... }

    // ...
}
```

Web API Parameter Binding

- Action methods in Web API controller can have one or more parameters of different types.
- It can be either **primitive** type or **complex** type.
- Web API binds action method parameters either with URL's **query string** or with **request body** depending on the parameter type.
- By default, if parameter type is of .NET **primitive** type such as int, bool, double, string, GUID, DateTime, decimal then sets the value of a parameter from the **query string**.
- And if the parameter type is **complex** type then Web API tries to get the value from **request body** by default.

Web API Parameter Binding (cont.)

- **Get Action Method with Primitive Parameter**
  - Consider the following example of Get action method that includes single primitive type parameter.

```
public Student Get(int id)
{

}
```

- Followings are valid HTTP GET Requests for the above action method.
  - http://localhost/api/student?id=1
  - http://localhost/api/student?ID=1

Web API Parameter Binding(cont.)

- **POST** Action Method with Primitive Parameter
- HTTP POST request is used to create new resource. It can include request data into HTTP request body and also in query string.
- Consider the following Post action method.
-

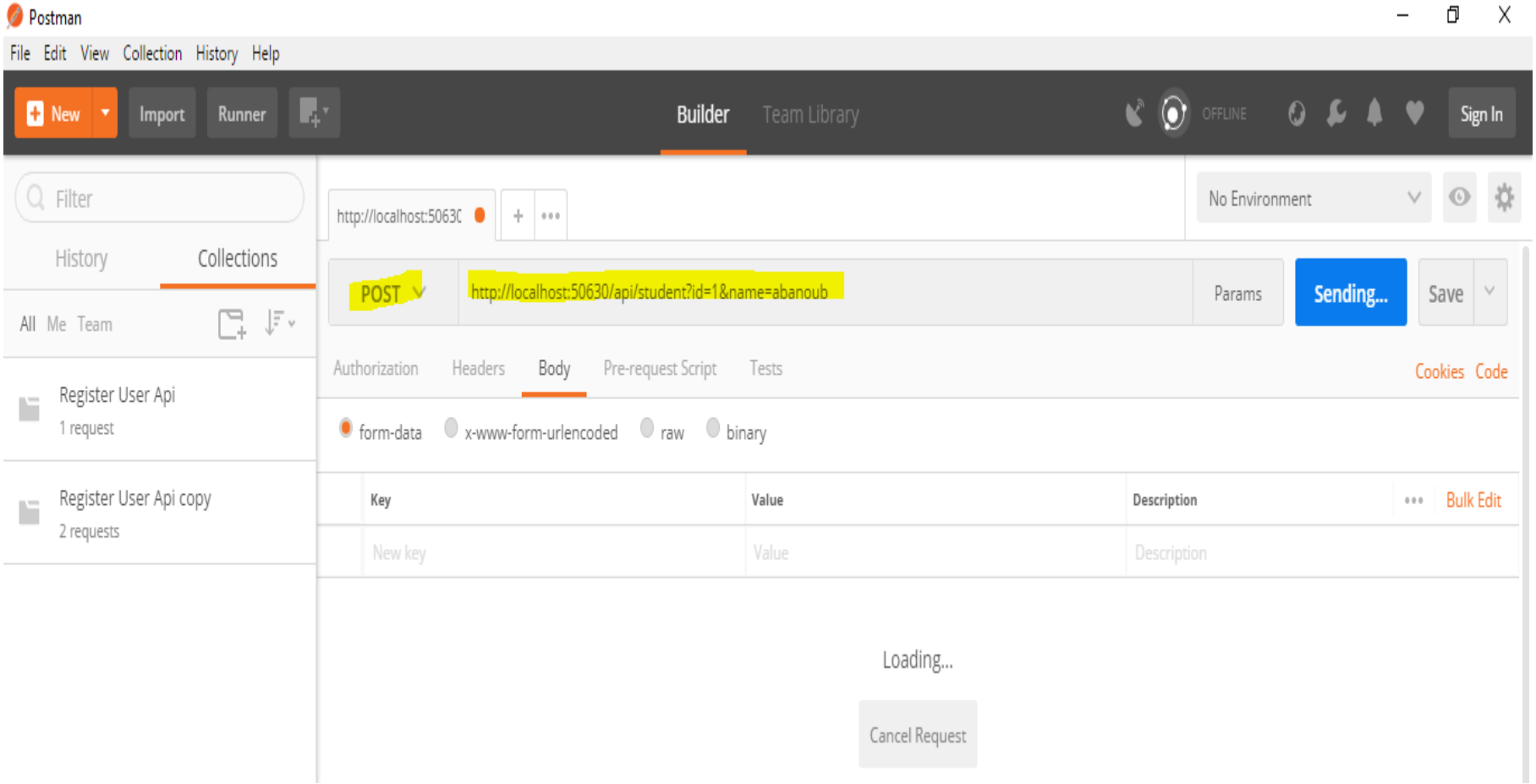
```
public class StudentController : ApiController
{
                    public Student Post(id id, string name)
    {

    }
}
```

Web API Parameter Binding(cont.)

- Bowser only support GET() verbs so now we will use **postman**.
- As you can see above, Post() action method includes primitive type parameters id and name. So, by default, Web API will get values from the query string. For example, if an HTTP POST request
  is **http://localhost/api/student?id=1&name=steve** then the value of id will be 1 and name will be "steve" in the above Post() method.

Web API Parameter Binding(cont.)

- Bowser only support GET() verbs so now we will use



Web API Parameter Binding(cont.)

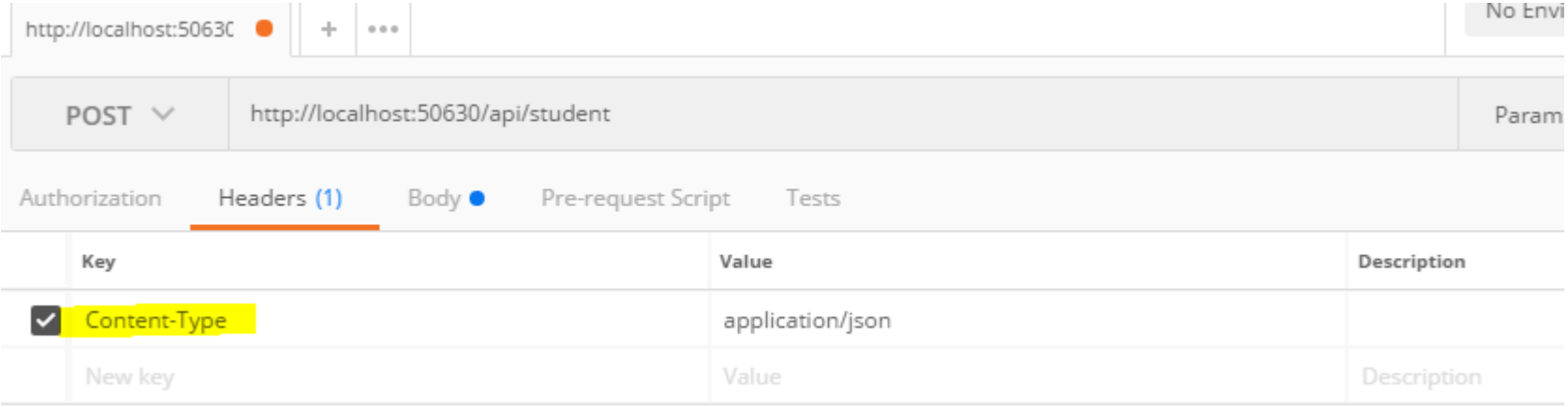
- Now, consider the following Post() method with complex type parameter.

Example: Post Method with Complex Type Parameter

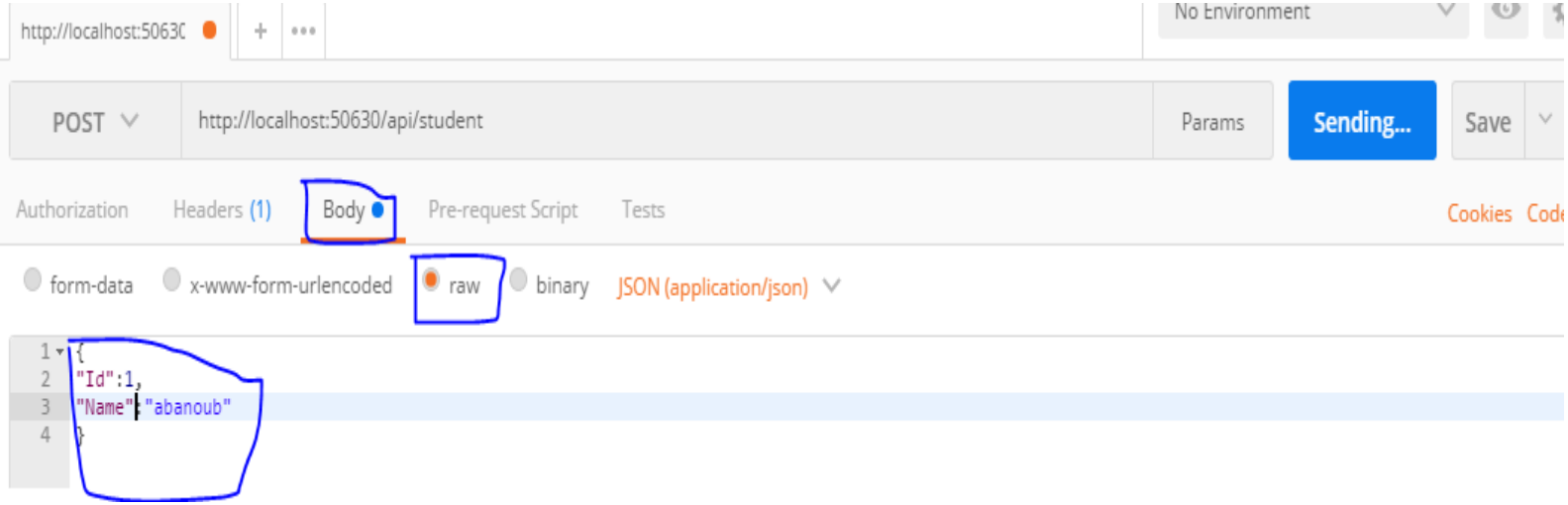
```
public class Student
{
                    public int Id { get; set; }
                    public string Name { get; set; }
}

public class StudentController : ApiController
{
                    public Student Post(Student stud)
    {

    }
}
```

Web API Parameter Binding(cont.)

- In postman
- 1-Set content-type of the request to application/json
-



Web API Parameter Binding(cont.)

- In postman
- 2-pass the object in the body raw
- 

Web API Parameter Binding(cont.)

- In postman
- 3-The action look like the following
-

```
public string post(Student std)
{
    return "You create student with id " + std.ID;
}
```

Web API Parameter Binding(cont.)

- **[FromUri] and [FromBody]**
- You have seen that by default Web API gets the value of a **primitive** parameter from the **query string** and **complex** type parameter from the **request body**.
- But, what if we want to change this default behaviour?
- Use **[FromUri]** attribute to force Web API to get the value of complex type from the **query string** and
- **[FromBody]** attribute to get the value of primitive type from the **request body**, opposite to the default rules.

Web API Parameter Binding(cont.)

- **[FromUri] and [FromBody]**
- For example, consider the following Get method.

```
public class StudentController : ApiController
{
    public Student Get([FromUri] Student stud)
    {

    }
}
```

Web API Parameter Binding(cont.)

- **[FromUri] and [FromBody]**
- In the above example, **Get** method includes complex type parameter with [**FromUri**] attribute. So, Web API will try to get the value of Student type parameter from the query string.
- For example, if an HTTP GET request **http://localhost:xxxx/api/student?id=1&name=steve** then Web API will create Student object and set its id and name property values to the value of id and name query string

Web API Parameter Binding(cont.)

- **[FromUri] and [FromBody]**

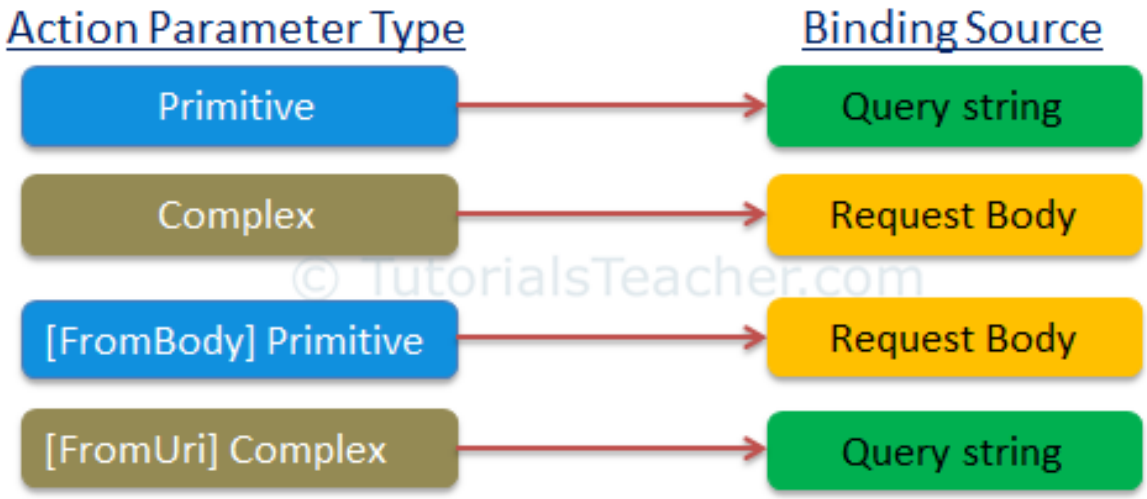
```
public class StudentController : ApiController
{
    public Student Post([FromBody]string name)
    {

    }
}
```

- Here you must pass the object in the **body request.**

Web API Parameter Binding(cont.)

- **[FromUri] and [FromBody]**

The following figure summarizes parameter binding rules.



Web API Parameter Bindings

Action Method Return Type

- The Web API action method can have following return types.

1-Void

2-Primitive type or Complex type

3-HttpResponseMessage

4-IHttpActionResult

Action Method Return Type(cont.)

- **Void**
- It's not necessary that all action methods must return something.
- It can have **void** return type.
- For example, consider the following Delete action method that just deletes the student from the data source and returns nothing.

```
public class StudentController : ApiController
{
    public void Delete(int id)
    {
        DeleteStudentFromDB(id);
    }
}
```

Action Method Return Type(cont.)

- **Primitive or Complex Type**
- An action method can return primitive or other custom complex types as other normal methods.

```
public int GetId(string name)
{
    int id = GetStudentId(name);

    return id;
}

public Student GetStudent(int id)
{
    var student = GetStudentFromDB(id);

    return student;
}
```

Action Method Return Type(cont.)

- **HttpResponseMessage**
- Web API controller always returns an **object** of **HttpResponseMessage** to the hosting infrastructure.
- The following figure illustrates the overall Web API request/response pipeline.

Web API Request Pipeline

Action Method Return Type(cont.)

- **HttpResponseMessage**
- As you can see in the above figure, the Web API controller returns **HttpResponseMessage** object. You can also create and return an object of **HttpResponseMessage** directly from an action method.
- The advantage of sending HttpResponseMessage from an action method is that you can configure a response your way.
- You can set the status code, content or error message (if any) as per your requirement.

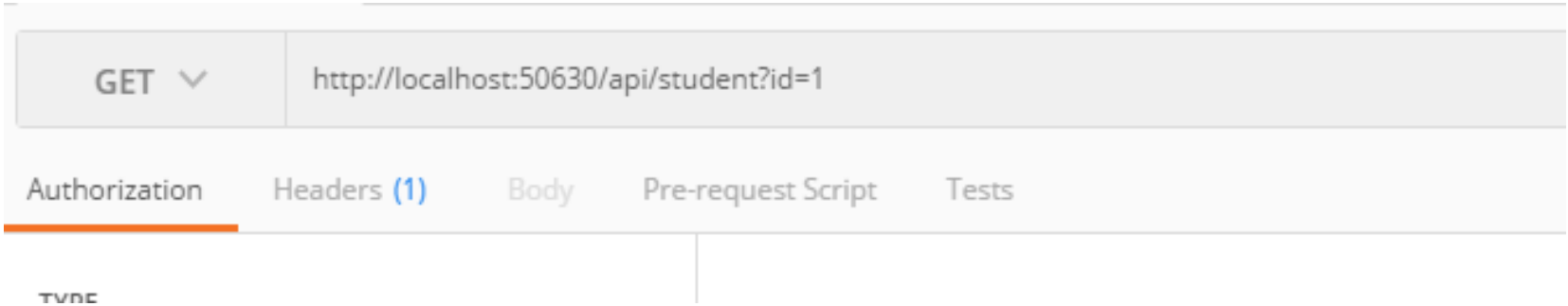
Action Method Return Type(cont.)

- **HttpResponseMessage**
- **Consider the following example**

```
public string getStudent(int id)
{
    var std= stdList.Where(s => s.ID == id).FirstOrDefault();
    if (std == null)
    {
        return null;
    }
    else
    {
        return std.Name;
    }

    |

}
```

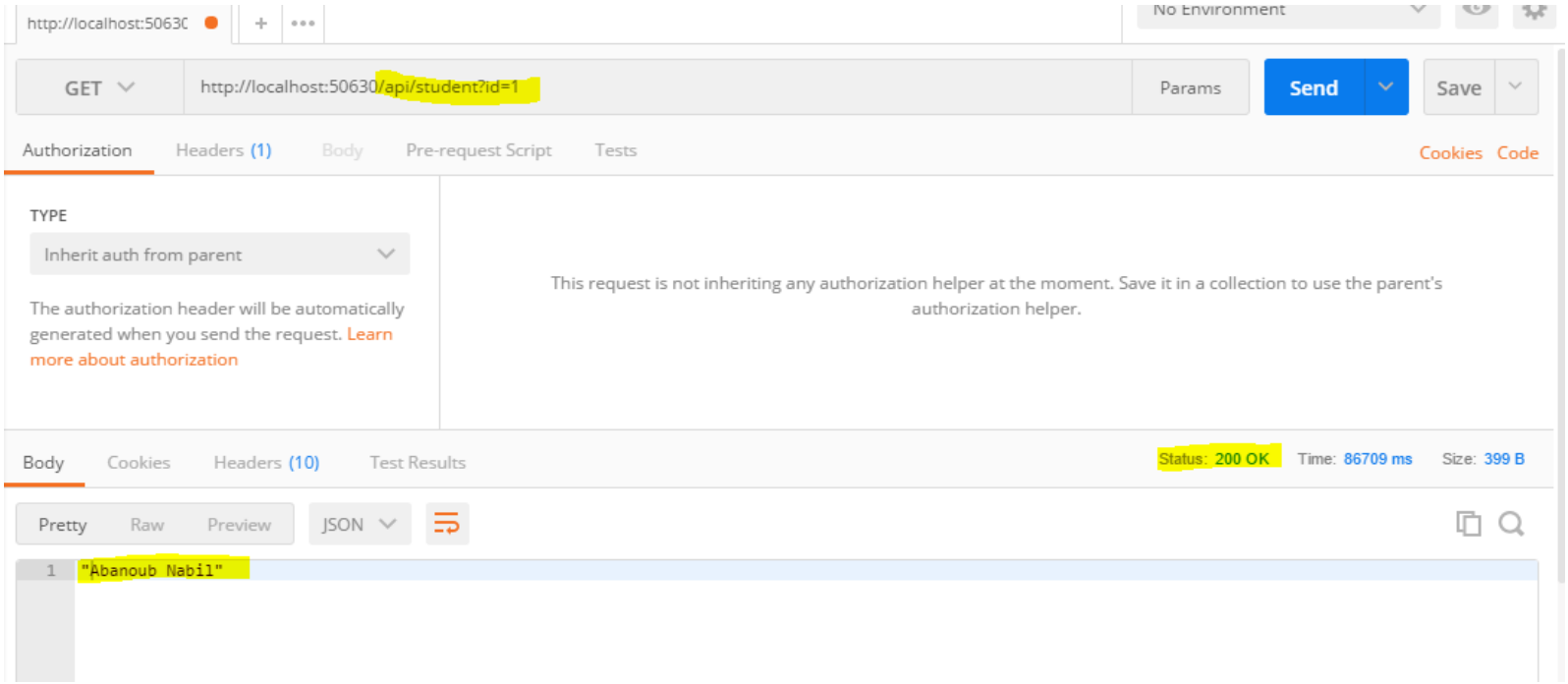
Action Method Return Type(cont.)

- **HttpResponseMessage**
- Then call the method from postman with existing student with the specified id.



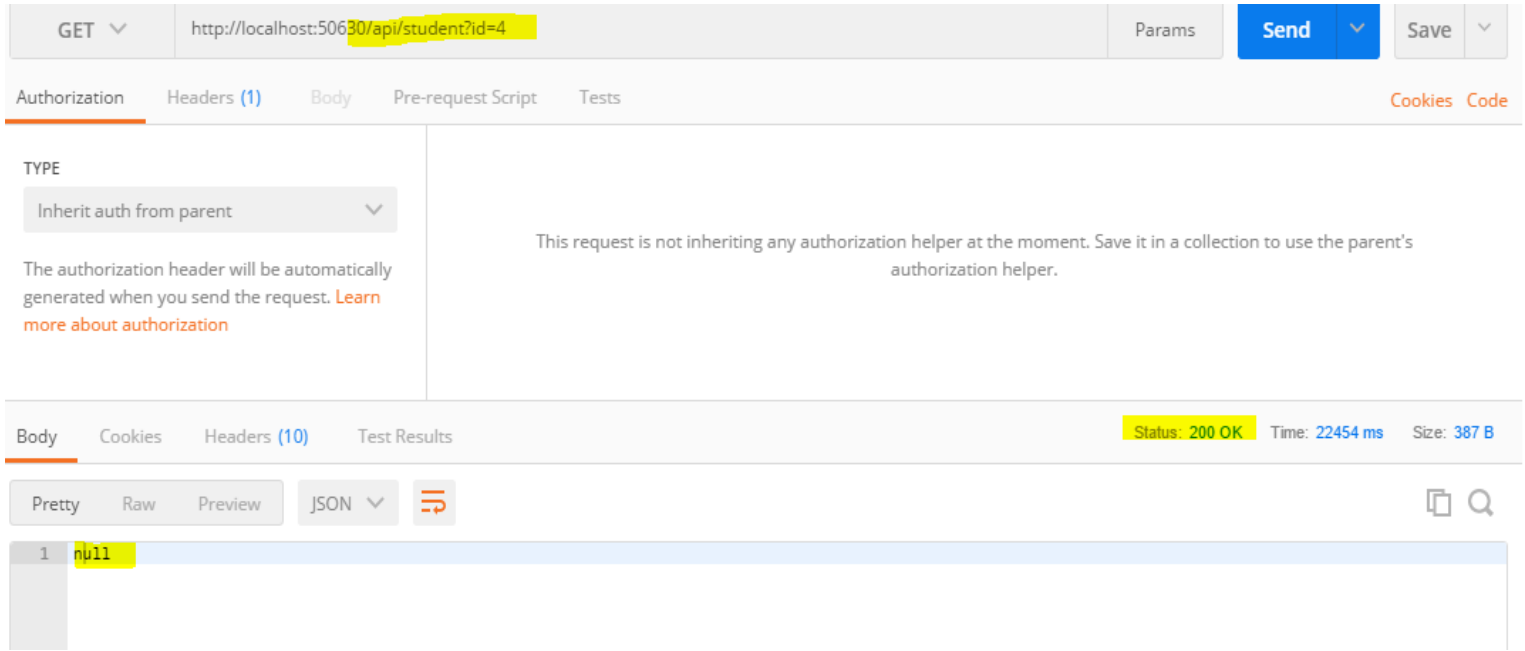
Action Method Return Type(cont.)

- **HttpResponseMessage**
- The response from the method in the postman will like this.



Action Method Return Type(cont.)

- **HttpResponseMessage**
- Then call the method from postman with not existing student with the specified id. The response will be



Action Method Return Type(cont.)

- **HttpResponseMessage**
- As you see also here the statuse code is 200 ok. But this is not true case so we must handle this byy using **HttpResponseMessage**
- The method will be like this.

```
public HttpResponseMessage getStudent(int id)
{
    var std= stdList.Where(s => s.ID == id).FirstOrDefault();
    if (std == null)
    {
        var res = "There is no student with id= " + id;
        return Request.CreateResponse(HttpStatusCode.NotFound, res);
    }
    else
    {
        return Request.CreateResponse(HttpStatusCode.OK,std);
    }



}
```

Action Method Return Type(cont.)

- **IHttpActionResult**
- The ***IHttpActionResult*** was introduced in Web API 2 (.NET 4.5).
- An action method in Web API 2 can return an implementation of **IHttpActionResult** class which is more or less similar to ActionResult class in ASP.NET MVC.

Action Method Return Type(cont.)

- **IHttpActionResult**

```
public IHttpActionResult getStudent(int id)
{
    var std= stdList.Where(s => s.ID == id).FirstOrDefault();
    if (std == null)
    {
        var res = "There is no student with id= " + id;
        // return Request.CreateResponse(HttpStatusCode.NotFound, res);
        return NotFound();
    }
    else
    {
        // return Request.CreateResponse(HttpStatusCode.OK,std);
        return Ok(std);
    }
```

Action Method Return Type(cont.)

ApiController Method	Description
BadRequest()	Creates a BadRequestResult object with status code 400.
Conflict()	Creates a ConflictResult object with status code 409.
Content()	Creates a NegotiatedContentResult with the specified status code and data.
Created()	Creates a CreatedNegotiatedContentResult with status code 201 Created.
CreatedAtRoute()	Creates a CreatedAtRouteNegotiatedContentResult with status code 201 created.
InternalServerError()	Creates an InternalServerErrorResult with status code 500 Internal server error.
NotFound()	Creates a NotFoundResult with status code404.

Action Method Return Type(cont.)

- **IHttpActionResult**

Ok()	Creates an OkResult with status code 200.
Redirect()	Creates a RedirectResult with status code 302.
RedirectToRoute()	Creates a RedirectToRouteResult with status code 302.
ResponseMessage()	Creates a ResponseMessageResult with the specified HttpResponseMessage.
StatusCode()	Creates a StatusCodeResult with the specified http status code.
Unauthorized()	Creates an UnauthorizedResult with status code 401.

Web API Request/Response Data Formats

- Here, you will learn how Web API **handles** different formats of **request and response data.**

Media Type:

- Media type (aka **MIME** type) specifies the format of the data as **type/subtype** e.g. text/html, text/xml, application/json, image/jpeg etc.

Web API Request/Response Data Formats(cont.)

- **In HTTP request**, MIME type is specified in the request header using **Accept** and **Content-Type** attribute.
- The **Accept header** attribute **specifies** the format of **response** data which the client expects and the
- **Content-Type** header attribute **specifies** the format of the data in the **request** body so that receiver can parse it into appropriate format.
- **For example,** if a client wants response data in JSON format then it will send following GET HTTP request with Accept header to the Web API.

Web API Request/Response Data Formats(cont.)

- The same way, if a client includes **JSON** data in the **request body** to send it to the receiver then it will send following POST HTTP request with **Content-Type** header with JSON data in the body.

```
Content-Type: application/json
Content-Length: 13

{
  id:1,
  name:'Steve'
}
```

ASP.NET Web API: Media-Type Formatters

- As you have seen in the previous section that **Web API handles JSON and XML formats** based on Accept and Content-Type headers.
- **But, how does it handle these different formats?** The answer is: By using **Media-Type formatters.**

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

Treat with database

- Let us create code first for product and category
- 1-Create product class.

```
public class Product
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Price { get; set; }
}
```

Treat with database(cont.)

- Let us create code first for product and category
- 2-Create Category class.

```
public class Category
{
    public int ID { get; set; }
    public string Name { get; set; }
}
```

Treat with database(cont.)

- Let us create code first for product and category
- 3-Make the relations between them.

```
public class Product
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Price { get; set; }

    //Relations

    public virtual Category Category { get; set; }
}
```

```
public class Category
{
    public int ID { get; set; }
    public string Name { get; set; }
    //Relations
    public virtual ICollection<Product> Products { get; set; }
}
```

Serialization

- Let us create code first for product and category
- 4-Define the connection string in the web.config file.

```
<connectionStrings> <add name=“TestAPI"
connectionString="Data Source=.;Initial
Catalog=ProdTest Integrated  Security=true"
providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Treat with database(cont.)

- Let us create code first for product and category
- **5-**Install Entity Framework.

- **6-**Make the context class.

```
public class EcommerceDBContext:DbContext
{
    public EcommerceDBContext():base("TestProduct")
    {

    }

    public DbSet<Product> Products { get; set; }
    public DbSet<Category> Categories { get; set; }
}
```

Treat with database(cont.)

- Let us create code first for product and category
- **6-**Make a scaffolding controller to category and then test in postman.
- **7**-Make a scaffolding controller to Product and then test in postman.

Serialization

- Now when you add product to a specific category as the following


```
{
"Id":1,
"Name":"Dell",
"Price":7000,
"Category":
{
    "ID":1
}
}
```


- you will catch the following error as the following


```
{
    "Message": "An error has occurred.",
    "ExceptionMessage": "The 'ObjectContent`1' type failed to serialize the response body for content type 'applica
    "ExceptionType": "System.InvalidOperationException",
    "StackTrace": null,
    "InnerException": {
        "Message": "An error has occurred.",
        "ExceptionMessage": "Self referencing loop detected with type 'FIRSTAPI_PROJ.Models.Product'. Path 'Categor
        "ExceptionType": "Newtonsoft.Json.JsonSerializationException",
        "StackTrace": "   at Newtonsoft.Json.Serialization.JsonSerializerInternalWriter.CheckForCircularReference(J
            JsonProperty property, JsonContract contract, JsonContainerContract containerContract, JsonProperty con
```

Serialization(cont.)

- So you need to solve this problem.
- 1-Use JsonIgnore

```
public class Category
{
    public int ID { get; set; }
    public string Name { get; set; }
    //Relations
    [JsonIgnore]
    public virtual ICollection<Product> Products { get; set; }
}
```

Serialization(cont.)

- So you need to solve this problem.
- Now test the post product method after update the following

```
// POST: api/Products
[ResponseType(typeof(Product))]
public IHttpActionResult PostProduct(Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    var cat =
          (from cates in db.Categories
           where cates.ID == product.Category.ID
           select cates).FirstOrDefault();
    product.Category = cat;
    db.Products.Add(product);
    db.SaveChanges();

    return CreatedAtRoute("DefaultApi", new { id = product.ID }, product);
}
```

Serialization(cont.)

- So you need to solve this problem.
- Now test the post product method after update the following

```
// POST: api/Products
[ResponseType(typeof(Product))]
public IHttpActionResult PostProduct(Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    var cat =
          (from cates in db.Categories
           where cates.ID == product.Category.ID
           select cates).FirstOrDefault();
    product.Category = cat;
    db.Products.Add(product);
    db.SaveChanges();

    return CreatedAtRoute("DefaultApi", new { id = product.ID }, product);
}
```

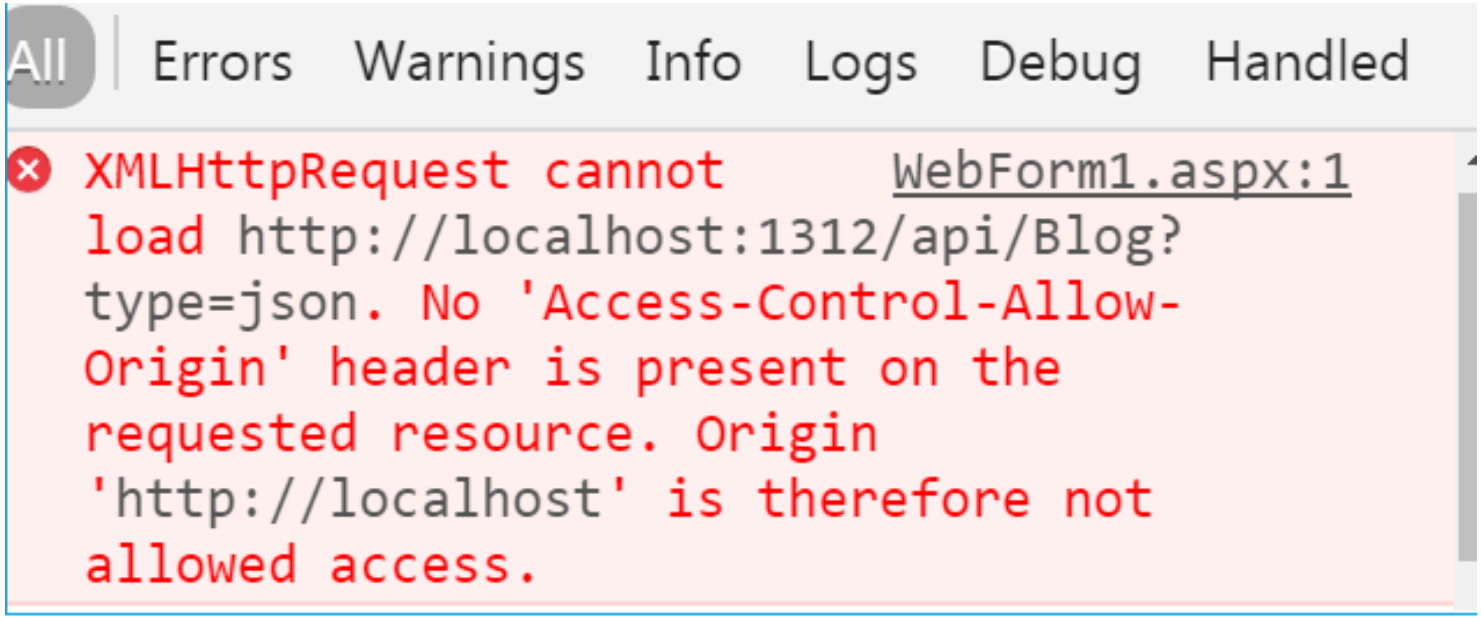
Consume WEB API .

- First create html page with ajax call to consume the web api.

```
<input type="button" value="Get Data" onclick="GetData()"/>

<script>
    function GetData()
    {
        //alert("")
        $.ajax({
            type: 'get',
            contentType: 'application/json',
            dataType:'jsonp',
            url: 'http://localhost:50630/api/categories',
            success: function (data) {
                console.log(data);

            },

        });
    }



</script>
```

Consume WEB API .

- Now if you click on get button you will find the following error.



- This happens because of **CORS** problem.

What is CORS?

- **CORS** stands for **C**ross-**O**rigin **R**esource **S**haring.
- 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.
- A web page may freely embed images, stylesheets, scripts, iframes, and videos.
-

What is CORS?(cont.)

- For security reasons, browsers restrict cross-origin HTTP requests initiated from within scripts. For example, XMLHttpRequest follows the same-origin policy. So, a web application using XMLHttpRequest could only make HTTP requests to its own domain. To improve web applications, developers asked browser vendors to allow XMLHttpRequest to make cross-domain requests.

What is CORS?(cont.)

- **What is same origin policy?**
- Browsers allow a web page to make AJAX requests only within the **same domain**. Browser security prevents a web page from making AJAX requests to another domain. This is called origin policy.

- We can **enable CORS** in WebAPI,

  1-Using JSONP

  2-Using Microsoft.AspNet.WebApi.Cors

What is CORS?(cont.)

- **Using JSONP(JSON with Padding) formatter**
- **What is JSONP?**
- JSONP stands for **JSON with Padding**.
- It helps to implement cross-domain request by browser's same-origin-policy.
- It wraps up a JSON response into a JavaScript function (callback function) and sends that back as a Script to the browser.
- This allows you to bypass the same origin policy and load JSON from an external server into the JavaScript on your webpage.

What is CORS?(cont.)

- **Using JSONP(JSON with Padding) formatter**
- **What is JSONP?**

```
// JSON:
{
    'name' : 'manas',
    'age' : 23,
    'temper' : 'moody'
}
```

With JSONP, when the server receives the "callback" parameter, it wraps up the result a little differently, and returns like this.

```
// JSONP:
newPerson({
  'name' : 'manas',
  'age' : 23,
  'temper' : 'moody'
});
```

What is CORS?(cont.)

- **Using JSONP(JSON with Padding) formatter**
- **What is JSONP?**

First, we need to enable CORS in WebAPI, then we call the service from other application AJAX request. In order to enable CORS, we need to install the JSONP package from NuGet (see Figure3).

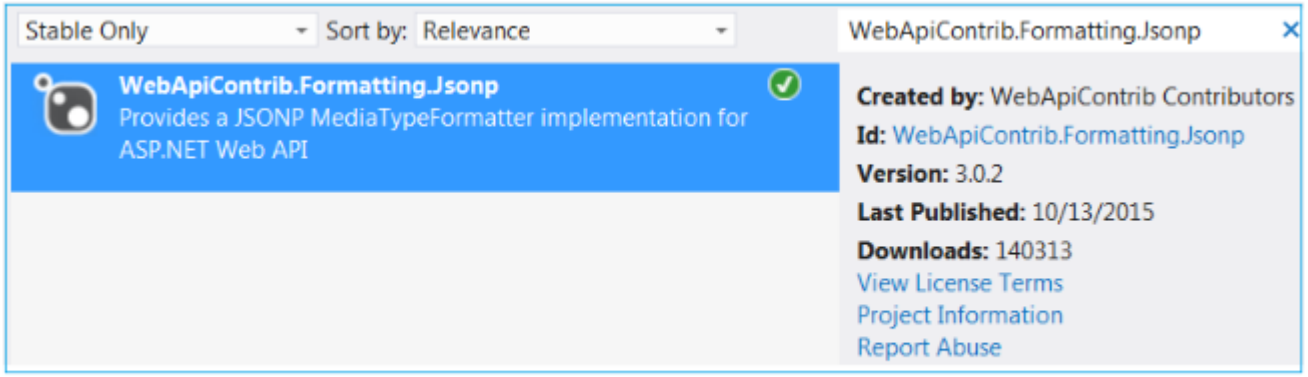


Figure 3: Adding Jsonp package from NuGet

What is CORS?(cont.)

- **Using JSONP(JSON with Padding) formatter**
- **What is JSONP?**

After adding Jsonp package, we need to add the following code-snippet in App_Start\WebApiConfig.cs file. It creates instance of *JsonpMediaTypeFormatter* class and adds to config formatters object.

```
var jsonpFormatter = new
JsonpMediaTypeFormatter(config.Formatters.JsonFormatter);
config.Formatters.Add(jsonpFormatter);
```

What is CORS?(cont.)

- **Using JSONP(JSON with Padding) formatter**
- **What is JSONP?**

Now, we are ready with CORS enabling in Server and the other application needs to send AJAX requests which are hosted on another domain. In the below code snippet, it sends datatype as *jsonp* which works for cross domain requests.

What is CORS?(cont.)

- **Using JSONP(JSON with Padding) formatter**
- **Disadvantages**
- Incompatible in old browser
- Security issue which can steal your cookies(leads to a whole bunch of potential problems).

What is CORS?(cont.)

- **Using JSONP(JSON with Padding) formatter**
- **Using Microsoft.AspNet.WebApi.Cors**
- 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**
-

What is CORS?(cont.)

- **Using JSONP(JSON with Padding) formatter**
- **Using Microsoft.AspNet.WebApi.Cors**
- ***Origins***
  Here, we need to set Origins which means from which domain the requests will accept.

- ***Request Headers***
  The Request header parameter specifies which Request headers are allowed. To allow any header set value to "*"

-

What is CORS?(cont.)

- **Using JSONP(JSON with Padding) formatter**
- **Using Microsoft.AspNet.WebApi.Cors**
- ***HTTP Methods***
  The methods parameter specifies which HTTP methods are allowed to access the resource.
- Use comma-separated values when you have multiple HTTP methods like "get,put,post". To allow all HTTP methods, use the wildcard value "*".
-

What is CORS?(cont.)

- **Using JSONP(JSON with Padding) formatter**
- **Using Microsoft.AspNet.WebApi.Cors**
- Then add the following code to register method in WebApiConfig.cs

```
EnableCorsAttribute cors = new
EnableCorsAttribute("*", "*", "*","");
 config.EnableCors(cors);
```

What is CORS?(cont.)

- **Using JSONP(JSON with Padding) formatter**
- **Disable CORS**
- Suppose you enabled CORS in Global or Controller level then all actions are enabled for CORS. However, if you want CORS to not be enabled for couple of actions due to some security reason, here *DisableCors* attribute plays vital role to Disable CORS which means other domains can't call the action.

```
[DisableCors()]
// GET api/values/5
public string Get(int id)
{
    return "value";
}
```

Web API Routing

- Web API routing is similar to ASP.NET MVC Routing.
- It routes an incoming HTTP request to a particular action method on a Web API controller.
- Web API supports two types of routing:
  - Convention-based Routing
  - Attribute Routing