**Quickreference WEB API**

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.

**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's make to know what other actions can be performed on the resource.

**Content Negotiation:**

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.

The formatters are used by the server for both request and response messages. When the client sends a request to the server, we set the Content-Type header to the appropriate value to let the server know the format of the data that we are sending. For example, if the client is sending JSON data, the Content-Type header is set to application/json. The server knows it is dealing with JSON data, so it uses JSON formatter to convert JSON data to .NET Type. Similarly when a response is being sent from the server to the client, depending on the Accept header value, the appropriate formatter is used to convert .NET type to JSON, XML etc.  
  
It's also very easy to change the serialization settings of these formatters. For example, if you want the JSON data to be properly indented and use camel case instead of pascal case for property names, all you have to do is modify the serialization settings of JSON formatters as shown below. For example below code goes in WebApiConfig.cs file in App\_Start folder.

config.Formatters.JsonFormatter.SerializerSettings.Formatting =

                            Newtonsoft.Json.Formatting.Indented;

config.Formatters.JsonFormatter.SerializerSettings.ContractResolver =

    new CamelCasePropertyNamesContractResolver();

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

**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);

**Return only XML 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 JsonFormatter which forces ASP.NET Web API to always return XML irrespective of the Accept header value in the client request. Use this technique when you want your service to support only XML and not JSON.  
  
config.Formatters.Remove(config.Formatters.JsonFormatter);  
  
With this change, irrespective of the Accept header value (application/xml or application/json), the Web API service is always going to return XML.

**Custom method names in an 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.

 Even if you rename it to 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

To instruct Web API to map HTTP verb GET to ABC() method, decorate the method with [HttpGet] attribute.

[HttpGet]

public IEnumerable<T> ABC()

{

    //Code goes here

}

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 |

### Web API query string parameters

Depending on the value we specify for query string parameter gender, the Get() method should return the data.

| **Query String** | **ExpectedData** |
| --- | --- |
| http://localhost/api/students?gender=All | All students |
| http://localhost/api/ students?gender=Male | Only Male students |
| http://localhost/api/ students?gender=Female | Only Female students |

**ATTRIBUTE ROUTING CONSTRAINTS**

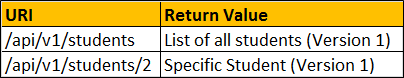
|  |  |  |
| --- | --- | --- |
| **Constraint** | **Description** | **Example** |
| min | Matches an integer with a minimum value | {x:min(0)} |
| max | Matches an integer with a maximum value | {x:max(100)} |
| length | Matches a string with the specified length or within a specified range of lengths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

IHttpAction Results

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

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

[Web API versioning using URI](http://csharp-video-tutorials.blogspot.com/2017/02/web-api-versioning-using-uri.html)

Clients of our Version1 service can use the following URLs to get either the list of all students or a specific student by Id. At the moment, the student objects the service return has Id and Name properties.   


Web API versioning using querystring parameter

**Here is what we want**

|  |  |
| --- | --- |
| **URI** | **Should Return** |
| /api/students?v=1 | Version 1 Students |
| /api/students?v=2 | Version 2 Students |

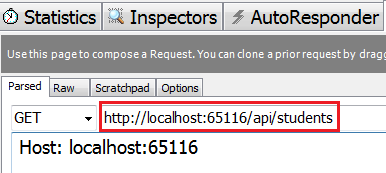
When a request is issued to the following URI, depending on the query string parameter "v" value we want to select the controller. If the value is 1, select StudentsV1Controller, and if it is 2, then select StudentsV2Controller.  
/api/students?v=1 

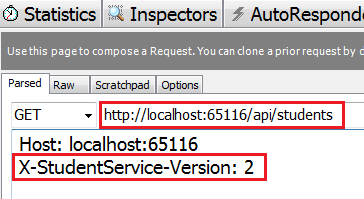
|  |  |
| --- | --- |
| **Query String "v" value** | **Controller to Select** |
| v=1 | StudentsV1Controller |
| v=2 | StudentsV2Controller |

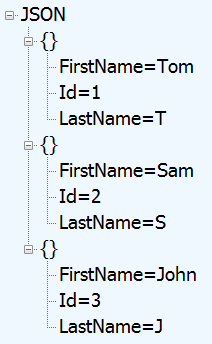
<http://localhost:16399//api/students/1?v=1>

<http://localhost:16399//api/students/1?v=2>

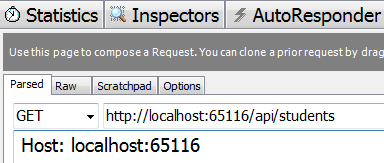
### Web API versioning using a custom header

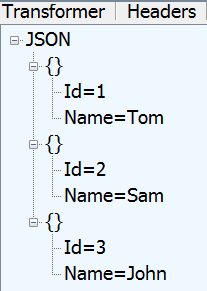






### Web API versioning using accept header



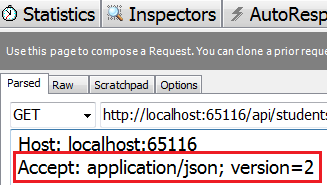
Response from the Web API service  


On the other hand if we specify the version parameter as part of the Accept header, we get the specified version as expected.

<http://localhost:16399/api/students>

Accept: application/json; Version=2

Host: localhost:16399



### Web API versioning using custom media types

### GET <http://localhost:xxxx/api/students>

### Accept: application/vnd.mysession.students.v1+json

### Response from the web api service  web api versioning vendor specific media types

### However, if we specify that we want xml format instead of json in the request as shown below. We still get JSON formatted result instead of XML formatted result.

### GET <http://localhost:xxxx/api/students>

### Accept: application/vnd.mysession.students.v1+xml

### 

### Accept: application/vnd.mysession.students.v1+json

**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 |

code in Register() method of WebApiConfig.cs file in App\_Start folder, enables CORS globally for the entire application i.e for all controllers and action methods 

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

config.EnableCors();

EnableCors attribute can be applied on a **specific controller** or **controller method.**  
  
If applied at a controller level then it is applicable for all methods in the controller. To apply it at the controller level  
  
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 StudentsController : 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.

