ASP.NET Web API

- API stands for ‘Application Programming Interface’.
- API’s are a set of functions and procedures that allow for the creation of applications that access data and features of other applications, services, or operating systems.
- 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. IOT’s.
- **What are IOT?**
  The term IOT stands for Internet Of Things. Internet Of Things are the objects or devices that have an IP address and can communicate over the internet with other internet enabled devices and objects. Examples for IOT include security systems, electronic appliances, thermostats, cars etc..., in addition to desktops, laptops, and smart phones.
- ASP.NET Web API framework is widely used to create RESTful services and it can also be used to create non restful API’s.
- **What are RESTful services**
- REST stands for Representational State Transfer.
- 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.

1) **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.

2) **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.

3) **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.

4) **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)
- **Difference between WCF and Web API. When to choose one over the other?**
  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. For example, you want to build a single service, that can be consumed by 2 different clients - Let's say, a Java client and .NET client. Java client expects transport protocol to be HTTP and message format to be XML for interoperability, where as the .NET client expects the protocol to be TCP and the message format to be binary for performance. For this scenario WCF is the right choice. What we do here is create a single WCF service, and then configure 2 end points one for each client (i.e one for the Java client and the other for the .NET client)
- 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.
- To create a ASP.Net web API project first select ASP.NET web application then select web API.

HTTP GET PUT POST DELETE

CRUD	HTTP Verb
Create	POST
Read	GET
Update	PUT
Delete	DELETE

- Terms and concepts related to HTTP request and response system:
- **Request Verbs :** These HTTP verbs (GET, POST, PUT & DELETE) describe what should be done with the resource. For example do you want to create, read,

update or delete an entity. GET, PUT, POST and DELETE http verbs are the most commonly used one's. For the complete list of the HTTP verbs, please check http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

- **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. For the complete list of HTTP status codes and what they mean, please visit http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html
- We will use a tool called fiddler to perform POST, PUT & DELETE actions.

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. In this case, the server picks the first formatter which is a JSON formatter and formats the data in JSON.
  Accept: application/xml,application/json
- You can also specify quality factor. In the example below, xml has higher quality factor than json, so the server uses XML formatter and formats the data in XML.
  application/xml;q=0.8,application/json;q=0.5
- If you don't specify the Accept header, by default the Web API returns JSON data.
- When the response is being sent to the client in the requested format, notice that the Content-Type header of the response is set to the appropriate value. For example, if the client has requested application/xml, the server send the data in XML format and also sets the Content-Type=application/xml.
- The formatters are used by the server for both request and response messages.

- 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. With our example this 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();
```

ASP.NET web API Media Type Formatter

- **What is Media typr formatter?**
  MediaTypeFormatter is an abstract class from which XMLMediaTypeformatter and JSONMediaTypeFormatter inherits from.
  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"

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.
- 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.
- To instruct Web API to map HTTP verb GET to method name which doen't contain get word,then decorate the method with [HttpGet] attribute.
- 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

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

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

If the value for gender is not Male, Female or All, then the service should return status code 400 Bad Request. For example, if we specify **ABC** as the value for gender, then the service should return status code 400 Bad Request with the following message.
Value for gender must be Male, Female or All. ABC is invalid.

- **Below is the modified Get() method**
    1. Gender is being passed as a parameter to the Get() method
    2. Default value is "All". The default value makes the parameter optional
    3. The gender parameter of the Get() method is mapped to the gender parameter sent in the query string

```
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

- When a PUT request is issued, 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)
{
```

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
- Browsers does not allow cross domain ajax requests. There are 2 ways to get around this problem

A) Using JSONP (JSON with Padding)

B) 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.
- **CORS** (Cross Origin Resource Sharing)

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

- For more info go through below link

https://csharp-video-tutorials.blogspot.com/2016/09/cross-origin-resource-sharing-aspnet.html

Enable SSL in Visual Studio Development Server

- To launch a service in https we need to enable ssl for the projrct by selecting ssl = true from proprties window.
- For more info https://csharp-video-tutorials.blogspot.com/2016/09/enable-ssl-in-visual-studio-development.html

Implementing basic authentication in ASP.NET Web API

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")
{
```

- For more info

https://csharp-video-tutorials.blogspot.com/2016/10/implementing-basic-authentication-in.html

Attribute Routing

- **What is Attribute Routing**
  Using the [Route] attribute to define routes is called Attribute Routing
- **What are the advantages of using Attribute Routing**
  Attribute routing gives us more control over the URIs than convention-based routing. Creating URI patterns like hierarchies of resources (For example, students have courses, Departments have employees) is very difficult with convention-based routing. With attribute routing all you have to do is use the [Route] attribute as shown below.

  ```
  [Route("api/students/{id}/courses")]
  ```
- **How to enable Attribute Routing**
  In ASP.NET Web API 2, Attribute Routing is enabled by default. The following line of code in WebApiConfig.cs file enables Attribute Routing.

  ```
  config.MapHttpAttributeRoutes();
  ```
- **Can we use both Attribute Routing and Convention-based routing in a single Web API project**
  Yes, both the routing mechanisms can be combined in a single Web API project. The controller action methods that have the [Route] attribute uses Attribute Routing, and the others without [Route] attribute uses Convention-based routing.

RoutePrefix attribute in Web API

- **RoutePrefix attribute :** As you can see from the example below, all the routes in the **StudentsController** start with the same prefix - api/students

```
public class StudentsController : ApiController
{
   [Route("api/students")]
   public IEnumerable<Student> Get()

   [Route("api/students/{id}")]
   public Student Get(int id)
```

```
    [Route("api/students/{id}/courses")]
    public IEnumerable<string> GetStudentCourses(int id)
}
```

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

```
[RoutePrefix("api/students")]
public class StudentsController : ApiController
{
    [Route("api/students")]
    public IEnumerable<Student> Get()

    [Route("api/students/{id}")]
    public Student Get(int id)

    [Route("api/students/{id}/courses")]
    public IEnumerable<string> GetStudentCourses(int id)
}
```

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

Web API attribute routing constraints

- To specify route constraint, the syntax is "{parameter:constraint}". With these constraints in place, if the parameter segment in the URI is an integer, then Get(int id) method with integer parameter is invoked, if it is a string then Get(string name) method with string parameter is invoked.

- Please note that **"alpha"** stands for uppercase or lowercase alphabet characters. Along with int and alpha, we also have constraints like decimal, double, float, long, bool etc. Please check MSDN for the full list of available constraints.

```
[Route("{id:int}")]
public Student Get(int id)
{
    return students.FirstOrDefault(s => s.Id == id);
}

[Route("{name:alpha}")]
public Student Get(string name)
{
    return students.FirstOrDefault(s => s.Name.ToLower() == name.ToLower());
}
```

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

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

minlength	Matches a string with a minimum length	{x:minlength(1)}
maxlength	Matches a string with a maximum length	{x:maxlength(100)}
range	Matches an integer within a range of values	{x:range(1,100)}

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

```
[Route("{id:int:min(1)}")]
public Student Get(int id)
{
   return students.FirstOrDefault(s => s.Id == id);
}
```

- With the above change, if you specify a positive number like 1 in the URI, then it will be mapped to Get(int id) method as expected
  /api/students/1

  However, if you specify 0 or a negative number less than ZERO, you will get an error. For example if you specify 0 as the value for id in the URI, you will get
  No HTTP resource was found that matches the request URI
  'http://localhost:65116/api/students/0'

- Along with the "min" constraint you can also specify "max" constraint as shown below. For example if you want the id value in the URI to be between 1 and 3 inclusive, then you can specify both "min" and "max" constraints as shown below.

```
[Route("{id:int:min(1):max(3)}")]
public Student Get(int id)
{
   return students.FirstOrDefault(s => s.Id == id);
}
```

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

```
[Route("{id:int:range(1,3)}")]
public Student Get(int id)
{
   return students.FirstOrDefault(s => s.Id == id);
}
```

Generating links using route names in asp.net web api

, to generate links in ASP.NET Web API
1. Set a name for the route using the Name property of the [Route] attribute

```
[Route("{id:int}", Name = "GetStudentById")]
public Student Get(int id)
{
   return students.FirstOrDefault(s => s.Id == id);
}
```

2. Use the name of the route to generate the link

```
public HttpResponseMessage Post(Student student)
{
   students.Add(student);
   var response = Request.CreateResponse(HttpStatusCode.Created);
   response.Headers.Location = new
            Uri(Url.Link("GetStudentById", new { id = student.Id }));
   return response;
}
```

IHttpActionResult vs HttpResponseMessage

- In Web API 1, we have HttpResponseMessage type that a controller action method returns. A new type called "IHttpActionResult" is introduced in Web API 2 that can be returned from a controller action method.
- Instead of returning HttpResponseMessage from a controller action, we can now return IHttpActionResult.
- There are 2 main advantages of using the IHttpActionResult interface.
    1. The code is cleaner and easier to read
    2. Unit testing controller action methods is much simpler. We will discuss, how easy it is to unit test a method that returns IHttpActionResult instead of HttpResponseMessage .
- We have replaced both instances of HttpResponseMessage with IHttpActionResult. To return status code 200, we used Ok() helper method and to return status code 404, we used NotFound() method. To the Ok() method we have passed the type we want to return from the action method. Also notice, the code is now much cleaner and simpler to read.

```
public class StudentsController : ApiController
{
   static List<Student> students = new List<Student>()
   {
      new Student() { Id = 1, Name = "Tom" },
      new Student() { Id = 2, Name = "Sam" },
      new Student() { Id = 3, Name = "John" }
   };

   public IHttpActionResult Get()
   {
      return Ok(students);
   }

   public IHttpActionResult Get(int id)
   {
      var student = students.FirstOrDefault(s => s.Id == id);
      if (student == null)
      {
         //return NotFound();
         return Content(HttpStatusCode.NotFound, "Student not found");
      }

      return Ok(student);
   }
}
```

- **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

- **Why is versioning required in Web API?**
- Once a Web API service is made public, different client applications start using your Web API services.
- As the business grows and requirements change, we may have to change the services as well, but the changes to the services should be done in way that does not break any existing client applications.
- This is when Web API versioning helps. We keep the existing services as is, so we are not breaking the existing client applications, and develop a new version of the service that new client applications can start using.
- **Different options available to version Web API services :** Versioning can be implemented using
  1. URI's
  2. Query String
  3. Version Header
  4. Accept Header
  5. Media Type
- For more info
  https://csharp-video-tutorials.blogspot.com/2017/02/web-api-versioning-using-uri.html

https://csharp-video-tutorials.blogspot.com/2017/02/web-api-versioning-using-querystring.html
https://csharp-video-tutorials.blogspot.com/2017/03/web-api-versioning-using-custom-header.html

Web API versioning using accept header

- **What is Accept header**
  The Accept header tells the server in what file format the browser wants the data. These file formats are more commonly called as MIME-types. MIME stands for Multipurpose Internet Mail Extensions.

https://csharp-video-tutorials.blogspot.com/2017/03/web-api-versioning-using-accept-header.html

https://csharp-video-tutorials.blogspot.com/2017/03/web-api-versioning-using-custom-media.html