**HTTP:-** Hypertext Transfer Protocol using which we can access any web application. HTTP is a TCP/IP based communication protocol, that is used to deliver data (HTML files, image files, query results, etc.) on the World Wide Web.

HTTP is not just for serving up web pages. It is also a powerful platform for building APIs that expose services and data. Almost any platform that you can think of has an HTTP library, so HTTP services can reach a broad range of clients, including browsers, mobile devices, and traditional desktop applications.

**Asp.Net Web API:-**

Asp.Net Web API is a framework for building HTTP services that can be consumed by a broad range of clients including browsers, mobiles, iphone and tablets. It is very similar to ASP.NET MVC since it contains the MVC features such as routing, controllers, action results, filter, model binders, IOC container or dependency injection. It is a part of the core ASP.NET platform and can be used with MVC and other types of Web applications like Asp.Net WebForms. It can also be used as a stand-alone Web services application.

# RESTful Web Services

REST stands for REpresentational State Transfer. REST is an architectural style not a protocol.

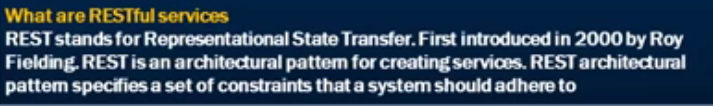
Advantages of RESTful Web Services

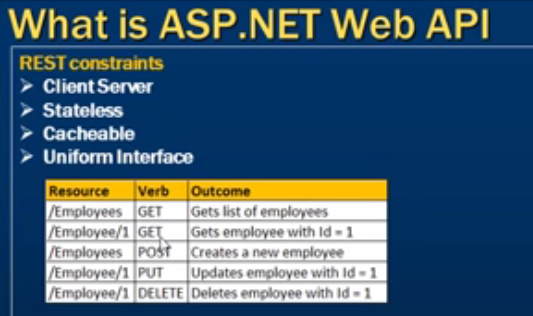
**Fast**: RESTful Web Services are fast because there is no strict specification like SOAP. It consumes less bandwidth and resource.

**Language and Platform independent**: RESTful web services can be written in any programming language and executed in any platform.

**Can use SOAP**: RESTful web services can use SOAP web services as the implementation.

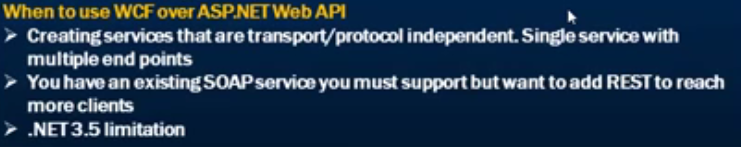
**Permits different data format**: RESTful web service permits different data format such as Plain Text, HTML, XML and JSON.





**What is Uniform Interface**

Uniform Interface is an important feature of REST. It has a group of methods; they are GET, PUT, POST, DELETE etc. These methods are easily understandable by the client and server. It is important that use the correct method for the specified action. If the request is based on the GET method and it chooses the DELETE method then the resource will be Deleted. That would be a problem for the other methods that do perform the appropriate action.



Why Asp.Net Web API (Web API)

Today, a web-based application is not enough to reach its customers. People are very smart, they are using iphone, mobile, tablets etc. devices in its daily life. These devices also have a lot of apps for making the life easy. Actually, we are moving from the web towards apps world.

**Asp.Net Web API VS Asp.Net MVC:-**

1. Asp.Net MVC is used to create web applications that returns both views and data but Asp.Net Web API is used to create HTTP services with easy and simple way that returns only data not view.

2. In Web API the request are mapped to the actions based on HTTP verbs but in MVC it is mapped to actions name.

3. Asp.Net Web API is new framework. The model binding, filters, routing and others MVC features exist in Web API are different from MVC and exists in the new System.Web. Http assembly. In MVC, these features exist within System.Web.Mvc.

4. Web API is light weight architecture and except the web application it can also be used with smart phone apps but using mvc except only web application.

5. The Web API helps the creation of RESTful services over the .Net Framework but the MVC does not support.

#### What are the Advantages of Using ASP.NET Web API?

Using ASP.NET Web API has a number of advantages, but core of the advantages are:

* It works the HTTP way using standard HTTP verbs like GET, POST, PUT, DELETE, etc. for all CRUD operations
* Complete support for routing
* Response generated in JSON or XML format using MediaTypeFormatter.
* It has the ability to be hosted in IIS as well as self-host outside of IIS
* Supports Model binding and Validation
* Support for OData.

#### WCF Vs ASP.NET Web API?

Actually, **Windows Communication Foundation** is designed to exchange standard SOAP-based messages using variety of transport protocols like HTTP, TCP, NamedPipes or MSMQ, etc.

On the other hand, **ASP.NET API** is a framework for building non-SOAP based services over HTTP only.

#### How to Return View from ASP.NET Web API Method?

(A tricky Interview question) No, we can't return view from ASP.NET Web API method. ASP.NET Web API creates HTTP services that renders raw data. Although, it's quite possible in ASP.NET MVC application.

#### Can we use Web API with ASP.NET Web Form?

Yes, ASP.NET Web API is bundled with ASP.NET MVC framework but still it can be used with ASP.NET Web Form.

It can be done in three simple steps as follows:

1. Create a Web API Controller
2. Add a routing table to Application\_Start method of Global.asax
3. Make a jQuery AJAX Call to Web API method and get data

**8) Which .NET framework supports Web API?**

NET 4.0 and above version supports web API.

**9) Web API uses which of the following open-source library for JSON serialization?**

Web API uses Json.NET library for JSON serialization.

**13) What is Web API Routing?**

Routing is pattern matching like in MVC.

All routes are registered in Route Tables.

For example:

Routes.MapHttpRoute(

Name: "ExampleWebAPIRoute",

routeTemplate: “api/{controller}/{id}

defaults: new { id = RouteParameter.Optional}

**20) What is the meaning of TestApi?**

Test Api is a utility library of APIs. Using this library tester developer can create testing tools and automated tests for a .NET application using data-structure and algorithms.

**7) How can you handle errors in Web API?**

Several classes are available in Web API to handle errors. They are HttpError, Exception Filters, HttpResponseException, and Registering Exception Filters.

**28) What New Features comes with ASP.NET Web API 2.0?**

The latest features of ASP.NET Web API framework v2.0 are as follows:

* Attribute Routing
* Cross-Origin Resource Sharing
* External Authentication
* Open Web Interface NET
* HttpActionResult
* Web API OData

**34) How to unit test Web API?**

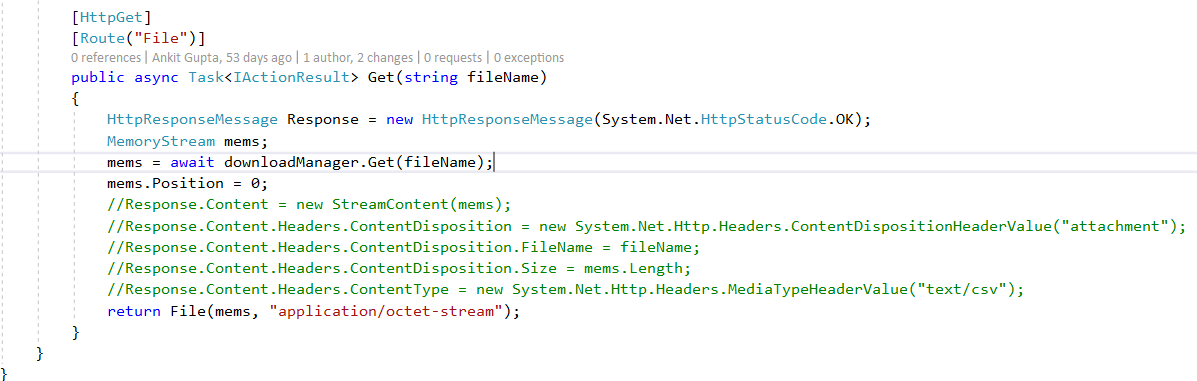
We can perform a Unit test using Web API tools like Fiddler.

Here, are some setting to be done if you are using

Fiddler –Compose Tab -> Enter Request Headers -> Enter the Request Body and execute.

**HTTP Headers in Web Api**

HTTP header is help us to format data or response.



# New Features in ASP.NET Web API 2

### Attribute Routing

Config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{Controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

[Route("books/{bookId}/authors")]

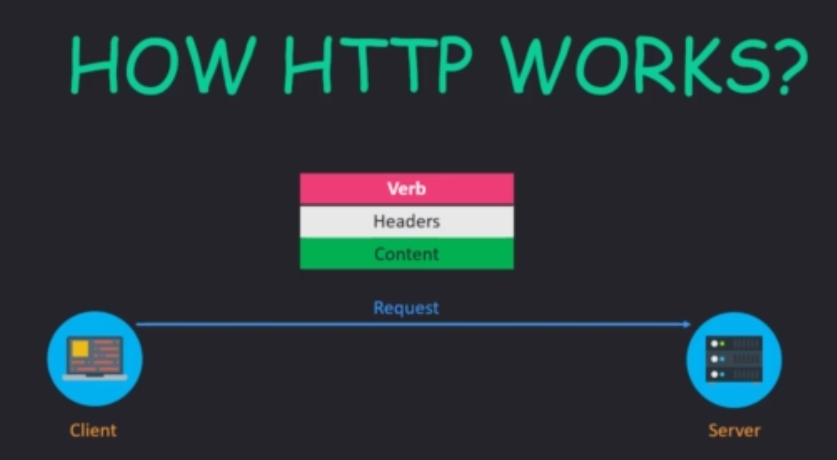
public IEnumerable<Author> GetAuthorByBook(int bookId) { ..... }

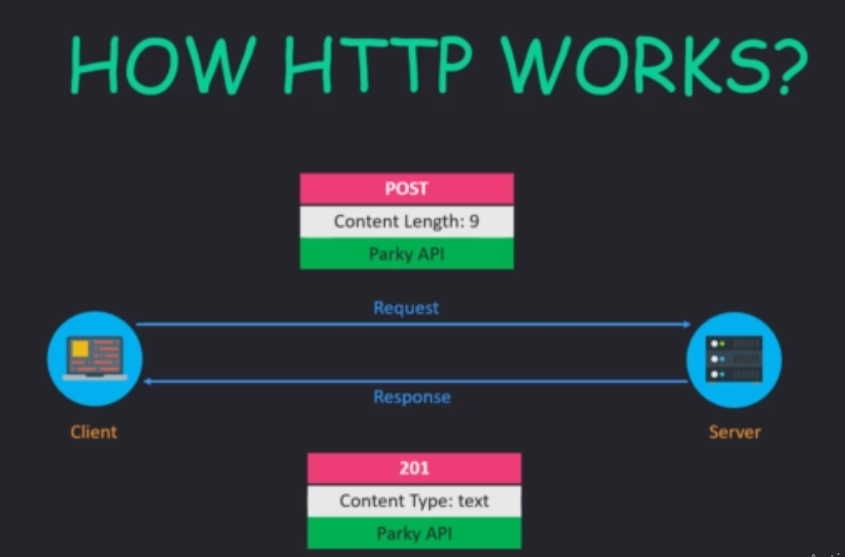
2-CORS(Cross Origin Resource Sharing)

3- IHTTPActionResult

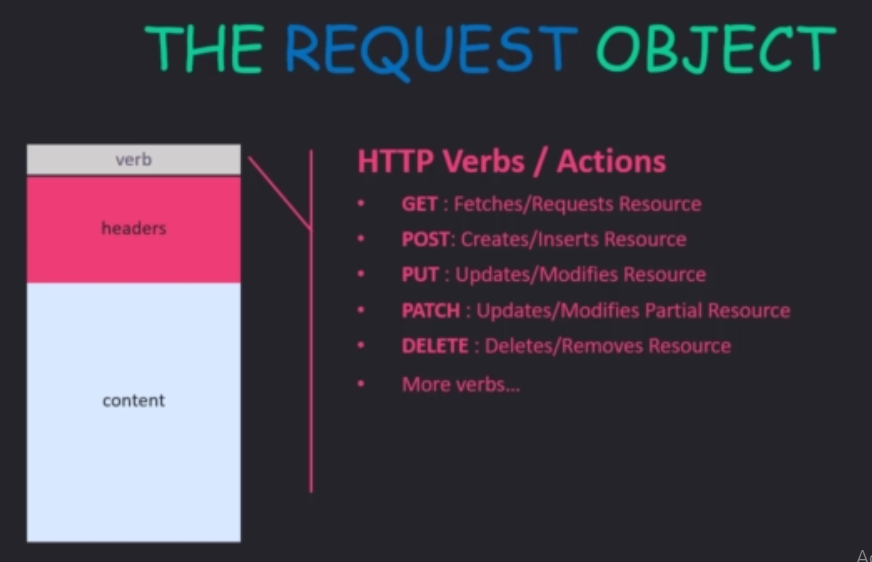
4-Controller Class now change to same Controller for both MVC and web API.

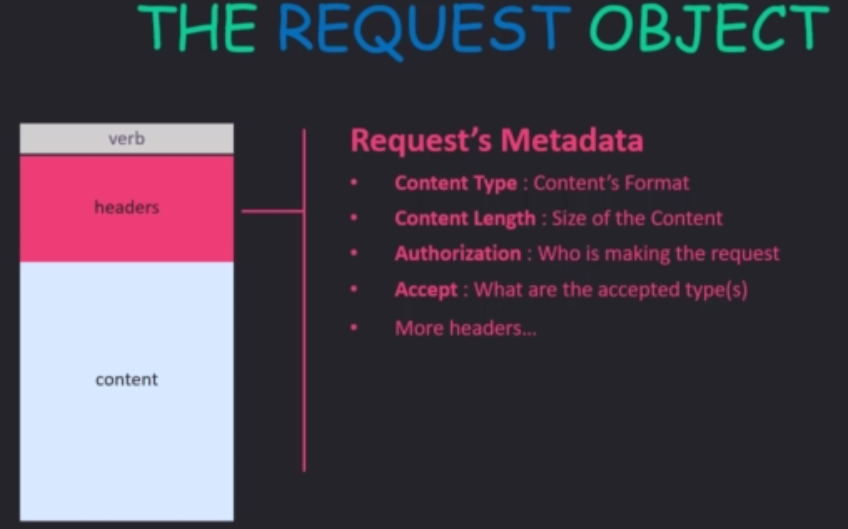
**How HTTP will work**

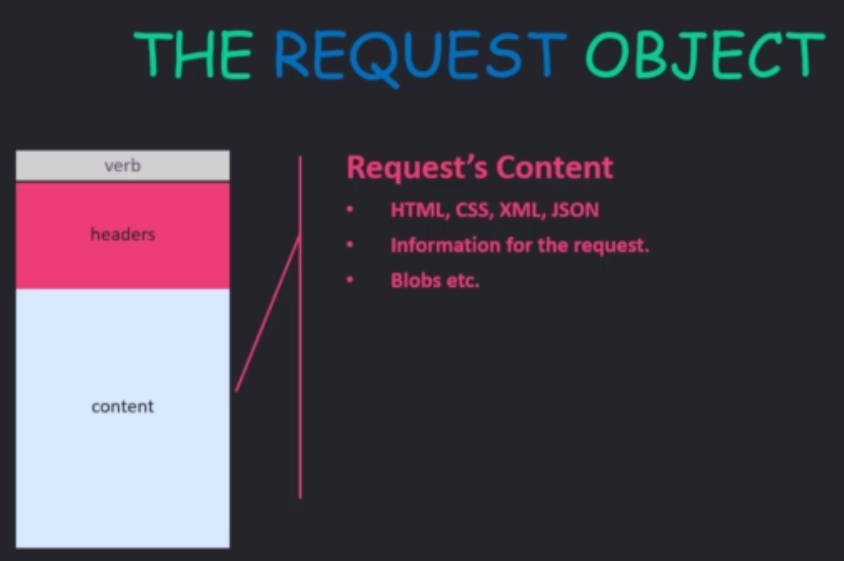




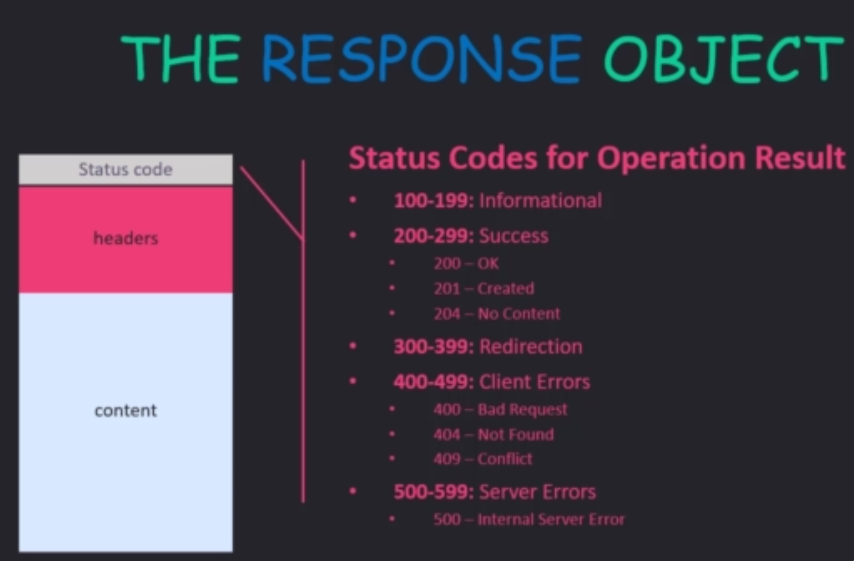
**The Request Object**

****

****

****

**HTTP Status Code**

****

200-Ok (Everything is fine)

201-Created (For creating the resourse in server like for Post request)

204-No Content (Common when update anything)

400-Bad Request (Try to pass something which is not expected by the API)

404-Not Found (Try to access the resource which is not available)

**403-** The HTTP **403** Forbidden client **error status response code** indicates that the server understood the request but refuses to authorize it

# ASP.NET Core 2.2 – Data Transfer Objects (DTOs) and AutoMapper

It is not a recommeded practice to return **entities** from our Web API. When we speak of entities, we refer to those classes that serve to model a table in our database. One of the drawbacks of this is that sometimes we do not want to show all the data contained in these entities. It is normal to only want to show a small data set, and not everything. One way to solve this is by using data transfer objects.

A data transfer object (in English: data transfer object, DTO) is an object used to transport data between processes. We will use these DTOs to represent the data we want the clients of our Web API to receive. Another name that the DTOs receive is View Model.

We are going to create our first DTO. We have our Author entity with the following fields:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Author {      public int Id { get; set; }      [Required]      public string Name { get; set; }      public string Identification {get; set;}      public DateTime Birthdate {get; set;}      public List<Book> Books {get; set;}  } |

We see that we have a field called Identification. This is a sensitive information of the author which we do not want to send to all API clients. One way to ensure that we do not send such information is by creating an author’s DTO which only contains the fields we want our API clients to see. Let’s create a class called AuthorDTO in the Models folder:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class AuthorDTO {     public int Id { get; set; }     [Required]     public string Name { get; set; }     public DateTime Birthdate {get; set;}     public List<BookDTO> Books {get; set;}  } |

Where BookDTO is:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class BookDTO {      public int Id { get; set; }      [Required]      public string Title { get; set; }      public int AuthorId {get; set;}  } |

With this, we make sure that the confidential information of the author is not distributed by accident. Now, we must go to our AuthorsController and change the type of return data from Author to AuthorDTO. Let’s start with the method that returns an individual Author:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | [HttpGet("{id}")]  public ActionResult<AuthorDTO> Get(int id)  {    var author = \_context.Authors.Include(x => x.Books).FirstOrDefault(x => x.Id == id);    if (author == null)      {        return NotFound();      }      return author;  } |

This causes us to have an error. The error is that we are currently trying to return an author when what we would like to return is an Author. One way to solve this is to create an AuthorDTO instance, fill in your fields, and return this. Although this is something we can do, in the long run it becomes tedious to perform these manual mappings between types. One solution to this problem is to use a library like AutoMapper, which is responsible for doing this property mapping for us.

### AutoMapper

We are going to install AutoMapper in our project. In Visual Studio, in the Package Manager Console you can use the following command:

*Install-Package AutoMapper.Extensions.Microsoft.DependencyInjection*

In the dotnet CLI:

*dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection*

Now we can configure AutoMapper, for that we are going to go to the ConfigureServices method of the Startup class and place the following:

|  |  |
| --- | --- |
| 1 | services.AddAutoMapper(); |

This is the basic configuration of AutoMapper. Now, we can go back to our Get action and place the following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | private readonly IMapper \_mapper;    public AuthorsController(ApplicationDbContext context,          IMapper mapper)  {      this.\_context = context;      this.\_mapper = mapper;  }    [HttpGet("{id}", Name = "GetAuthor")]  public ActionResult<AuthorDTO> Get(int id)  {      // …      var authorDTO = \_mapper.Map<AuthorDTO>(author);      return authorDTO;  } |

As you can see, with this simple line of code we have managed to map the properties of Author to AuthorDTO. This mapping also performs the mapping from Book to BookDTO. Notice that I also had to inject the IMapper service to our controller.

AutoMapper also works with element collections. For example, if we have a collection of authors, we can map it to a collection of AuthorDTO without any problem:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | [HttpGet]  public ActionResult<IEnumerable<AuthorDTO>> Get()  {      var authors = \_context.Authors.ToList();      var authorsDTO = \_mapper.Map<List<AuthorDTO>>(authors);      return authorsDTO;  } |

**What is Swashbuckle?**

**Swashbuckle.AspNetCore** is an open source project which will use for generating Swagger documents for ASP.NET Core Web APIs.

So for configure the swagger first we need to install the **Swashbuckle.AspNetCore**  package form nugut.

Then need to do the below configuration in Startup.cs file.

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<ApplicationDBContext>

(options => options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

services.AddScoped<INationalParkRepository, NationalParkRepository>();

services.AddAutoMapper(typeof(ParkyMappings));

services.AddSwaggerGen(options=> {

options.SwaggerDoc("ParkyOpenAPISpec", new Microsoft.OpenApi.Models.OpenApiInfo()

{

Title = "Parky API",

Version = "1"

});

});

services.AddControllers();

}

**-**Need to define the document name like "ParkyOpenAPISpec" and Open API Information like below

{

Title = "Parky API",

Version = "1"

});

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseHttpsRedirection();

app.UseSwagger();

app.UseSwaggerUI(options =>

{

options.SwaggerEndpoint("/swagger/ParkyOpenAPISpec/swagger.json", "Parky API");

options.RoutePrefix = "";

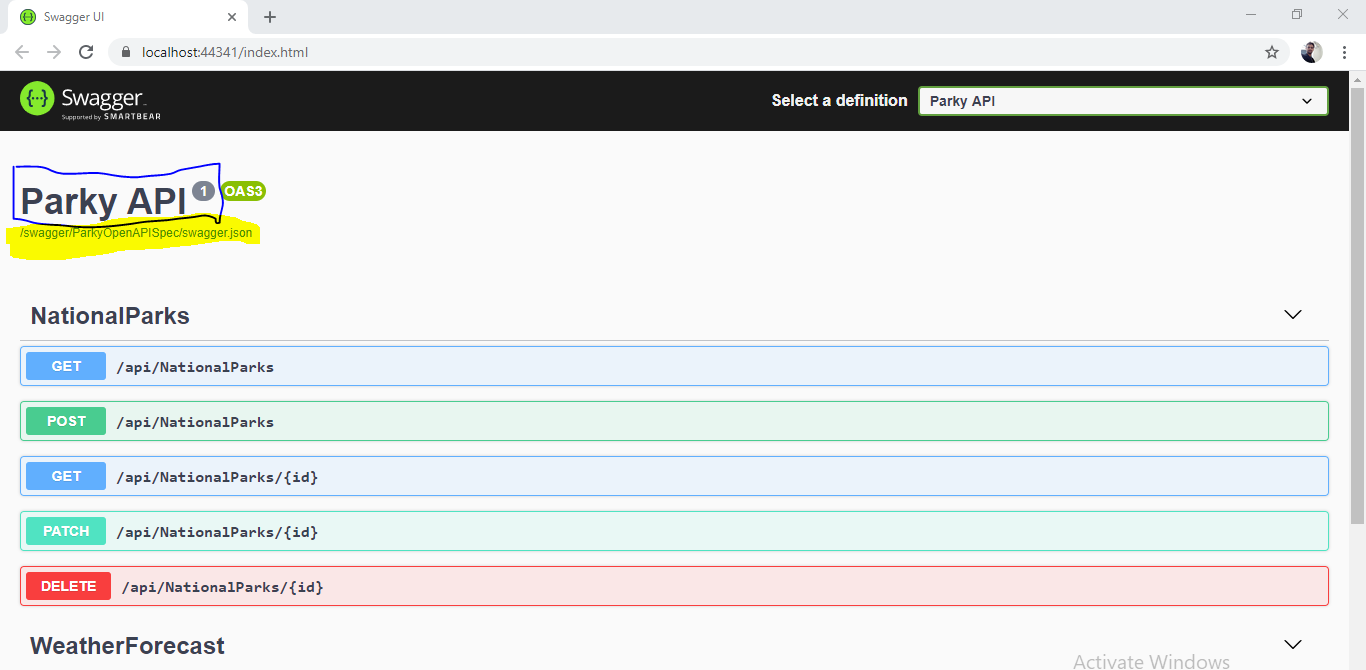
} ) ;

}

**-**Then need to define the endpoint.

- options.RoutePrefix = "";---- Will use to set the Swegger UI and default page.

- Once all the configuration will done below page will open when run the Project.



-In Blue color highlighted show the title and Version Info & yellow color show the swagger End Point.

**REST headers**

Every REST request must contain three HTTP header fields: Accept, Content-Type, and Cookie.

**Understanding the Content-Type header**

The *Content-Type* header describes the format the body of your request is being sent as. For example, the body of your requests can be sent as JSON or XML, but you need to declare in the Content-Type header which one is being used. This header is required in all requests.

A *request* is any query made by your application to . expects one of these values as the [Content-Type](http://www.w3.org/Protocols/rfc1341/4_Content-Type.html) header:

**♦** To send JSON in a request, use *application/json*.

**♦** To send XML in a request, use *application/xml*.

**Understanding the Accept header**

The *Accept* header describes which format you want a response body to arrive as. For example, responses can be delivered either as XML or JSON by modifying the Accept header. This header is required in all requests.

A *response* is any reply from to your application. expects one of these values as the [Accept](http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html) header:

**♦** To receive JSON in a response body, use *application/json*.

**♦** To receive XML in a request body, use *application/xml*.

**How to get response XML format If request in json format.**

To get the response in xml format when request body sent the request as json below step we need to follow.

-First we need to add the below service in ConfigureService() in StartUp.cs class.

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

services.AddControllers().AddXmlDataContractSerializerFormatters();

}

-Then decorate the Api method with the below.

[HttpGet("api/WeatherForecast.{format}"), FormatFilter]

public IEnumerable<WeatherForecast> Get()

{

var rng = new Random();

return Enumerable.Range(1, 5).Select(index => new WeatherForecast

{

Date = DateTime.Now.AddDays(index),

TemperatureC = rng.Next(-20, 55),

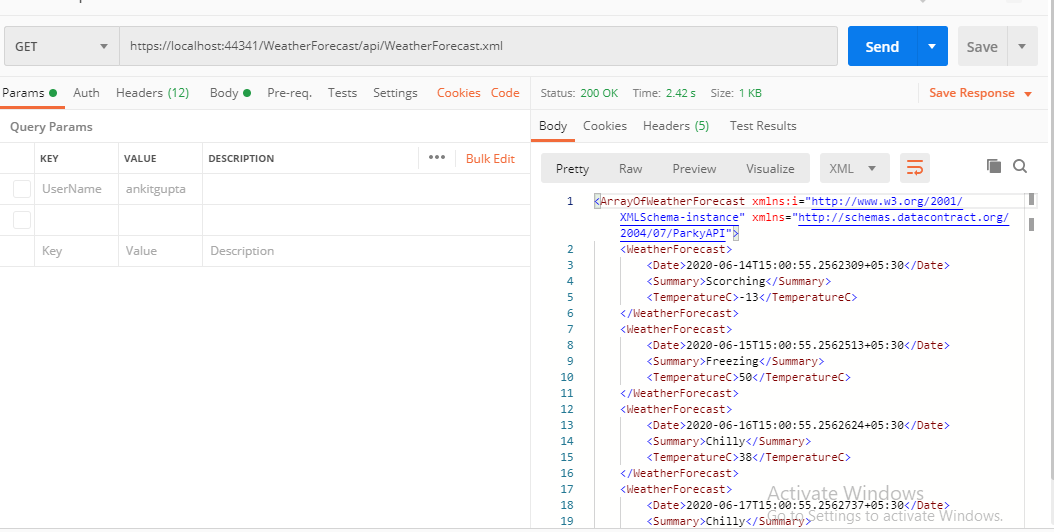
Summary = Summaries[rng.Next(Summaries.Length)]

})

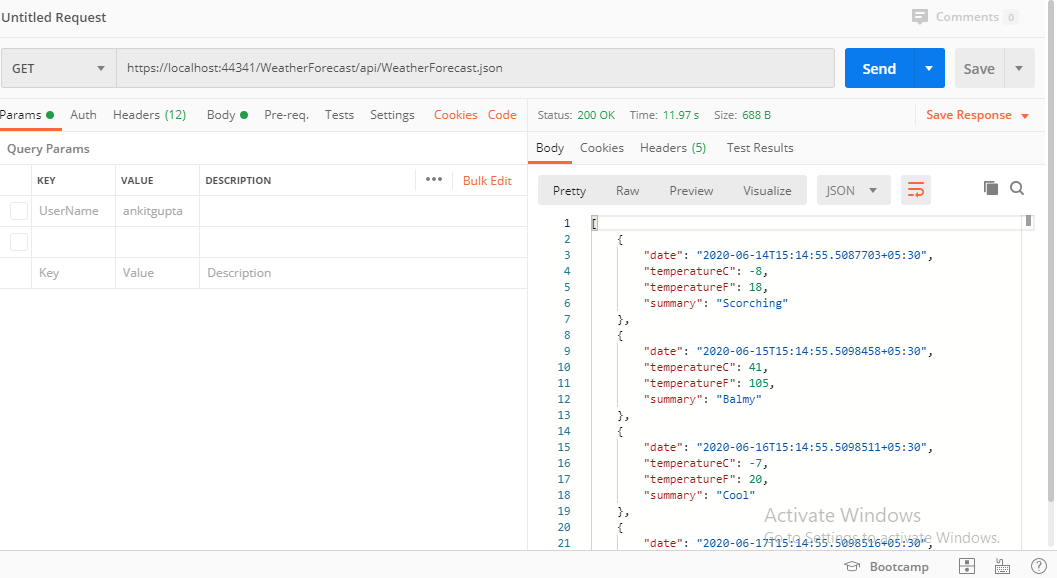
.ToArray();

}

-For checking the response in XML format we need to pass the request in below format using postman of url.



-For checking the response in json format we need to pass the request in below format using postman of url.



**What is JWT?**

JWT is an acronym for JSON Web Token. JWT is a way for securely transmitting information between parties as a JSON object. This information is verified and trusted because it’s digitally signed. JWT’s can be using a secret (with the HMAC algorithm) or public/private key pair using RSA/ECDSA.

JWT structure contains three parts,

Header

Payload

Signature

**Header:**

The header consists of two-part one is signing algorithm (HMAC, SHA256, etc.) and the other one is the type of the token which is JWT.

Example:{“alg”: “HMAC”,“typ”: “JWT”}

This JSON form is Base64Url encoded to form the first part.

**Payload:**

The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data. Again the payload formed JSON is Base64Url encoded to form second part.

Ex- Claims can contains like:- UserId,EmailAddress,UserName which will help to authenticate the user.

**Signature:**

For the signature created you need to use encoded header, encoded payload, and secret with the algorithm specified at the header.

Example:HMAC(base64UrlEncode(header) + “.” +base64UrlEncode(payload),secret)

Putting all of these three parts together form Json Web Token.

**Return Types in Web API**

**Return Void**

**Return specific type**

This is the most simplistic and straightforward way to return values from an API. Consider the following API action:

[HttpGet]

public List<Customer> Get()

{

return db.Customers.ToList();

}

In this approach the Get() action returns a known type - List of Customer objects in this case. This approach is good if you simply want to return data to the client without accounting for unexpected conditions such as exceptions and HTTP codes such as 404 and 200.

**Return IActionResult**

When your return value is a mix of data and HTTP codes you can't use the previous approach. That's because the return type is a fix well-known type. If you want to return NotFoundResult or OkResult or ObjectResult you can't use the preceding approach. In such case you can return the values as IActionResult. Consider the following example:

[HttpGet("{id}")]

public IActionResult Get(string id)

{

Customer cust = db.Customers.Find(id);

if (cust == null)

{

return NotFound();

}

return Ok(cust);

}

In this case we want to return a particular customer based on the CustomerID passed in the action. It is possible that the CustomerID is invalid and Find() may fail to return a Customer object. If Find() return null we want to return 404 - Not Found response to the client. On the other hand if Customer is found we want to return 200 - Ok to the client along with the found Customer object.

This can be accomplished once we set the return type of IActionResult. The IActionResult interface is implemented by classes such as NotFoundResult and OkResult. The NotFound() and Ok() methods return these respective objects to the client.

**Return ActionResult<T>**

ActionResult<T> allows you to combine both of the approaches discussed earlier. You can return a type derived from ActionResult or a specific type. Consider the following Get() action:

[HttpGet("{id}")]

public ActionResult<Customer> Get(string id)

{

Customer cust = db.Customers.Find(id);

if (cust == null)

{

return NotFound();

}

return cust;

}

As you can see you don't need to wrap cust object in Ok() or ObjectResult. You can either return NotFoundResult or ActionResult<Customer>

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.

Example: Void Return Type

public class StudentController : ApiController

{

public void Delete(int id)

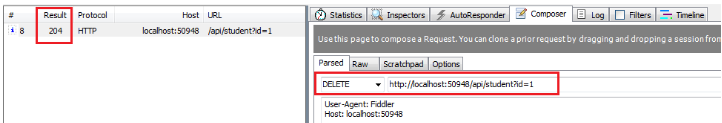
{

DeleteStudentFromDB(id);

}

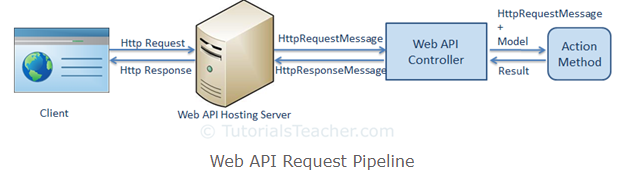
}

As you can see above Delete action method returns void. It will send 204 "No Content" status code as a response when you send HTTP DELETE request as shown below.



HttpResponseMessage

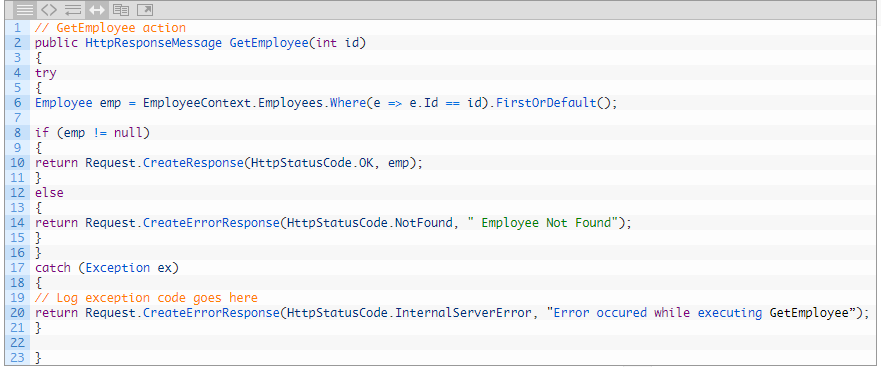
Web API controller always returns an object of HttpResponseMessage to the hosting infrastructure. The following figure illustrates the overall Web API request/response pipeline.



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.

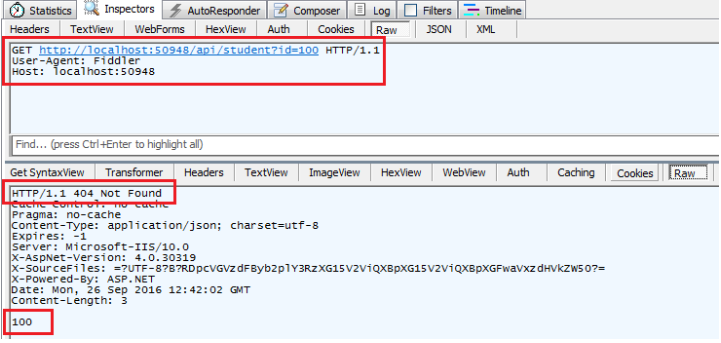
The advantage of sending HttpResponseMessage from an action method is that you can configure a response your way. You can set the status code, content or error message (if any) as per your requirement.

Example: Return HttpResponseMessage

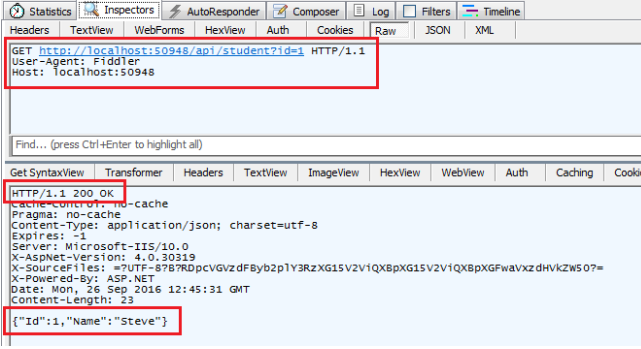


In the above action method, if there is no student with specified id in the DB then it will return HTTP 404 Not Found status code, otherwise it will return 200 OK status with student data.

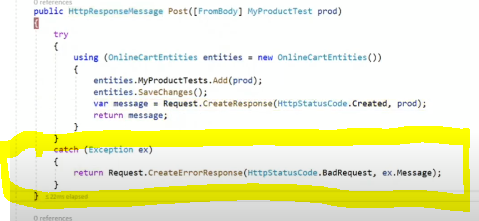
For example, an http GET request http://localhost:xxxx/api/student?id=100 will get following response considering student with id=100 does not exists in the DB.



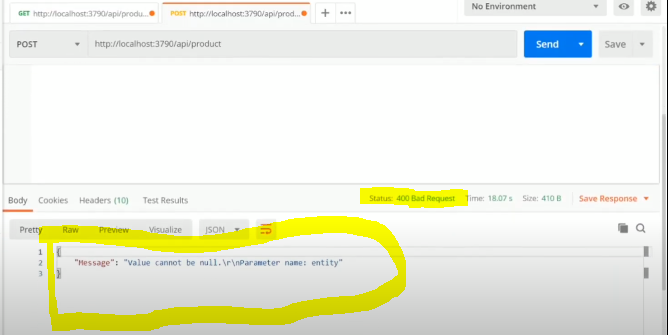
The same way, an HTTP GET request http://localhost:60464/api/student?id=1 will get following response considering student with id=1 exists in the database .



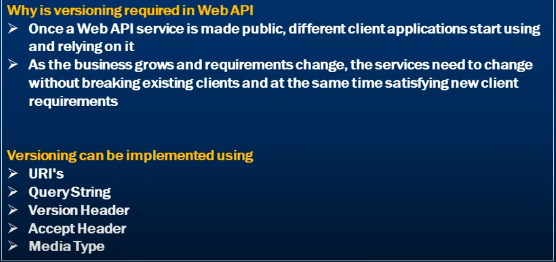
**How to get the status code when Exception Occur**

****

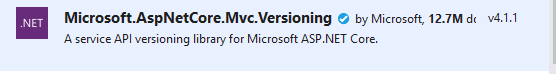
**-**As per the above screen shot when ever exception will occur it will create the error response using the CreateErrorResponse() method and return the response like below.



**Versioning in Web API**



To do versioning in ASP.NET Core Web API, first, we have to install the below the Nuget package which will provide necessary methods for versioning. Right-click on the solution and click on Manage Nuget Package and search for package "Microsoft.AspNetCore.Mvc.Versioning" and install it.



Once the NuGet package is successfully installed, then the next step is to open Startup.cs and add the below code in the ConfigureServices method.

1. **public** **void** ConfigureServices(IServiceCollection services)
2. {
3. services.AddControllers();
4. services.AddApiVersioning(x =>
5. {
6. x.DefaultApiVersion = **new** ApiVersion(1, 0);
7. x.AssumeDefaultVersionWhenUnspecified = **true**;
8. x.ReportApiVersions = **true**;
9. });
10. }

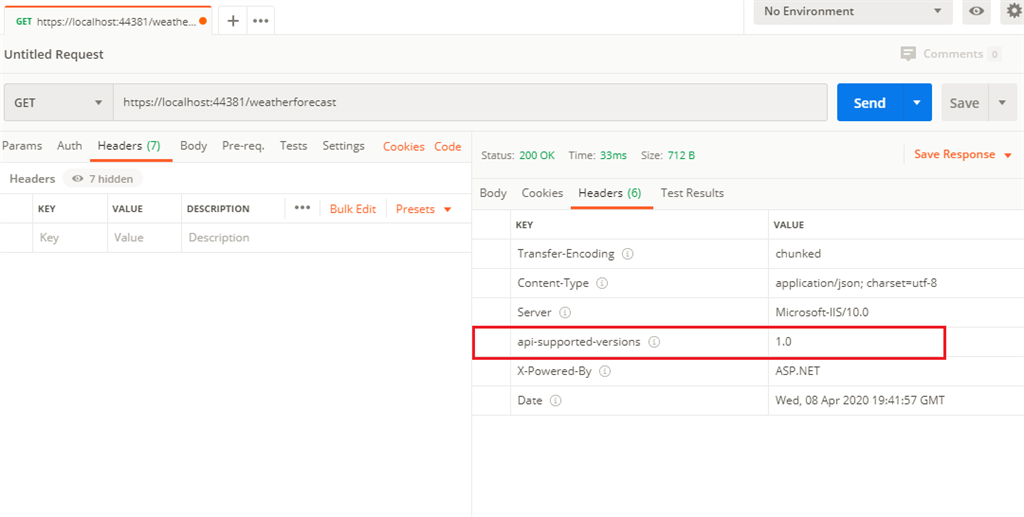
Let's understand the above code that we added.

Line #6 - DefaultApiVersion is used to set the default version to API

Line #7 - This flag AssumeDefaultVersionWhenUnspecified flag is used to set the default version when the client has not specified any versions. If we haven't set this flag to true and client hit the API without mentioning the version then UnsupportedApiVersion exception occurs.

Line #8 - To return the API version in response header.

So if we hit the same API, we can see the default version in the response header.



Before discussing about different types, let's create two controllers (i.e EmployeeV1Controller and EmployeeV2Controller) as shown below.

**EmployeeV1Controller.cs**

1. **using** Microsoft.AspNetCore.Mvc;
3. **namespace** api\_versioning\_demo.Controllers
4. {
5. [Route("api/employee")]
6. [ApiController]
7. **public** **class** EmployeeV1Controller : ControllerBase
8. {
9. [HttpGet]
10. **public** IActionResult Get()
11. {
12. **return** **new** OkObjectResult("employees from v1 controller");
13. }
14. }
15. }

**EmployeeV2Controller.cs**

1. **using** Microsoft.AspNetCore.Mvc;
3. **namespace** api\_versioning\_demo.Controllers
4. {
5. [Route("api/employee")]
6. [ApiController]
7. **public** **class** EmployeeV2Controller : ControllerBase
8. {
9. [HttpGet]
10. **public** IActionResult Get()
11. {
12. **return** **new** OkObjectResult("employees from v2 controller");
13. }
14. }
15. }

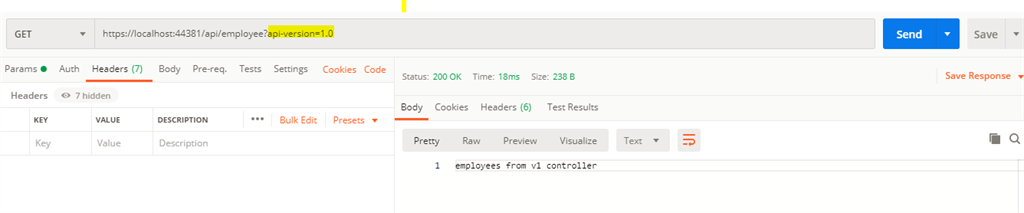
Query String-Based Versioning

In the case of query string based versioning, we have to specify the API version in the query string to call the specific controller. Now we mention the specific version to each controller using [ApiVersion(version\_name)] attributes. So our controller looks like the code below:

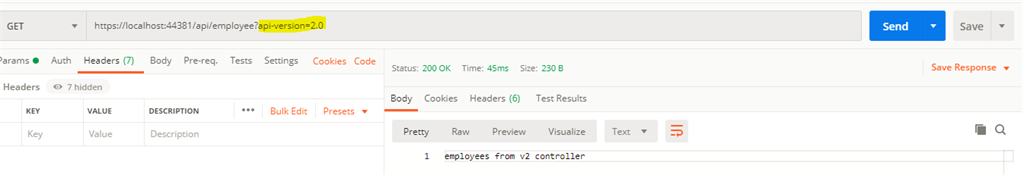
1. **using** Microsoft.AspNetCore.Mvc;
3. **namespace** api\_versioning\_demo.Controllers
4. {
5. [ApiController]
6. [ApiVersion("1.0")]
7. [Route("api/employee")]
8. **public** **class** EmployeeV1Controller : ControllerBase
9. {
10. [HttpGet]
11. **public** IActionResult Get()
12. {
13. **return** **new** OkObjectResult("employees from v1 controller");
14. }
15. }
16. }
17. **using** Microsoft.AspNetCore.Mvc;
19. **namespace** api\_versioning\_demo.Controllers
20. {
21. [ApiController]
22. [ApiVersion("2.0")]
23. [Route("api/employee")]
24. **public** **class** EmployeeV2Controller : ControllerBase
25. {
26. [HttpGet]
27. **public** IActionResult Get()
28. {
29. **return** **new** OkObjectResult("employees from v2 controller");
30. }
31. }
32. }

We can call controller using the below routes with the query string method.

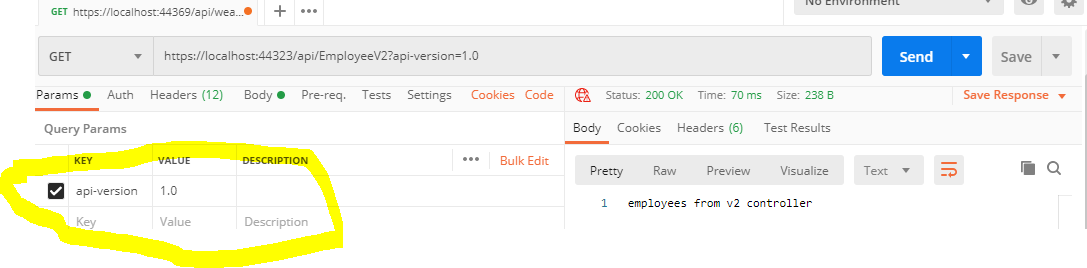
To call EmployeeV1Controller, we have to hit as https://localhost:44381/api/employee?api-version=1.0



For EmployeeV2Controller we have to type: https://localhost:44381/api/employee?api-version=2.0



Query string will show in the parameter section in the postman which is not showing in screen shot like below.



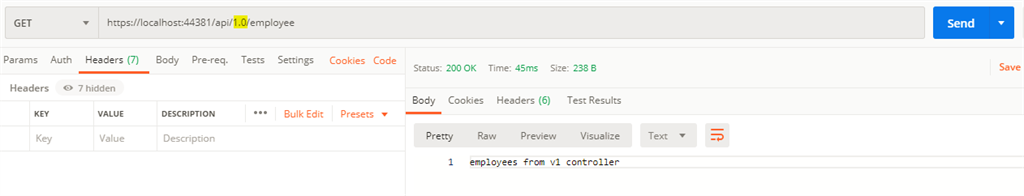
URL based Versioning

In this type of versioning, we can define versions in a URL so that it is more readable. Most users prefer this type over other types. You can to URL based versioning by changing the routs as [Route("api/{v:apiVersion}/Values")].

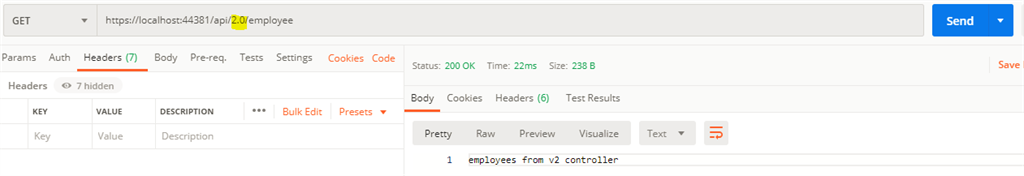
Let's modify the Route attribute in our both controller, as follows.

1. **using** Microsoft.AspNetCore.Mvc;
3. **namespace** api\_versioning\_demo.Controllers
4. {
5. [ApiController]
6. [ApiVersion("1.0")]
7. [Route("api/{v:apiVersion}/employee")]
8. **public** **class** EmployeeV1Controller : ControllerBase
9. {
10. [HttpGet]
11. **public** IActionResult Get()
12. {
13. **return** **new** OkObjectResult("employees from v1 controller");
14. }
15. }
16. }
17. **using** Microsoft.AspNetCore.Mvc;
19. **namespace** api\_versioning\_demo.Controllers
20. {
21. [ApiController]
22. [ApiVersion("2.0")]
23. [Route("api/{v:apiVersion}/employee")]
24. **public** **class** EmployeeV2Controller : ControllerBase
25. {
26. [HttpGet]
27. **public** IActionResult Get()
28. {
29. **return** **new** OkObjectResult("employees from v2 controller");
30. }
31. }
32. }

To call EmployeeV1Controller, we have to hit as https://localhost:44381/api/1.0/employee



To call EmployeeV2Controller we need: https://localhost:44381/api/2.0/employee

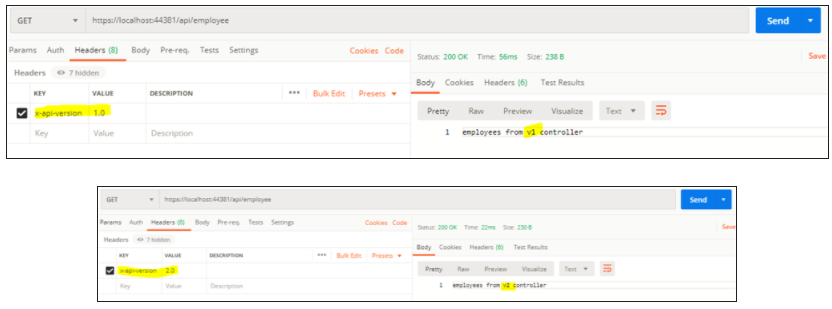


HTTP Header-Based Versioning

In this type, we have to send the version in the Http header when we call the controller. So open the *Startup.cs*and add the below line of code into *services.AddApiVersioning*method. Also, delete the version mentioned in routes.

1. **public** **void** ConfigureServices(IServiceCollection services)
2. {
3. services.AddControllers();
4. services.AddApiVersioning(x =>
5. {
6. x.DefaultApiVersion = **new** ApiVersion(1, 0);
7. x.AssumeDefaultVersionWhenUnspecified = **true**;
8. x.ReportApiVersions = **true**;
9. x.ApiVersionReader = **new** HeaderApiVersionReader("x-api-version");
10. });
11. }

Now enable the Http herder versioning type. When a client consumes the API, they have so send *x-api-version* into the header with specific version value to call the correct controller.



#### Using Media Type (Accept header) Versioning

Another approach of version the API is using the content negotiation process provided by HTTP. When client requests a resource using the Accept header, they could explicitly include the version number in the media type itself.

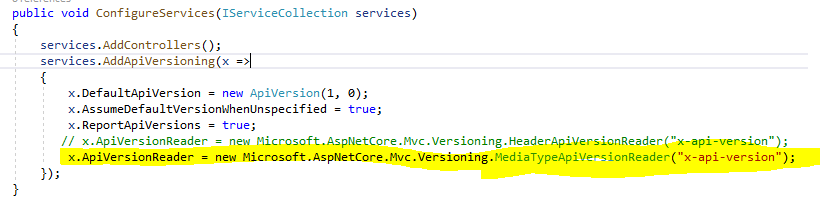
In GET request to the values endpoint shown below, the client is explicitly mentioning that it accepts the response of media type application/json with a version number of 1.1 from the server.

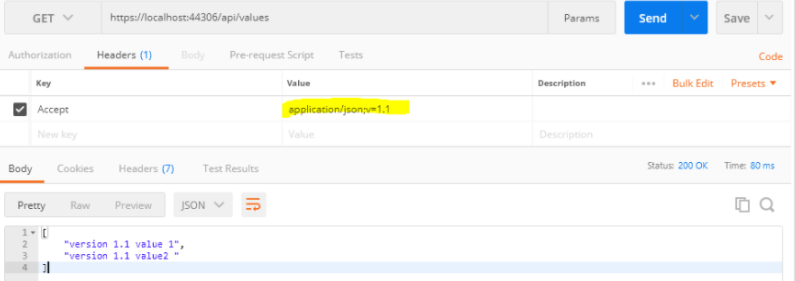


Server could read the Accept header from the request, and respond with the appropriate API version.

One of the disadvantages of using Media Type versioning scheme is that it can be quite obscure, difficult to implement and not immediately obvious to clients that they can request different API versions using the Accept header.

To implement versioning using media type set the ApiVersionReader to a instance of MediaTypeApiVersionReader class.

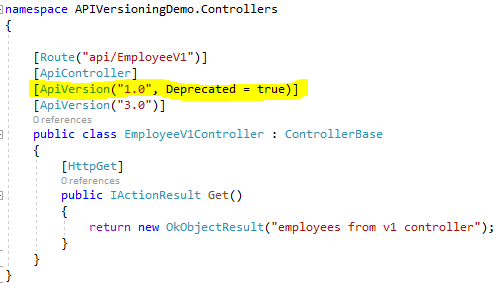


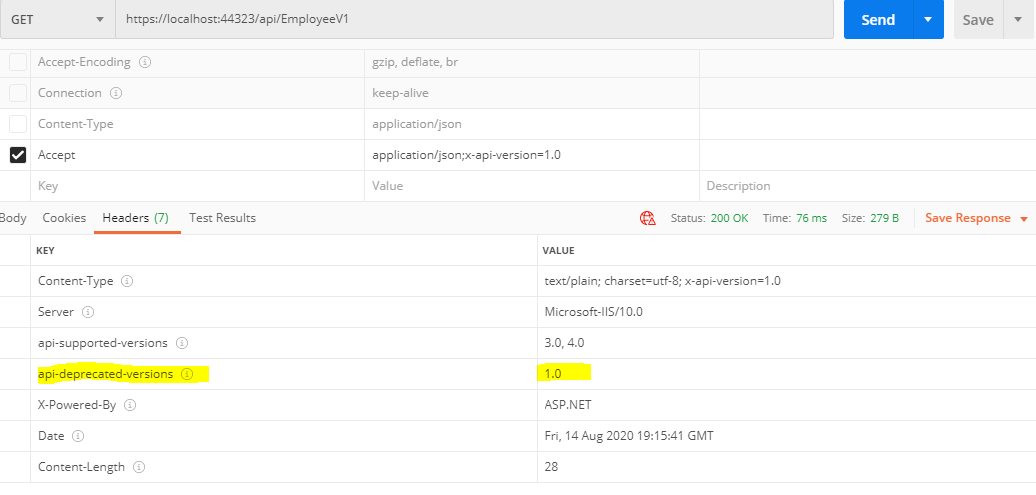


Advertising the Deprecated Versions

Similar to advertising the supported API versions for an endpoint, API versions which will be deprecated in the near future can also be advertised by setting the deprecated property to true in the **ApiVersion** attribute. The client could read the **api-deprecated-versions** in the response header and identify the deprecated API versions.

A deprecated API is one that you are no longer recommended to use, due to changes in the API. While deprecated classes, methods, and fields are still implemented, they may be removed in future implementations, so you should not use them in new code, and if possible rewrite old code not to use them.

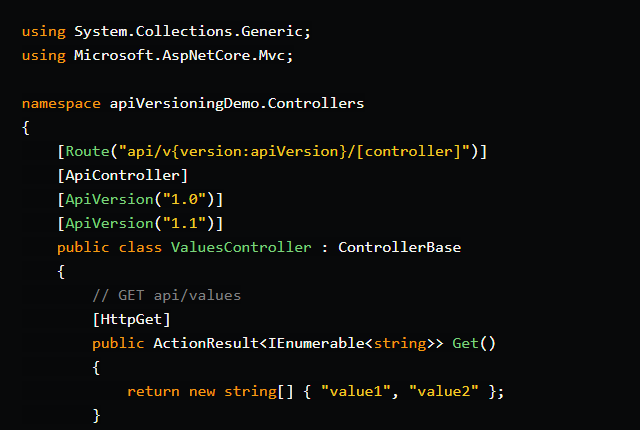


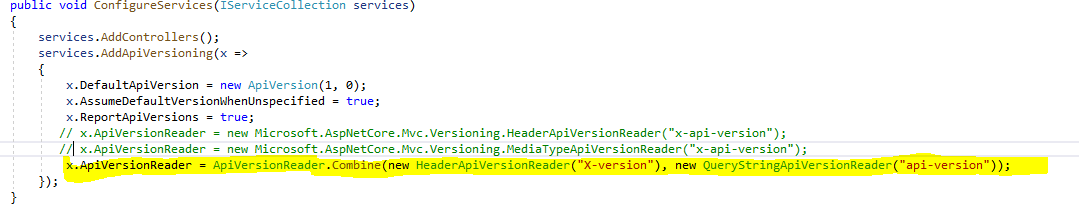


Deprecated is just for the advertisement it will show the response if we passed the deprecated version also.

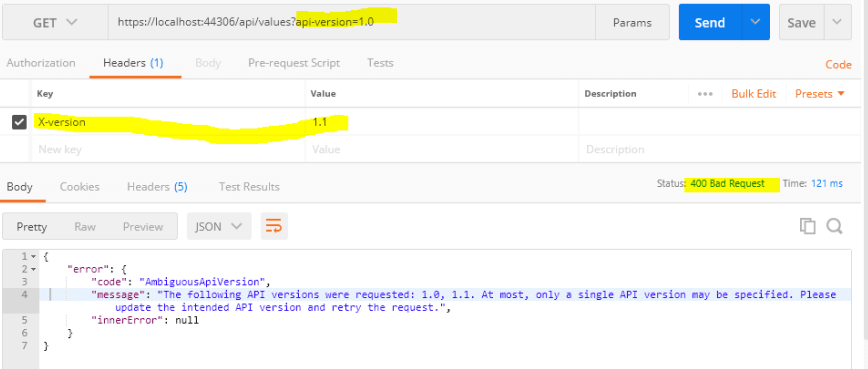
Supporting multiple versioning Schemes

Supporting for multiple API versioning schemes provide flexibility and give options to the clients to use the versioning scheme of their choice. The code snippet below demonstrate how to provide support for both the query params versioning scheme and request header versioning scheme using the static method **Combine** avaliable in the **ApiVersionReader** class.





Now that two different API versioning schemes are supported, the clients should make sure to use only one of the supported schemes. Technically clients could use both the schemes to specify the API version in a request as long as they specify the same version number. If two different versions are requested in the same request using the different API versioning scheme, it will result in a bad request error.

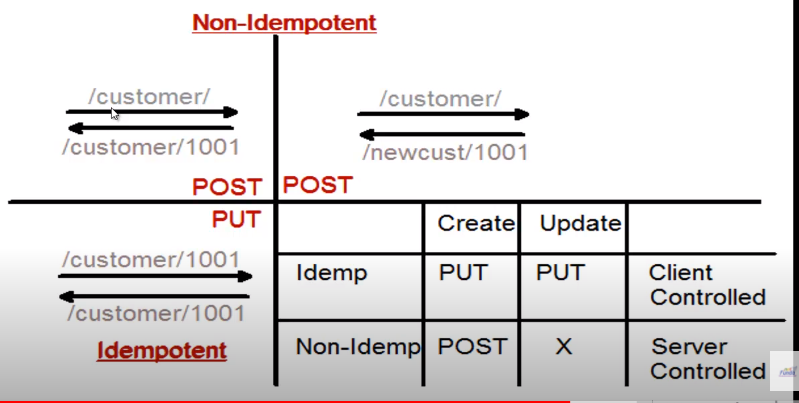


In the above example, API v1.0 is requested using the query param scheme and v1.1 is requested using the header scheme, which results in the Ambiguous API Version error.

**Difference between POST and Put Verbs**

-Put and Post both we can use for creating the new resource or update the resource. We just need to use the better option when developing the API. Mostly we preferred to use Post only.

- The only difference is that Post API is idempotent and Put is non-idempotent. Idempotent means the URL is different in request and response and non-idempotent means url is same for request and response. Like below screen shot.

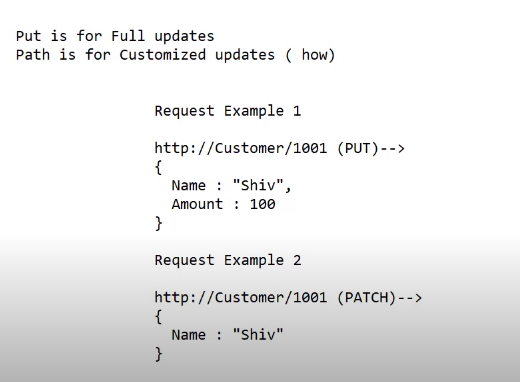


PUT method is [idempotent](https://restfulapi.net/idempotent-rest-apis/). So if you send retry a request multiple times, that should be equivalent to single request modification because URL is same.

POST is NOT idempotent. So if you retry the request N times, you will end up having N resources with N different URIs created on server because URL is different.

**Difference between Patch and Put**

Both will use to perform the update operation only deference is that put will use for full update and patch will use to update partially means some of the column.



**HTTP DELETE**

DELETE operations are **idempotent**. 204 (No Content) if the action has been performed but the response does not include an entity..

**Content Negotiation In Web API**

Content negotiation means to design the api in which we can get the response particular format based on the our requirement like we can send the request in json format but the response should be in xml format that is called the content negotiation which we can achieve using the content type , accept header.

To implement the same we need to do some configuration in startup.cs class.