ASP.NET WEB API

- The ASP.NET Web API is a great framework provided by Microsoft for building HTTP services that can be consumed by a broad range of clients including browsers, mobiles, iPhones, and tablets, etc. Moreover, ASP.NET Web API is an open-source and ideal platform for building REST-full services over the .NET Framework.

- These Web API services can be consumed by a variety of clients such as
  1. Browsers
  2. Mobile applications
  3. Desktop applications
  4. IOTs, etc.

• REST (Representational State Transfer):

- An architectural pattern used for exchanging data over a distributed environment.
- The REST architectural pattern treats each service as a resource and a client can access those resources by using HTTP Protocol methods such as GET, POST, PUT, PATCH, and DELETE.

• HTTP Verbs/Methods:

1. **GET**: The GET HTTP Method is used to retrieve the Data.

2. **POST**: Whenever you want to create a new resource in your application, then you need to use the POST method.

3. **PUT**: The PUT method is used to update all the properties of the current resource.

4. **PATCH**: The PATCH method is similar to the PUT method, but it is used to update few properties of the current resource.

5. **DELETE**: The DELETE method is used to delete the resource from the database.

- In modern applications, we use two concepts for delete. One is Soft Delete and another one is Hard Delete.
  1. **Soft Delete:** In your table, if you have some column like IsDeleted or IsActive, or something similar to this and you just want to update that column, then you cannot use SOFT DELETE Method. In that case, you need to use the PATCH method. This is because you are not deleting the record from the database, you just update the record.

  2. **Hard Delete:** If you want to remove the existing entity from the table, then you need to use the HARD DELETE method. For example, delete an existing product from the Product table in the database, etc.

- **What is Difference between REST & SOAP?**

- SOAP stands for Simple Object Access Protocol whereas REST stands for Representational State Transfer.
- The SOAP is an XML-based protocol whereas REST is not a protocol rather it is an architectural pattern i.e., resource-based architecture.
- SOAP has specifications for both stateless and state-full implementation whereas REST is completely stateless.
- SOAP enforces message format as XML whereas REST does not enforce message format as XML or JSON.
- The SOAP message consists of an envelope that includes SOAP headers and body to store the actual information we want to send whereas REST uses the HTTP build-in headers (with a variety of media-types) to store the information and uses the HTTP Methods such as GET, POST, PUT, PATCH, and DELETE to perform CRUD operations.
- SOAP Performance is slow as compared to REST.

- **WCF:**

- It is a framework used for developing SOA (service-oriented applications).
- WCF can only be consumed by clients, which can understand XML.
- WCF supports protocols like – HTTP, TCP, Named Pipes, etc.

- **Web API:**

- It is a framework that helps us to develop HTTP Based services i.e., Restful Services.
- Web API is an open-source platform.
- It supports most of the MVC features which keep Web API over WCF.

Web API	WCF
- It is used to develop both SOAP-based services and RESTful services.	- It is used to deploy only SOAP-based services.
- It supports various MVC features such as routing, model binding, etc.	- It does not support any MVC features.
- It only supports HTTP protocol.	- It supports various protocols such as HTTP, UDP, custom transport.
- It is considered best for developing RESTFUL services.	- It supports only limited RESTFUL services.
- It is good when one wants to expose an expensive range of clients such as iPhones, browsers, mobile phones, tablets, etc.	- It is good for creating services that uses expedite transport channels such as TCP, UDP, Named pipes, etc.

Web API	WCF
- It offers support for UTF-8 encoding format.	- It offers TEXT, Binary encoding support, MTOM (Message Transmission Optimization Mechanism), etc.

- **What are the advantages of ASP.NET WEB API?**

- It supports all the HTTP features. That means you can use all the built-in HTTP Heapers such as Content-Type, Accept, Authorization, etc. and HTTP Status codes such as 500, 200, 404, etc. and HTTP verbs such as GET, POST, PUT, PATCH, and DELETE to perform CRUD operations.
- It supports Attribute Routing which is good for SEO as well as user-friendly URLs.
- It supports content negotiation i.e., as per the client request, the server sends the response in that format (if possible). The 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.

- **What new features are introduced in ASP.NET Web API 2.0?**

- Attribute Routing
- External Authentication (third party authentication)
- CORS (Cross-Origin Resource Sharing)
- OWIN (Open Web Interface for .NET) Self Hosting
- IHttpActionResult Return type

- The Application_Start() method within the Global.asax file. This method is executed when the application starts for the first time and it is also going to be executed only once. In the Application_Start() method we have the configuration for Filters, Bundles, etc.

- **Content Negotiation:**

- We can define Content Negotiation as "the process of selecting the best representation for a given response when there are multiple representations available".
- One of the standards of the REST service is that the client should have the ability to decide in which format they want the response – whether they want the response in XML or JSON etc. This is called Content Negotiation.
- Now, the fact should be clear that "**ASP.NET Web API Content Negotiation**" means the client and server can negotiate.
- Always It is not possible to return data in the requested format by the Server. That's why it is called negotiation, not demand.
- In such cases, the Web API Server will return the data in the default format.

- **MediaTypeFormatters in ASP.NET Web API:**

- The Media type formatters are the classes that are responsible for serializing the request/response data so that the Web API Framework can understand the request data format and also send data in the format which the client expects.
- ASP.NET Web API includes the following built-in media type formatters:

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

- The Web API Media Type Formatters serializes the object which is going to be returned from the action method into either JSON or XML based on the Accept header value of the HTTP Request.
- Once the ASP.NET Web API Media Type Formatters serializes the data into XML or JSON then these JSON or XML data are written into the response message body.
- The ASP.NET Web API media type formatters that serialize the object into the JSON and XML are JsonMediaTypeFormatter and XmlMediaTypeFormatter respectively.
- Both the classes are deriving from MediaTypeFormatter class. The process through which the MediaTypeFormatter is chosen is called content negotiation.
- You can also specify a quality value indicating the relative preference. The range is 0–1, with 0 being unacceptable and 1 being the most preferred. The default value is 1.

- **GET Implementation:**

```
//GET : Returns list of all employees from database
[HttpGet]
0 references
public HttpResponseMessage ListAllEmployees()
{
    using(EmployeeDBContext dbContext = new EmployeeDBContext())
    {
        var Employees =  dbContext.Employees.ToList();
        return Request.CreateResponse(HttpStatusCode.OK, Employees);
    }

}

//GET: Returns single employee as requested by emp id
0 references
public HttpResponseMessage Get(int id)
{
    using(EmployeeDBContext dBContext = new EmployeeDBContext())
    {
        var entity = dBContext.Employees.FirstOrDefault(e => e.ID == id);
        if(entity != null)
        {
            return Request.CreateResponse(HttpStatusCode.OK, entity);
        }
        else
        {
            return Request.CreateResponse(HttpStatusCode.NotFound, "Employee with ID: "+ id.ToString() +" is not found!");
        }
    }
}
```

- **POST Implementation:**

```
//POST: Create a new object/entry in database
//[HttpPost]
0 references
public HttpResponseMessage Post([FromBody]Employee employee)
{
    try
    {
        using(EmployeeDBContext dBContext = new EmployeeDBContext())
        {
            dBContext.Employees.Add(employee);
            dBContext.SaveChanges();

            var message = Request.CreateResponse(HttpStatusCode.Created, employee);
            message.Headers.Location = new Uri(Request.RequestUri + employee.ID.ToString());
            return message;
        }
    }
    catch(Exception ex)
    {
        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ex);
    }
}
```

- **PUT Implementation:**

```
//PUT: Update the record in the database
//[HttpPut]
0 references
public HttpResponseMessage Put(int id, [FromBody]Employee employee)
{
    try
    {
        using(EmployeeDBContext dBContext = new EmployeeDBContext())
        {
            var emp = dBContext.Employees.FirstOrDefault(e => e.ID == id);
            if(emp != null)
            {
                emp.FirstName = employee.FirstName;
                emp.LastName = employee.LastName;
                emp.Gender = employee.Gender;
                emp.Salary = employee.Salary;

                dBContext.SaveChanges();

                return Request.CreateResponse(HttpStatusCode.OK, emp);
            }
            else
            {
                return Request.CreateErrorResponse(HttpStatusCode.NotFound,
                "Employee with Id " + id.ToString() + " not found to update");
            }
        }
    }
    catch(Exception  ex)
    {
        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ex);
    }
}
```

- **Delete Implementation:**

```
//DELETE : Removes the specific entry from the database
0 references
public HttpResponseMessage Delete(int id)
{
    try
    {
        using(EmployeeDBContext dBContext = new EmployeeDBContext())
        {
            var emp = dBContext.Employees.FirstOrDefault(e => e.ID == id);

            if(emp == null)
            {
                return Request.CreateErrorResponse(HttpStatusCode.NotFound, "The employee with Employee ID: "+ id.ToString() +" not found!");
            }
            else
            {
                dBContext.Employees.Remove(emp);
                dBContext.SaveChanges();
                return Request.CreateResponse(HttpStatusCode.OK, emp);
            }
        }
    }
    catch(Exception ex)
    {
        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ex);
    }
}
```

- **Custom Method Name:**

- We use attribute such as [HttpGet], [HttpPost] to give method a different meaningful name than Get() or Post().

• Parameter Binding:

- The Parameter Binding in ASP.NET Web API means how the Web API Framework binds the incoming HTTP request data (query string or request body) to the parameters of an action method of a Web API controller.
- The ASP.NET Web API action methods can take one or more parameters of different types.
- An action method parameter can be either of a complex type or primitive types.
- The Web API Framework binds the action method parameters either with the URL's query string or from the request body of the incoming HTTP Request based on the parameter type.
- By default, if the parameter type is of the primitive type such as int, bool, double, string, GUID, DateTime, decimal, or any other type that can be converted from the string type then Web API Framework sets the action method parameter value from the query string.
- And if the action method parameter type is a complex type, then Web API Framework tries to get the value from the body of the request and this is the default nature of Parameter Binding in Web API Framework.

HTTP Method	Query String	Request Body
GET	Primitive Type, Complex Type	NA
POST	Primitive Type	Complex Type
PUT	Primitive Type	Complex Type
PATCH	Primitive Type	Complex Type
DELETE	Primitive Type, Complex Type	NA

- We do not have a request body for the GET and DELETE HTTP requests. So, in that case, it tries to get the data from the query string.
- Note that name of the complex type properties and query string parameters must match.
- multiple FormBody is not allowed in a single action.
- We can change this default behaviour of the ASP.NET Web API Parameter Binding process by using **[FromBody]** and **[FromUri]** attributes.

• same-origin policy:

- The browser's default behaviour is that, it allows a web page to make AJAX calls only within the same domain. means the browser security prevents a web page to make AJAX requests to other domains. This is called the same-origin policy.

- The origin of a request consists of Scheme, Host, and Port number. So, two requests are considered to be of the same origin if they have the same scheme, same host, and same port number.
- If any of these differ, then the requests are considered to be cross-origin.
- browsers do not allow cross-domain AJAX requests. There are 2 ways to get around this problem
    1. Using JSONP (JSON with Padding)
    2. Enabling CORS (Cross-Origin Resource Sharing)

- JSONP stands for JSON with Padding. The job of JSONP is to wraps the data into a function.
- Browsers allow consuming JavaScript (JavaScript function) 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.

• ASP.NET Web API Routing:

- The ASP.NET Web API Routing module is responsible for mapping the incoming HTTP requests to a particular controller action method.
- The Route table is created only once during the application start.
- ASP.NET Web API Framework supports two types of routing. They are as follows:
    1. Convention-based Routing.
    2. Attribute Routing.

• Convention-Based Routing:

- In Convention-Based Routing, the ASP.NET Web API Framework uses route templates to determine which controller and action method to execute.
- At least one route template must be added to the route table in order to handle the incoming HTTP requests.
- If you want to allow multiple HTTP verbs on a single action method, then you need to use the [AcceptVerbs] attribute, which takes a list of HTTP methods.
- If you want to prevent an action method from getting invoked as a response to an HTTP request, then you need to decorate that action method with the [NonAction] attribute.
- In ASP.NET Web API by using the [ActionName] attribute you can also override the action method name.
- If you want to prevent an action method from getting invoked as a response to an HTTP request, then you need to decorate that action method with the [NonAction] attribute.
- The Routing module has three main phases:
    1. Matching the URI to a route template.
    2. Selecting a controller.
    3. Selecting an action.

    - The ASP.NET Web API Framework tries to match the segments in the URI path with the route template present in the Route table. The Literals in the template must match exactly.
    - A placeholder matches any value unless we specify some constraints.
    - The Web API framework does not match other parts of the URI such as the hostname or the query parameters.
    - The framework always selects the first route in the routing table that matches the URI.

- When the ASP.NET Web API Framework finds a match for a URI, then it creates a dictionary that will contain the value for each placeholder.
- As we know the dictionary contains the data in the form of a key-value pair. Here, the keys are nothing but the placeholder names but excluding the curly braces, and the values are taken from the URI path or from the defaults. The dictionary is stored in the IHttpRouteData object.

- **Attribute Routing:**

- The ASP.NET Web API 2 and ASP.NET MVC 5 supports a new type of routing called Attribute Routing.
- The Attribute routing provides more control over the URIs in your Web API application by defining routes directly on the actions and controllers.
- We can combine both approaches (Attribute + Conventional Routing) in the same project.
- One advantage of convention-based routing is that all the URI templates are defined in a single place, and the routing rules are applied consistently across all the controllers.
- But the convention-based routing in ASP.NET Web API makes it hard to support certain URI patterns that are common in RESTful APIs. For example, resources often contain child resources: Customers have orders, movies have actors, books have authors, etc. It's natural to create URIs that reflects these relations.

- where attribute routing makes it easy:
  1. API versioning
  2. Overloaded URI segments
  3. Multiple parameter types

- **Optional Parameters / RoutePrefix / Route Name:**

- You can make a URI parameter as optional by adding a question mark ("?") to the route parameter. If you make a route parameter as optional then you must specify a default value by using parameter = value for the method parameter.
- All the routes in the StudentsController start with the same prefix – students that mean students is the common prefix for all the routes available in the Student Controller.
- Here, you can set the common prefix "students" for the entire Student Controller by using the [RoutePrefix] attribute as shown below at the controller level.
- The RoutePrefix attribute is used to specify the common route prefix at the controller level to eliminate the need to repeat the common route prefix on each and every controller action.
- Use ~ character to override the route prefix.
- The Web API Attribute Routing Route Constraints are nothing but a set of rules that we can apply on our routing parameters to restrict how the parameters in the route template are matched.
- In ASP.NET Web API, each and every route has a name. The Route names are useful for generating links so that you can include a link in an HTTP response.
- To specify the route name, we need to set the Name property on the attribute.

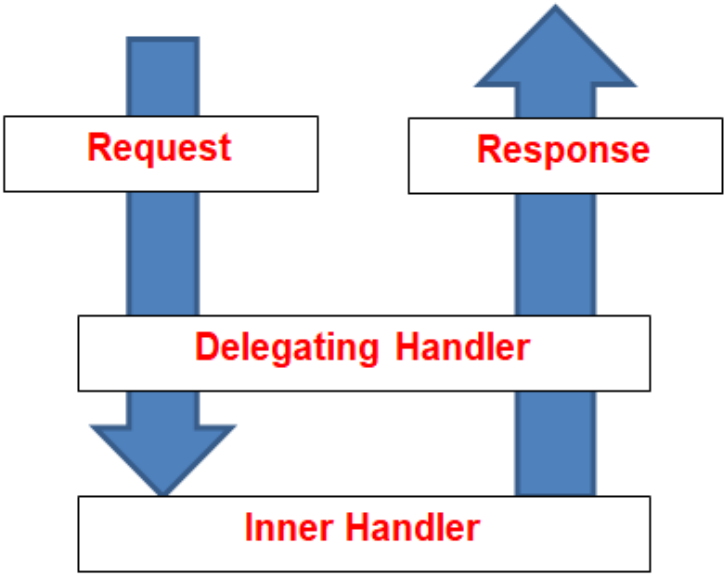
- **Authentication and Authorization:**

- Authentication is the process of identifying the user.
- Authorization is the process of deciding whether the authenticated user is allowed to perform an action on a specific resource (Web API Resource) or not.
- When the host (IIS Server) authenticates the user, it generally creates a principal object (i.e., IPrincipal object) under which the code is going to run. So, once the principal object (IPrincipal object) is created, then the host (i.e., IIS Server) attaches that principal object to the current thread by setting Thread.CurrentPrincipal.
- If you are going to implement your own custom logic for authenticating the user then you can set the principal object at two places which are as follows:
- Thread.CurrentPrincipal: This is the standard way to set the thread's principal in .NET.
- HttpContext.Current.User: This property is specific to ASP.NET.
- The Authorization Process is going to happen before executing the Controller Action Method which provides you the flexibility to decide whether you want to grant access to that resource or not.
- The ASP.NET Web API Framework provides a built-in authorization filter attribute i.e., AuthorizeAttribute and you can use this built-in filter attribute to checks whether the user is authenticated or not. If not, then it simply returns the HTTP status code 401 Unauthorized, without invoking the controller action method.
- You can apply the above built-in filter globally, at the controller level, or at the action level.

- If you want to check the authentication for all the Web API controllers, then it is better to add the AuthorizeAttribute filter to the global filter list within the Register method of the WebApiConfig class.
- If you want to provide authentication for all the action methods of a specific controller, then it is better and recommended to add the Authorize filter at the controller level.
- If you want to provide authentication for specific action methods of a controller, then it is better to add the Authorize filter attribute to the action method which required authentication.
- If we want any action method to be accessed by the anonymous users then we need to decorate that action method with the **[AllowAnonymous]** attribute.
- Along the way, we can also limit access to specific users or to users with specific roles.
- The point to remember here is that the AuthorizeAttribute filter for Web API is located in the System.Web.Http namespace. In MVC there is also an AuthorizeAttribute filter which is located in the System.Web.Mvc namespace, which is not compatible with Web API controllers.

• Server-Side HTTP Message Handlers:

- An HTTP Message Handler in ASP.NET Web API is a class that receives an HTTP request and returns an HTTP response. The Message Handler is derived from the abstract HttpMessageHandler class.
- To handle the HTTP Request and to generate the HTTP Response in ASP.NET Web API, a series of message handlers are chained together. The first handler in the chain receives the HTTP request. Do some processing, and gives the request to the next handler. At some point, the response is generated and goes back up in the chain. This pattern is called delegating handler.
- The following diagram shows this process:



- There are two types of message handlers are available:

    1. Server-Side HTTP Message Handlers
    2. Client-Side HTTP Message Handlers

- ASP.NET Web API Framework uses some built-in message handlers which are as follows:
    1. HttpServer: This built-in message handler gets the request from the host.
    2. HttpRoutingDispatcher: This message handler dispatches the request based on the route.
    3. HttpControllerDispatcher: This message handler sends the request to an ASP.NET Web API controller.
- You can also create your own custom handlers and then add them to the Web API pipeline. The Message handlers are good for cross-cutting concerns (such as authentication and authorization) that operate at the level of HTTP messages rather than controller actions. For example, a custom message handler might do the following things
    1. Read or modify the HTTP request headers.
    2. Add a response header to the HTTP response.
    3. Validate the requests before they reach the controller (i.e., Authentication and Authorization).

• HTTP Client Message Handlers:

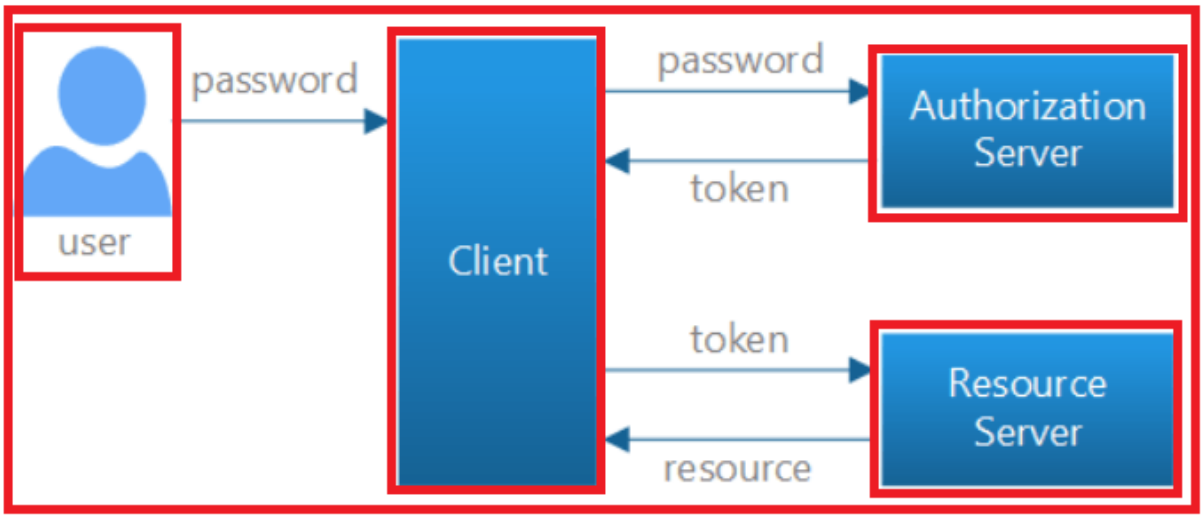
- The HttpClient class uses a message handler to process the requests on the client side.
- The default handler provided by the dot net framework is HttpClientHandler. This HTTP Client Message Handler sends the request over the network and also gets the response from the server.
- To add a custom message handler to HttpClient pipeline, we need to use the HttpClientFactory.Create method as shown below.

HttpClient client = HttpClientFactory.Create(new MessageHandler1(), new MessageHandler2());

- The Message handlers are called in the order that we pass them into the Create method of the HttpClientFactory class. The reason is handlers are nested the response message travels in the other direction. That is, the last handler is the first to get the response message.

• Token Based Authentication

- The most preferred approach nowadays to secure the Web API resources is by authenticating the users in Web API server by using the signed token (which contains enough information to identify a particular user) which needs to be sent to the server by the client with each and every request. This is called the Token-Based Authentication approach.



- Token-Based Authentication works as Follows:
  1. The user enters his credentials (i.e., the username and password) into the client (here client means the browser or mobile devices, etc).
  2. The client then sends these credentials (i.e., username and password) to the Authorization Server.
  3. Then the Authorization Server authenticates the client credentials (i.e., username and password) and generates and returns an access token. This Access Token contains enough information to identify a user and also contains the token expiry time.
  4. The client application then includes the Access Token in the Authorization header of the HTTP request to access the restricted resources from the Resource Server until the token is expired.

• Advantages of using Token Based Authentication:

1. Scalability of Servers:

The token which is sent to the server by the client is self-contained means it holds enough data to identify the user needed for authentication. As a result, you can add easily more servers to your web farm, there is no dependent on shared session stores.

2. Loosely Coupling:

The client application is not tied or coupled with any specific authentication mechanism. The token is generated, validated and perform the authentication are done by the server only.

3. Mobile-Friendly:

The Cookies and browsers like each other, but handling the cookies on native platforms like Android, iOS, Windows Phone is not an easy task. The token-based approach simplifies this a lot.

• Refresh Token:

- A Refresh Token is a special kind of token that can be used to obtain a new renewed access token which allows access to the protected resources. You can request for the new access tokens by using the Refresh Token in Web API until the Refresh Token is blacklisted.
- The idea of using the refresh token is to issue a short-lived access token (up to 30 minutes) for the first time and then use the refresh token to obtain a new access token and use that access token to access the protected resources.
- So, the user needs to provide the username and password along with the client info (i.e., the client id and client secret) to authenticate himself, and if the information provided by the user is valid, then a response contains a short-lived access token along with a long-lived refresh token gets generated.
- The refresh token is not an access token it is just an identifier for the access token. Now once the access token is expired, the user can use the refresh token to obtain another short-lived access token and so on.
- Mainly there are three main reasons to use the refresh tokens are as follows
  1. Updating the Access Token Content.
  2. Revoking the Access from Authenticated users
  3. No need to store or ask for the username and password frequently.

• Web API versioning:

- day by day the business grows and once the business grows then the requirement may change, and once the requirement change then you may need to change the services as well, but the important thing you need to keep in mind is that you need to do the changes to the services in such a way that it should not break any existing client applications who already consuming your services.
- This is the ideal scenario when the Web API versioning plays an important role. You need to keep the existing services as it is so that the existing client applications will not break, they worked as it is, and you need to develop a new version of the Web API service which will start consuming by the new client applications.
- The different options that are available to maintain versioning are as follows
  1. URI's
  2. Query String
  3. Version Header
  4. Accept Header
  5. Media Type

- in the media type, we have specified the version of the service that we want. The custom media types are prefixed with **vnd** which indicates that this media type is a vendor-specific media type.

==========

ASP.NET WEB API Interview Questions

01. What is different between REST API and RESTful API?

- **REST** (Representation State Transfer) **API**: It is basically an architectural style that makes productive use of existing technology and protocols of the web. It is a set of rules that developers need to follow when they develop their API or services that are scalable. It is used with HTTP protocol using its verbs such as GET, DELETE, POST, PUT.

- **RESTful API**: It is simply referred to as web services executing such as architecture.

REST API	RESTful API
REST is an architectural pattern used for creating web services.	RESTful API is used to implement that pattern.
The data format of REST is based on HTTP.	The data format of RESTful is based on JSON, HTTP, and Text.
Working of URL is based on request and response.	Working of RESTful is based on REST applications.
It is more user-friendly and highly adaptable to all business enterprises and IT.	It is too flexible.
It is required to develop APIs that allow interaction among clients and servers.	It simply follows REST infrastructure that provides interoperability among different systems on the whole network.

02. What are the advantages of using Rest in Web API?

REST is very important and beneficial in Web API because of the following reasons:

- It allows less data transfer between client and server.
- It is easy to use and lightweight.
- It provides more flexibility.
- It also handles and controls various types of calls, returning various data formats.
- It is considered best for using it in mobile apps because it makes less data transfer between client and server.

- It uses simple HTTP calls for inter-machine communication rather than using more complex options like CORBA, COM+, SOAP, or RPC.

03. What is REST and SOAP? What is different between them?

REST (Representational State Transfer): It is a new and improved form of web service. It describes the architectural style of networked systems. It does not require greater bandwidth when requests are sent to the server. It just includes JSON message. For example:

```
{"city":"Mumbai","state":"Maharashtra"}
```

SOAP (Simple Object Access Protocol): It is a simple and lightweight protocol that is generally used for exchanging structured and typed information on the Web. It works mostly with HTTP and RPC (Remote Procedure Call). This protocol is mainly used for B2B applications one can define a data contract with it. SOAP messages are heavier in content and therefore use greater bandwidth.

For example:

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope" SOAP-ENV:encodingStyle=" http://www.w3.org/2001/12/soap-encoding">
<soap:Body>
<Demo.guru99WebService xmlns="http://tempuri.org/"> <EmployeeID>int</EmployeeID>
</Demo.guru99WebService>
</soap:Body>
</SOAP-ENV:Envelope>
```

REST	SOAP
It is basically an architectural pattern.	It is basically a messaging protocol.
It usually works with various text formats such as plain text, HTML, JSON, XML, etc.	It only works with XML formats.
It is totally stateless.	It has some specifications for both stateless and stateful implementation.
Its performance is faster as compared to SOAP.	Its performance is slower as compared to REST.
It uses XML and JSON to send and receive data.	It uses WSDL (Web Service Description Language) for communication among consumers or users and providers.
REST has to resend transfer whenever it determines any errors.	SOAP includes built-in error handling for communications errors using WS-ReliableMessaging specification.
It calls services using the URL path.	It calls services by calling RPC (Remote Procedure Call) method.

04. What are Web API filters?

- Filters are basically used to add extra logic at different levels of Web API framework request processing. Different types of Web API filters are available as given below:

01. **Authentication Filter:** It handles authentication and authenticates HTTP requests. It also helps to authenticate user detail. It checks the identity of the user.
02. **Authorization Filter:** It handles authorization. It runs before controller action. This filter is used to check whether or not a user is authenticated. If the user is not authenticated, then it returns an HTTP status code 401 without invoking the action.
03. **AuthorizeAttribute:** is a built-in authorization filter provided by Web API.
04. **Action Filter:** It is attributing that one can apply to controller action or entire controller. It is used to add extra logic before or after controller action executes. It is simply a way to add extra functionality to Web API services.
05. **Exception Filter:** It is used to handle exceptions that are unhandled in Web API. It is used whenever controller actions throw an unhandled exception that is not HttpResponseException. It will implement an "IExceptionFilter" interface.
06. **Override Filter:** It is used to exclude specific action methods or controllers from the global filter or controller level filter. It is simply used to modify the behavior of other filters for individual action methods.

05.What are Exception filters in ASP.NET Web API?

Exception filter is generally used to handle all unhandled exceptions that are generated in web API. It implements IExceptionFilters interface. It is the easiest and most flexible to implement. This filter is executed whenever the controller method throws any unhandled exception at any stage that is not an HttpResponseExecption exception.

06. How to register an exception filter globally?

One can register exception filter globally using following code:

```
GlobalConfiguration.Configuration.Filters.Add(new
MyTestCustomerStore.NotImplExceptionFilterAttribute());
```

07. What do you mean by Caching and What are its types?

Caching is basically a technique or process of storing data somewhere or in the cache for future requests. The cache is a temporary storage area. Caching keeps all frequently or recently accessed files or data in the cache memory and accesses them from the cache itself rather than actual address of data or files. The cache interface simply improves the storage mechanism for request/response object pairs that are being cached.

Advantages of Caching:

- It is considered the best solution to ensure that data is served where it is needed to be served that too at a high level of efficiency which is best for both client and server.
- It delivers web objects faster to the end-user.

- It reduces load time on the website server.
- It leads to faster execution of any process.
- It decreases network costs.

Types of Caching:
There are basically three types of caching as given below:

- Page Caching
- Data Caching
- Fragment Caching

08. Can we return View from ASP.NET Web API method?

- No, we cannot return the view from the ASP.NET Web API method. ASP.NET web API develops HTTP services that provide raw data or information. ApiController in ASP.NET MVC application only renders data that is serialized and sent to the client. One can use a controller to provide normal views.

09. What is the use of DelegatingHandler?

- DelegatingHandler is used to develop a custom Server-Side HTTP Message Handler in ASP.NET Web API. It is used to represent Message Handlers before routing in Web API.

10. What are HTTP Status Codes?

- HTTP Status Code Is 3-digit integer in which the first digit of the Status-Code defines the class of response. Response Header of each API response contains the HTTP Status Code. HTTP Status Codes are grouped into five categories based upon the first number.

Sr. No.	HTTP Status Code	Description
1.	1XX	Informational
2.	2XX	Success
3.	3XX	Redirection
4.	4XX	Client-Side Error
5.	5XX	Server-Side Error

- Some of the commonly seen HTTP Status Codes are: 200 (Request is Ok), 201 (Created), 202 (Accepted), 204 (No Content), 301 (Moved Permanently), 400 (Bad Request), 401 (Unauthorized), 403 (Forbidden), 404 (Not Found), 500 (Internal Server Error), 502 (Bad Gateway), 503 (Service Unavailable) etc.

11. What are the Differences between Code-Behind and Code Inline?

- Code Behind is the code written in a separate class file whereas Code In line is the code written inside an ASP.Net Web Page.
- Code Behind has an extension .aspx.cs or .aspx.vb whereas Code Inline (as it is inside ASP.Net) has an extension .aspx only. Code Inline is written inside <script> tag along with the HTML.
- Code for all the web pages is compiled into a .dll file (Data Link Library File) which is kept free from the Inline Code.

12. What are main return types supported in Web API?

A Web API controller action can return following values:

- **Void** – It will return empty content.
- **HttpResponseMessage** – It will convert the response to an HTTP message.
- **IHttpActionResult** – internally calls ExecuteAsync to create an HttpResponseMessage.
- **Other types** – You can write the serialized return value into the response body.

==========