Contents

1. What is Kestrel and what are advantages of Kestrel in Asp.Net Core?

Asp.Net Core application uses Kestrel by default.

Kestrel is an event-driven, I/O-based, open-source, cross-platform, and asynchronous server which hosts .NET applications. It is provided as a default server for .NET Core; therefore, it is compatible with all the platforms and their versions which .NET Core supports.

It is a listening server with a command-line interface.

It can be used to reverse proxy servers such as IIS, Nginx etc.

Features of Kestrel are:

- Lightweight and fast.
- Cross-platform and supports all versions of .NET Core.
- Supports HTTP & HTTPS.
- Easy configuration
- Multiple apps on the same port is not supported
- Windows authentication is not supported

2. What is the difference between IIS and Kestrel? Why do we need two web servers?

The main difference between IIS and Kestrel is that Kestrel is a cross-platform server. It runs on Windows, Linux, and Mac, whereas IIS only runs on Windows.

Another essential difference between the two is that Kestrel is fully open-source, whereas IIS is closed-source and developed and maintained only by Microsoft.

IIS is very old software and comes with a considerable legacy and bloat. With Kestrel, Microsoft started with high-performance in mind. They developed it from scratch, which allowed them to ignore the legacy/compatibility issues and focus on speed and efficiency.

However, Kestrel doesn’t provide all the rich functionality of a full-fledged web server such as IIS, Nginx, or Apache. Hence, we typically use it as an application server, with one of the above servers acting as a reverse proxy.

3. What is the purpose of launchSettings.json in asp.net core?

The launchSettings.json file is used to store the configuration information, which describes how to start the ASP.NET Core application, in Visual Studio.

It mainly contains the run time profiles to configure application urls' and environment.

The file is used only during the development of the application using Visual Studio. It contains only those settings that required to run the application.

launchSettings.json is only used by Visual Studio.

You don't need launchSettings.json for publishing an app (on production server).

4. What is generic host or HostBuilder in .NET Core?

.NET generic host called 'HostBuilder' helps us to manage all the below tasks.

- Dependency Injection
- Service lifetime management
- Configuration
- Logging

The generic host was previously present as 'Web Host', in .NET Core for web applications. Later, the 'Web Host' was deprecated and a generic host was introduced to cater to the web, Windows, Linux, and console applications.

5. What is the purpose of the .csproj file?

The project file is one of the most important files in our application. It tells .NET how to build (compile) the project.

The csproj file stores list of package dependencies of the current project, target .net version and other compilation settings.

The .csproj file also contains all the information that .NET tooling needs to build the project. It includes the type of project you are building (console, web, desktop, etc.), the platform this project targets, and any dependencies on other projects or 3rd party libraries.

Here is an example of a .csproj file that lists the NuGet packages and their specific versions.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
<PropertyGroup>
  <TargetFramework>net6.0</TargetFramework>
</PropertyGroup>

<ItemGroup>
  <PackageReference Include="PackageName" Version="1.0.0.0" />
</ItemGroup>
</Project>
```

5. What is IIS?

IIS stands for Internet Information Services. It is a powerful web server developed by Microsoft. IIS can also act as a load balancer to distribute incoming HTTP requests to different application servers to allow high reliability and scalability.

It can also act as a reverse proxy, i.e. accept a client's request, forward it to an application server, and return the client's response. A reverse proxy improves the security, reliability, and performance of your application.

A limitation of IIS is that it only runs on Windows. However, it is very configurable. You can configure it to suit your application's specific needs.

6. What is the "Startup" class in ASP.NET core prior to Asp.Net Core 6?

The startup class is the entry point of the ASP.NET Core application. Every Asp.NET Core (Asp.Net Core 5 and earlier) application must have this class. It contains the necessary code to bootstrap the application. This class contains two methods Configure() and ConfigureServices().

- Configure(): The Configure() method is used to essential middleware(s) to the application request pipeline.
- ConfigureServices(): The ConfigureServices() method is used to add services to the IoC container.

In Asp.Net Core 6 (.NET 6), Microsoft unifies Startup.cs and Program.cs into a single Program.cs.

In asp.net core 6 (and up), we need to add all such as registering middleware to the application pipeline, adding services to the IoC container, configuring the ‘application configuration’, configuring the logger, authentication and adding DbContext in Program.cs file.

7. What does WebApplication.CreateBuilder() do?

This method does the following things.

- Configure the app to use Kestrel as web server.
- Specify to use the current project directory as root directory for the application.
- Setup the configuration sub-system to read setting from appsettings.json and appsettings.{env}.json to environment specific configuration.
- Set Local user secrets storage only for the development environment.
- Configure environment variables to allow for server-specific settings.
- Configure command line arguments (if any).
- Configure logging to read from the Logging section of the appsettings.json file and log to the Console and Debug window.
- Configure integration with IIS
- Configure the default service provider.

8. What is HTTP?

HTTP stands for Hypertext Transfer Protocol. It is a set of rule which is used for transferring the information like text, audio, video, graphic image and other multimedia files on the WWW (World Wide Web).

HTTP is a protocol that is used to transfer the hypertext from the client end to the server.

9. What is the format of a Request Message?

HTTP Requests are messages which are sent by the client to initiate an action on the server.

It consists of various things:

1. **Request Line:** The Request-Line includes with HTTP method, Url, HTTP version.

2. **Request Headers:** Contains request-header fields that allow the client to pass additional information to the server, logically equivalent to the parameters while method invocation in a programming language.

3. **Request body:** Contains actual content to send to server; such as query string, JSON, XML etc.

What are the important HTTP methods (or HTTP verbs) – (GET, POST, PUT, PATCH, HEAD, DELETE)?

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource.

The “GET” and “POST” are most used in HTML forms. The default request type in browsers while opening a page, is “GET”.

The “GET”, “POST”, “PUT”, “PATCH”, “HEAD” and “DELETE” are used in RESTful HTTP services, such as “Web API controllers”.

- **GET:** This method retrieves information from the given server using a given URI. GET request can retrieve the data. It cannot apply other side effects (changes) on the data.
- **POST:** The POST request sends the data to the server. For example sending user details in a registration form, or in a login form.
- **PUT:** The PUT method is used to replace (update) an existing record with the provided new record. The client sends the entire record that needs to be updated.
- **PATCH:** The PATCH method is to update a part of an existing record. The client sends part of the record that needs to be updated.
- **HEAD:** The HEAD method is the same as the GET method. It is used to transfer the response start line and headers section only (without response body). It can be used when there is no need of sending response body to server; but the server wants to communicate to the client that the necessary operation (such as database updation) has been completed.
- **DELETE:** The DELETE method is used to remove an existing record based on the parameters supplied in the request.

10. What are the important HTTP status codes?

An HTTP status code is a server response to a browser’s request. It indicates status of completed action as a response to the request.

HTTP status code classes:

- 1xx – Informational
- 2xx – Success
- 3xx – Redirection

- 4xx – Client errors
- 5xx – Server errors

Important HTTP status codes:

- **200 – OK:** The ideal & commonly-used status code that represents normal functioning of a server resource (Eg: page)
- **201 – Created:** Indicates that the server has created (inserted) a record in the data store. It is generally used as response in POST request in RESTful services using as Web API.
- **301 – Moved Permanently:** Represents one URL needs to be redirected to another permanently. A 301 redirect means that visitors and bots that land on that page will be passed to the new URL. That means the first URL will no longer work in future.
- **302 – Found:** Represents a temporary redirection from one URL to another. That means, the first URL might work in the near future.
- **400 – Bad Request:** The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).
- **401 – Unauthorized:** Although the HTTP standard specifies "unauthorized", semantically this response means "unauthenticated". That is, the client must authenticate itself to get the requested response.
- **404 – Not Found:** This means the file or page that the browser is requesting wasn't found by the server. 404s don't indicate whether the missing page or resource is missing permanently or only temporarily.
- **405 – Method Not Allowed:** This is mostly used for RESTful services such as Web API. It indicates that the request method is known by the server but the HTTP method is not supported by the server resource. For example, an API may allow GET and POST only; may not allow calling PUT request.
- **500 – Internal Server Error:** Indicates there is some runtime error (exception) while executing the code at server side.

11. What is Content Negotiation in HTTP?

In HTTP, content negotiation is the mechanism that is used for serving different representations of a resource to the same URI to help the user agent specify which representation is best suited for the user (for example, which document language, which image format, or which content encoding).

For example, the client may ask for XML data instead of receiving content in JSON format.

It is generally done using “Accept” request header.

Eg:

Accept: application/xml

Explain how HTTP protocol works?
Hypertext Transfer Protocol (HTTP) is an application-layer protocol for transmitting hypermedia documents, such as HTML. It handles communication between web browsers and web servers. HTTP follows a classical client-server model. A client, such as a web browser, opens a connection to make a request, then waits until it receives a response from the server.

HTTP is a protocol that allows the fetching of resources, such as HTML documents. It is the foundation of any data exchange on the Web, and it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser.

12. What is a web server?

The term web server can refer to both hardware and software, working separately or together.

On the hardware side, a web server is a computer with more processing power and memory that stores the application’s back-end code and static assets such as images and JavaScript, CSS, HTML files. This computer is connected to the internet and allows data flow between connected devices.

On the software side, a web server is a program that accepts HTTP requests from the clients, such as a web browser, processes the request, and returns a response. The response can be static, i.e. image/text, or dynamic, i.e. calculated total of the shopping cart.

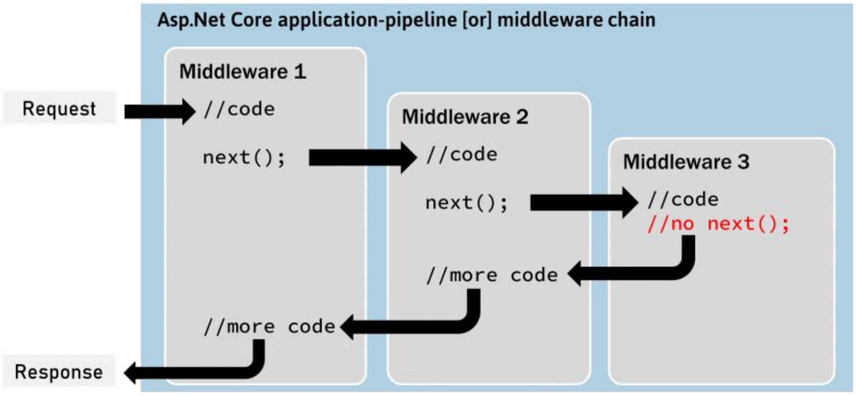
Popular examples of web servers include Apache, Nginx, and IIS.

13 What is middleware?

It is the code that is injected into the application pipeline to handle requests and responses. They are just like chained to each other and form as a pipeline. The incoming requests are passed through this pipeline where all middleware is configured, and middleware can perform some action on the request before passing it to the next middleware. Same as for the responses, they are also passing through the middleware but in reverse order.

The main responsibilities of a middleware:

- Generate an HTTP response for an incoming HTTP request
- Intercept and make changes to an incoming HTTP request and pass it on to the next piece of middleware.
- Intercept and make changes to an outgoing HTTP response, and pass it on to the next piece of middleware.



14 What is the difference between IApplicationBuilder.Use() and IApplicationBuilder.Run()?

We can use both the methods while defining application request pipeline. Both are used to add middleware delegates to the application request pipeline. The middleware that is added using IApplicationBuilder.Use() may call the next middleware in the pipeline whereas the middleware that is added using the IApplicationBuilder.Run() method never calls the subsequent middleware. After IApplicationBuilder.Run method, system stop adding middleware in the request pipeline.

The IApplicationBuilder.Run() adds a terminating middleware; so no other middleware will run after the middleware added with Run().

```
}
```

15. What is the use of the "Map" extension while adding middleware to the ASP.NET Core pipeline?

It is used for branching the pipeline. It branches the ASP.NET Core pipeline based on request path matching. If the request path starts with the given path, middleware on to that branch will execute.
How do you create a custom middleware?
The custom middleware class can be used to separate the middleware code from the Program.cs (or Startup.cs in earlier versions).

You can create a custom middleware either by implementing IMiddleware interface or by using convention middleware

Implementing IMiddleware:

```
public class MyCustomMiddleware : IMiddleware
{
 public async Task InvokeAsync(HttpContext context, RequestDelegate next)
 {
  //TO DO: before logic
  await next(context);
  //TO DO: after logic
 }
}
```

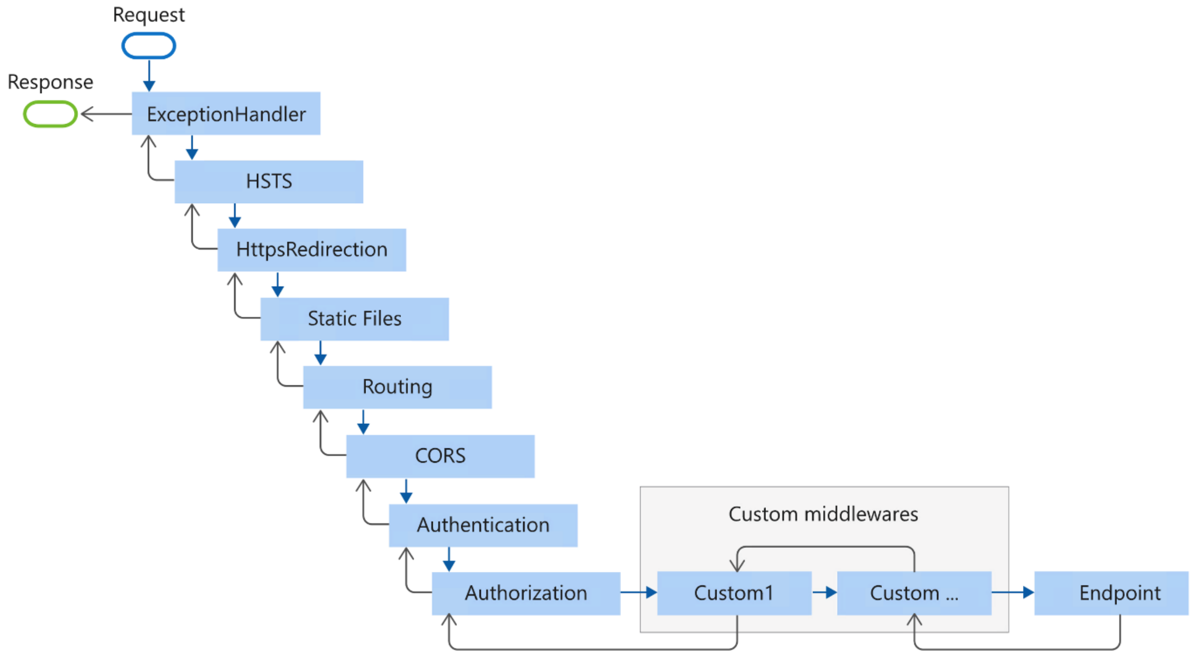
Convention Middleware:

```
public class MyCustomMiddleware
{
 private readonly RequestDelegate _next:
 public MiddlewareClassName(RequestDeletegate next)
 {
 }

 public async Task InvokeAsync(HttpContext context)
 {
  //TO DO: before logic
  await _next(context);
  //TO DO: after logic
 }
}
```

16. What is the right order of middleware used in production-level applications?

You can see how, in a typical app, existing middlewares are ordered and where custom middlewares are added. You have full control over how to reorder existing middlewares or inject new custom middlewares as necessary for your scenarios.



17. What is Routing?

Routing is functionality that map incoming request to the route handler. The route can have route parameters to receive values from the URL. Using the route, routing can find a route handler based on the URL. All the routes are registered when the application is started. There are two types of routing supported by ASP.NET Core

1. The conventional routing
2. Attribute routing

The Routing uses routes to map incoming requests with the route handler and Generates URL that is used in response. Mostly, the application has a single collection of routes and this collection is used for the process of the request. The RouteAsync method is used to map incoming requests (that match the URL) with available in route collection.

-by Sneh Gour

18. How Routing works in ASP.NET Core?

Routing is used to handle incoming HTTP requests for the app. Routing finds matching executable endpoint for incoming requests. These endpoints are registered when app starts. Matching process use values from incoming request url to process the requests. You can configure the routing in middleware pipeline of configure method in startup class.

```
app.UseRouting(); // It adds route matching to middleware pipeline

// It adds endpoints execution to middleware pipeline
app.UseEndpoints(endpoints =>
{
 endpoints.MapGet("/", async context =>
 {
  await context.Response.WriteAsync("Hello World!");
 });
});
```

19. When will you prefer attribute routing over conventional routing?

In ASP.NET Core, you can define routes for your web application using either attribute routing or conventional routing. The choice between the two depends on the complexity and requirements of your application. Here are some scenarios where you might prefer attribute routing over conventional routing:

1. Fine-grained control: Attribute routing allows you to define routes directly on the action methods or controllers using attributes. This gives you more granular control over the routing behavior for specific actions. You can have different routes for different actions, making it easier to manage and understand the routing logic.
2. Controller-specific routes: With attribute routing, you can define routes specific to a controller by placing the routing attributes directly on the controller class. This is useful when you have actions that share a common route prefix, simplifying the route definitions.
3. Action-specific routes: In some cases, you may want to have multiple routes for a single action method. Attribute routing allows you to define additional routes for an action by adding multiple attributes on the same method.
4. Complex routing requirements: If your application has complex routing requirements, attribute routing can provide a more straightforward and

concise way to handle those scenarios. You can use route parameters, optional segments, and custom route constraints directly in the attribute, making the routing logic more readable.

On the other hand, conventional routing can be more suitable for simpler applications with straightforward routing requirements. It relies on the route template defined in the Startup class, which makes it easier to see all the routes at once. It's a good choice when your application has a small number of routes and doesn't require fine-grained control over individual actions.

20. What are the important route constraints?

In ASP.NET Core, route constraints are used to restrict the values that can be matched for route parameters. They are an essential part of defining more specific and controlled routes. Here are some important route constraints you can use:

1. **int**: Constrains the parameter to be an integer.
2. **long**: Constrains the parameter to be a long integer.
3. **bool**: Constrains the parameter to be a Boolean value, i.e., "true" or "false".
4. **double**: Constrains the parameter to be a double-precision floating-point number.
5. **float**: Constrains the parameter to be a floating-point number.
6. **guid**: Constrains the parameter to be a GUID (Globally Unique Identifier).
7. **datetime**: Constrains the parameter to be a valid date and time value.
8. **alpha**: Constrains the parameter to contain only letters (no digits or special characters).
9. **regex**: Allows you to define a custom constraint using a regular expression pattern.
10. **length**: Constrains the parameter to have a specific length. For example, `{id:length(5)}` will only match when the `id` parameter has a length of 5 characters.
11. **min** and **max**: Allows you to specify minimum and maximum values for numeric parameters. For example, `{age:min(18)}` will only match if the `age` parameter is 18 or greater.
12. **range**: Similar to min and max, but allows you to specify a range of values. For example, `{year:range(1900, 2023)}` will only match if the `year` parameter is between 1900 and 2023.
13. **required**: Indicates that the parameter is required and must be present in the URL for the route to match.

14. **nonempty**: Ensures that the parameter is not empty (not null, empty string, or whitespace).
15. **maxlength** and **minlength**: Restricts the length of a string parameter. For example, `{username:maxlength(20)}` will only match if the `username` parameter has a length of 20 characters or less.

21. What is the purpose of the wwwroot folder?

The `wwwroot` folder is a special folder in an ASP.NET Core web application that serves as the web root. Its purpose is to store static files, such as HTML, CSS, JavaScript, images, and other client-side assets that need to be directly accessible by the web browser.

When a web application receives a request, the web server looks for the requested resource within the `wwwroot` folder. If the resource is found in this folder, the web server serves it directly to the client without involving the ASP.NET Core middleware pipeline.

How do you change the path of wwwroot folder?
We need to set path of the wwwroot folder in the WebRootPath propertyof the WebApplicationOptions class.

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions() {
 WebRootPath = "foldername"
});
```

22. What is Controller?

Controller is a class that is used to group-up a set of action methods.

Action methods do perform certain operation when a request is received & returns the IActionResult that can be sent as response to browser.

It performs the following tasks:

- Reading requests such as receiving query string parameters, request body, request cookies, request headers etc.
- Validation of the request details.
- Invoking models (business logic)
- Creating objects of ViewModel or DTO
- Sending DTO objects to view (in case of view result)

The controller class should be either or both:

- The class name should be suffixed with “Controller”. Eg: HomeController
- The [Controller] attribute is applied to the same class or to its base class.

23. What is an Action Method?

An action method is a method in a controller class with the following restrictions:

- It must be public. Private or protected methods are not allowed.
- It cannot be overloaded (unless you use different Http methods or different action names).
- It cannot be a static method.
- An action method executes an action in response to an HTTP request.

24. Explain different types of Action Results in asp.net core?

Controller is a class that is used to group-up a set of action methods.

IActionResult

Defines a contract that represents the result of an action method.

ActionResult

A default implementation of IActionResult.

ContentResult

Represents a text result.

EmptyResult

Represents an ActionResult that when executed will do nothing.

JsonResult

An action result which formats the given object as JSON.

PartialViewResult

Represents an ActionResult that renders a partial view to the response.

ViewResult

Represents an ActionResult that renders a view to the response.

ViewComponentResult

An IActionResult which renders a view component to the response.

StatusCodeResult

An IActionResult which sends a specific HTTP status code in response, without any response body.

UnauthorizedResult

An IActionResult which sends HTTP 401 status code in response, with / without any response body.

BadRequestResult

An IActionResult which sends HTTP 400 status code in response, with / without any response body.

NotFoundResult

An IActionResult which sends HTTP 404 status code in response, with / without any response body.

ObjectResult

An IActionResult which sends the data of the specified object in response body.

FileResult

An IActionResult which sends content of the specified file in response body.

RedirectToActionResult

An IActionResult which sends HTTP 301 or 302 status code in response to redirect the request to the specific action method.

LocalRedirectResult

An IActionResult which sends HTTP 301 or 302 status code in response to redirect the request to the specified local URL (within the same domain).

RedirectResult

An IActionResult which sends HTTP 301 or 302 status code in response to redirect the request to the specified local URL (within the same domain) or an external URL.

25. What’s the HttpContext object? How can you access it within a Controller?

HttpContext encapsulates all HTTP-specific information about an individual HTTP request. You can access this object in controllers by using the ControllerBase.HttpContext property.

The HttpContext object has properties such as Items, Request, Response, Session, User etc.

26. What is model binding in ASP.NET CORE?

Controllers and views need to work with data that comes from HTTP requests. For example, routes may provide a key that identifies a record, and posted form fields may provide model properties. The process of converting these string values to .NET objects could be complicated and something that you have to do with each request. Model binding automates and simplifies this process.

The model binding system fetches the data from multiple sources such as form fields, route data, and query strings. It also provides the data to controllers and views in method parameters and properties, converting plain string data to .NET objects and types in the process.

Example:

Let’s say you have the following action method on the PostsController class:

```
[HttpGet("posts/{id}")]
public ActionResult<Post> GetById(int id, bool archivedOnly)
```

And the app receives a request with this URL:

```
http://yourapp.com/api/Posts/5?ArchivedOnly=true
```

After the routing selects the action method, model binding executes the following steps.

Locate the first parameter of GetByID, an integer named id, look through the available sources in the HTTP request and find id = "5" in route data.

Convert the string "5" into an integer 5.

Find the next parameter of GetByID, a boolean named archivedOnly.

Look through the sources and find "ArchivedOnly=true" in the query string. It ignores the case when matching the parameters to the strings.

Convert the string "true" into boolean true.

Some other examples of attributes include:

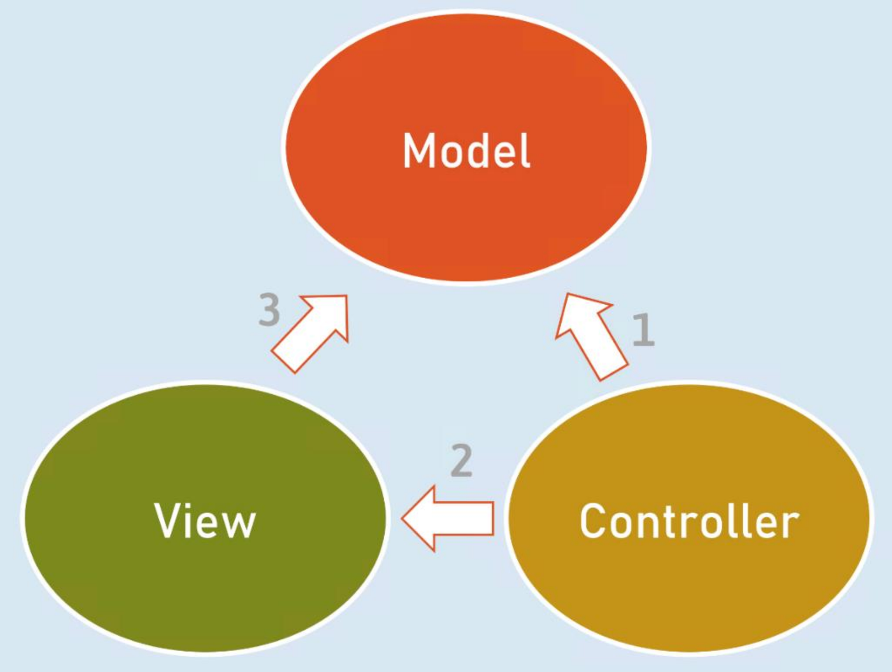
1. [FromQuery] - Gets values from the query string.
2. [FromRoute] - Gets values from route data.
3. [FromForm] - Gets values from posted form fields.
4. [FromBody] - Gets values from the request body.
5. [FromHeader] - Gets values from HTTP headers.

27. How validation works in ASP.NET CORE MVC and how they follow DRY principle?

ASP.NET CORE MVC supports the DRY (Don’t Repeat Yourself) principle where you specify the behavior once and it reflects at multiple places in the application. Model Validation is one such example where you can specify validation rules by using data annotations in the model class and enforce the rules everywhere else (in controller and view) in the application.

28. What is the MVC pattern?

The Model-View-Controller (MVC) is an architectural pattern that separates an application into three main logical components: the model, the view, and the controller. It’s different to note that this pattern is unrelated to the layered pattern we saw earlier. MVC pattern operates on the software side, while the layered pattern dictates how and where we place our database and application servers.



In an application that follows the MVC pattern, each component has its role well specified. For example, model classes only hold the data and the business logic. They don't deal with HTTP requests. Views only display information. The controllers handle and respond to user input and decide which model to pass to which view. This is known as the separation of responsibility or separation of concerns. It makes an application easy to develop and maintain over time as it grows in complexity.

Though Model-View-Controller is one of the oldest and most prominent patterns, alternate patterns have emerged over the years. Some popular patterns include MVVM (Model-View-ViewModel), MVP (Model-View-Presenter) and MVA (Model-View-Adapter).

29 Explain the role of the various components of the MVC pattern?

Model: Represents all the data and business logic that the user works within a web application. In ASP.NET core, the model is represented by C# classes that hold the data and the related logic that operates on that data. The 'Models' directory stores the model classes. You can also write POCO (Plain-Old-CLR-Object) classes only for storing the data and write business logic in a separate model class (a.k.a ‘Services’).

View: Represents all the UI logic of the application. In a web application, it represents the HTML that's sent to the user and displayed in the browser.

One important thing to remember is that all HTML code is not static or hard-coded. The HTML code in view can be generated dynamically using a model's data.

Controller: Acts as an interface between Model and View. It processes the business logic and incoming requests, manipulates data using the Model, and interacts with the Views to render the final output.

In ASP.NET, these are C# classes that form the glue between a model and a view. Controllers have action methods that act as middleware that execute upon receiving a HTTP request from the browser, then retrieve the data from model and pass it to the view to dynamically render a response. The controllers can be present in any folder – the common name of the folder is “Controllers”.

30 Explain the differences between ViewData and ViewBag

1:ViewData is Key-Value paired Dictionary collection.

ViewBag is internally “object” type.

2:ViewData is a property of “Microsoft.AspNetCore.Mvc.Controller” class.

ViewBag is “dynamic” property of “Microsoft.AspNetCore.Mvc.Controller” class.

3:ViewBag internally uses ViewData. So you can set / get data of ViewData using ViewBag.

ViewBag is the syntactic sugar to easily access the ViewData.

4:Type Conversion code is required while reading values from ViewData.

Type conversion is not required while reading values from ViewBag.

5:The lifetime of both ViewData and ViewBag is per-request. They will be deleted automatically at the end of the request processing. When a new request is received, a new ViewData will be created.

31. Explain strongly-typed views.

Strongly-typed views are tightly bound to a model.

The strongly-typed view can receive model object (of specific model class) from the controller.

The controller can supply model object by using the return View(model) method, at the end of action method.

You can use the strongly-typed tag helpers (such as asp-for) in strongly-typed views.

You will create a strongly typed view by mentioning the model class by using @model directive at the top of the view.

@model ModelClassName

32. Explain the purpose and usage of layout views in asp.net core?

- Layout views help to maintain consistent UI across all pages of the web application.
- Layout views contain common HTML structures such as scripts and stylesheets are also frequently used by many pages within an app. All of these shared elements may be defined in a layout file, which can then be referenced by any view used within the app. Layouts reduce duplicate code in views.
- Generally, layout views are presented with in the "Views/Shared" folder.
- Their name is recommended to be prefixed with "_" (underscore).
- Layout views contains @RenderBody() to represent the position to render the content of a view.
- You can reference a layout view from a view by using view's "Layout" property.
- Common layout view path can be applied by using _ViewStart.csthml file either in the "Views" folder or in the "Views\ControllerName" folder. The inner-most takes precedence, because it can override the outer setting.
- Nested layout views are possible.

33. Explain partial views in asp.net core?

A partial view is a Razor markup file (.cshtml) that renders HTML output within another markup file's rendered output.

Partial views are an effective way to:

- Break up large views into smaller components.
- Reduce the duplication of common markup content across views.

You will invoke the partial view either by using the <partial> tag helper or an asynchronous html helper called Html.PartialAsync() or Html.RenderPartialAsync().

34. Explain the difference between PartialAsync() and RenderPartialAsync()

1:The Html.PartialAsync() method returns the content to the parent view.

The Html.RenderPartialAsync() method streams the content to the browser; so it offers faster performance is there is large amount of content in the partial view.

2:The return type of PartialAsync() method is IHtmlContent. Hence its output can be stored in a variable.

The return type of RenderPartialAsync() method is void. So you can’t store its output into a variable. It will be directly streamed to the browser.

For faster performance in case of larger partial views, RenderPartialAsync() is recommended.

35. Describe view components?

ViewComponent was introduced in ASP.NET Core MVC. It can do everything that a partial view can and can do even more. ViewComponents are completely self-contained objects that consistently render html from a razor view. ViewComponents are very powerful UI building blocks of the areas of application which are not directly accessible for controller actions.

Let's suppose we have a page of social icons and we display icons dynamically. We have separate settings for color, urls and names of social icons and we have to render icons dynamically. It’s good to create it as a ViewComponent, because we can pass data to ViewComponent dynamically.

```
public class MyViewComponent : ViewComponent
{
 public async Task<IViewComponentResult> InvokeAsync()
 {
  return View(model); //invokes the view of ViewComponent
 }
}
```

36. When would you use ViewComponent over a partial view?

If a view needs to render partial view without needing to pass any model data, partial view could be enough.

But if we would like to pass model data (generally, data from database), view component is recommended.

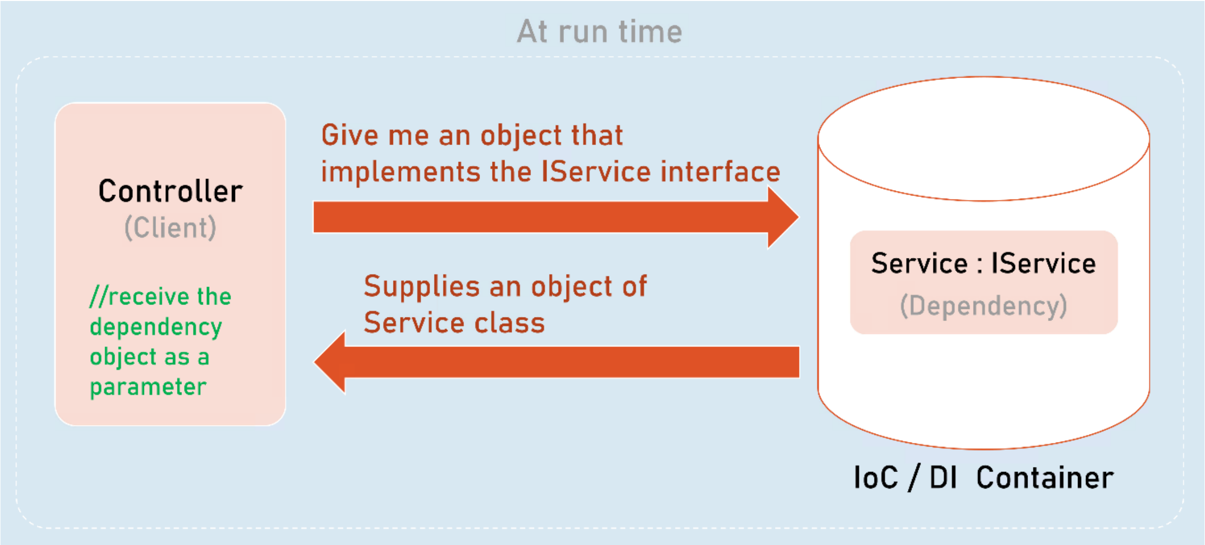
View Components are completely self-contained objects that consistently render html from a Razor view. View Components offers separation of concerns as they don’t depend on controllers.

37. Explain how dependency injection works in ASP.NET Core?

ASP.NET Core injects instances of dependency classes by using the built-in IoC (Inversion-of-Control) container. This container is represented by the IServiceProvider interface.

The types (classes) managed by the container are called services. We first need to register them with the IoC container in the Startup class.

ASP.NET Core supports two types of services, namely framework and application services. Framework services are a part of ASP.NET Core framework such as ILoggerFactory, IHostingEnvironment, etc. In contrast, a developer creates the application services (custom types or classes) specifically for the application.


At run time
Controller
(Client)
//receive the dependency object as a parameter
Give me an object that implements the IService interface
Supplies an object of Service class
Service : IService
(Dependency)
IoC / DI Container

ASP.NET Core has dependency injection to manage services; are you aware of the different lifetimes? What are they, and what does each mean?

ASP.NET Core has three lifetimes of Singleton, Scoped, and Transient.

1. **Singleton:** ASP.NET Core services container will create services registered as a singleton only once for the duration of the application's lifetime. Singletons are helpful for expensive services or services with little to no internal state.

2. **Scoped:** As the name suggests, with regard to scoped services, they are created within a scope. The scope is typically the lifetime of an HTTP request,

but not necessarily always. I, as a developer, might create custom scopes in code, but anyone should be careful to use this technique sparingly.

3. **Transient:** Finally, ASP.NET Core creates transient services when a dependent instance asks for them. I might consider registering dependencies as transient as the “safest” approach to creating dependencies as there’s no chance for contention, race conditions, or deadlocks. Still, it can also come at the expense of performance and resource utilization.

Each lifetime has its use, and it depends on the dependency we are registering.

38. What are the benefits of Dependency Injection?

DI helps to implement decoupled architecture where you change at one place and changes are reflected at many places.

Dependency injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself. When using dependency injection, objects are given their dependencies at run time rather than compile time (car manufacturing time).

- It allows your code to be more loosely coupled because classes do not have hard-coded dependencies
- Decoupling the creation of an object (in other words, separate usage from the creation of an object)
- Making isolation in unit testing possible/easy. It is harder to isolate components in unit testing without dependency injection.
- Explicitly defining dependencies of a class
- Facilitating good design like the single responsibility principle (SRP) for example
- Promotes Code to an interface, not to implementation principle
- Enabling switching/ability to replace dependencies/implementations quickly (Eg: DbLogger instead of ConsoleLogger)

39. What is IoC (DI) Container?

A Dependency Injection container, sometimes, referred to as DI container or IoC container, is a framework that helps with DI. It “creates” and “injects” dependencies for us automatically.

40. What is Inversion of Control?

Inversion of control is a broad term but for a software developer it's the most commonly described as a pattern used for decoupling components and layers in the system.

It inverses the control by shifting the control to IoC container.

For example, say your application has a text editor component and you want to provide spell checking. Your standard code would look something like this:

```
public class TextEditor {
 private SpellChecker checker;

 public TextEditor() {
  this.checker = new SpellChecker();
 }
}
```

What we've done here creates a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

```
public class TextEditor {
 private IocSpellChecker checker;

 public TextEditor(IocSpellChecker checker) {
   this.checker = checker;
 }
}
```

You have inverted control by handing the responsibility of instantiating the spell checker from the TextEditor class to the caller.

```
SpellChecker sc = new SpellChecker; // dependency
TextEditor textEditor = new TextEditor(sc);
```

41. How do you create your own scopes in asp.net core?

We can create child scopes by using IServiceScopeFactory.

```
//controller
public ControllerName(IServiceScopeFactory serviceScopeFactory)
{
  _serviceScopeFactory = serviceScopeFactory;
}

[Route("route-path")]
public IActionResult ActionMethod()
{
 using (IServiceScope scope = _serviceScopeFactory.CreateScope())
 {
  IService service = scope.ServiceProvider.GetRequiredService<IService>();
  //call service methods here
 }
 return View();
}
```

42. How do you inject a service in view?

We can do that by using @inject directive in razor view.

Eg:

```
@inject IService service
```

43. Why you prefer Autofac over built-in Microsoft DI?

Autofac is an IoC container for .NET. It manages the dependencies between classes so that applications stay easy to change as they grow in size and complexity. Autofac is the most popular DI/IoC container for ASP.NET core.

Though the default Microsoft DI may offer enough functionality, there is a certain limitations like resolving a service with some associated Metadata, Named/Keyed services, Aggregate Services, Multi-tenant support, lazy instantiation, and much more. As the system grows you might need such features, and Autofac gives you all these features.

44. What exception do you get when a specific service that you injected, can’t be found in the IoC container?

I’ll get an “InvalidOperationException” with error message “Unable to resolve service for type 'type’ while attempting to activate ‘class_name'.

I’ll get above exception when I injected a service class into another service or controller; but the service that I injected is not added to the IoC container at application startup.

45. What is the purpose of the appsettings.json file?

Appsettings.json contains all of the application's configuration settings, which allow you to configure your application behavior.

It includes with configuration settings related to logging, connection strings etc.

You can also write environment-specific configuration with “appsettings.Environment.json” file.

You can also load custom json files, custom INI files, InMemory configuration or Secrets manager to store configuration settings.

46. You have configuration values needed to access your application resources. Which configuration providers do you prefer for development and which do you prefer for production?

Many cloud providers and Docker hosting platforms support environment variables, so environment variables make much sense for production environments. However, in case of sensitive information, I prefer the user secrets configuration provider for local development as there’s no way to add sensitive secrets to source control mistakenly. Finally, for non-sensitive data, I like JSON configuration since it’s one of the default configuration options, and it’s easy to add and manage into source control.

47. How do you use Options pattern in Asp.Net Core?

The configuration system in ASP.NET allows (actually, enforces) strongly typed settings using the IOptions<> pattern.

Services:

```
app.Services.Configure<Model>(builder.Configuration.GetSection(“ParentKey”));
```

Controller:

```
private readonly Model _options;

public ControllerName(IOptions<Model> options)
{
  _options = options.Value; //returns an instance of Model class that has
configuration values from appropriate configuration sources
}
```

48. How do you enable Secrets manager and why?

A feature in ASP.NET Core named User Secrets allows you to store user secrets outside your project tree in a JSON file, and can even be managed using a command-line tool called the Secrets Manager.

So as a benefit, you will not push sensitive values like passwords or API keys into your source control – so those values are kept secret to the same developer machine.

To enable & store secrets using “secret manager tool” (in PowerShell):

```
dotnet user-secrets init
dotnet user-secrets set “Key” “value”
dotnet user-secrets list
```

49. Explain how attribute-based routing works?

Attribute routing is an alternative routing strategy for conventional routing. It can be used for both MVC controllers and Web API controllers. It uses a set of attributes to map action methods directly to route templates.

Attribute routing directly defines the routes on action methods. We can also use these attributes on the controllers. It enables us to get fine-grained control over what routes map to which actions. With attribute routing, the controller and action names play no part in determining the action method.

For example, we use attributes Blog and Home to map an incoming URL such as myapp.com/blog/post/3 to the Show method on the PostsController.

```
[Route("blog")]
public class PostsController : Controller
{
 [HttpGet("post/{id:int}")]
 public IActionResult Show(int id = 0)
 {
  Post post = new Post()
  {
   ID = id
  };
 return View("Show", post);
}

 [HttpGet("edit/{id:int}")]
 public IActionResult Edit(int id)
 {
  Post postToEdit = _service.Get(id);
  return View("Edit", postToEdit);
 }
}
```

In the above example, the attribute [Route(“blog”)] is placed on the controller, whereas the route [HttpGet(“post/{id:int}”)] is placed on the action method. A controller route applies to all actions in that controller. For example, the second [“edit/{id:int}”] route matches the url myapp.com/blog/edit/3.

In addition to the above route templates, ASP.NET Core provides the following HTTP verb templates.

- [HttpGet]
- [HttpPost]
- [HttpPut]
- [HttpDelete]
- [HttpHead]
- [HttpPatch]

You can map routes to endpoints explicitly (attribute routing) or through convention (convention routing); which do you prefer and why?

I prefer explicitly registering routes, as they are visible in the codebase and often easier to rationalize and debug. There is a potential drawback, though, as the more routes an application has, the more it can impact route resolution and performance of an application. Performance degradation can happen in applications when an extreme amount of route registrations occur, but many folks shouldn’t worry about it until they notice a drop in performance.

50. You have a page with a form, but when you submit, nothing occurs. How would you go about debugging the issue?

I always find it’s best to start at the beginning. In my case, I clicked a submit button on a form. But, first, I would open the dev tools in my browser and make sure the client page made a network call to the backend ASP.NET Core app.

If the page didn’t make a request, I would ensure that my form has an action attribute pointing at a known endpoint in my application and that the method matches what the app expects on the endpoint.

If the client made a request, before leaving the dev tools of the client, I would read any responses sent from the server to pinpoint the exact point of failure. A “not found” response would lead me to believe the endpoint isn’t registered in my app correctly. If the endpoint does exist, I will look at any route constraints, filters, or exceptions that may stop the request from getting to my endpoint. The typical response, in this case, is a “bad request” response. An excellent place to look is the logs to see if any error messages are visible.

Methodically starting from the origin of the problem and working backward is an excellent way to fix issues and quickly move on to more work.

51. How do you implement buffering and streaming file uploading files into asp.net core app?

Buffering:

The entire file is read into a IFormFile object at-a-time

It’s good for smaller files because, the server’s disk will be utilized during this process.

```
public IActionResult ActionMethod(IFormFile file)
{
  //”file” represents the buffered file that is uploaded from the client
}
```

Streaming:

In this approach, the file is uploaded in a multipart request and directly processed or saved by the application. For uploading file streaming approach consumes less memory or disk space as compared to the buffering approach. Streaming should be the preferred approach when uploading larger files on the webserver.

```
public IActionResult ActionMethod()
{
 using MemoryStream stream = new MemoryStream())
 {
   await Request.Body.CopyToAsync(memoryStream);
   //the “stream” streams the file that is uploaded from the client
 }
}
```

52. What is the difference between ViewModel and DTO?

The canonical definition of a DTO is the data shape of an object without any behavior.

ViewModels are the model of the view.

Basically, both serve the same purpose but DTO is a broader term - it can be used among any two layers of the app.

Eg:

- between Controller to View
- between HttpRequest to Controller
- between Controller to Business layer (and vice versa)
- between Business layer to DAL (and vice versa)

53. Explain tag helpers

Tag Helpers in ASP.NET Core are the server-side components. They are basically used to perform defined transformations on HTML Elements. As they are server-side components, so they are going to be processed on the server to create and render HTML elements in the Razor files.

They allow you to conditionally modify or add HTML elements from server-side code.

There are built-in tag helpers such as asp-controller, asp-action, asp-route-x, asp-for etc.

The best use case of tag helpers is binding the input tag with a model property using “asp-for” tag helper.

54. What is Entity Framework?

Working with databases can often be rather complicated. You have to manage database connections, convert data from your application to a format the database can understand, and handle many other subtle issues.

The .NET ecosystem has libraries you can use for this, such as ADO.NET. However, it can still be complicated to manually build SQL queries and convert the data from the database into C# classes back and forth.

EF, which stands for Entity Framework, is a library that provides an object-oriented way to access a database. It acts as an object-relational mapper, communicates with the database, and maps database responses to .NET classes and objects.

Entity Framework (EF) Core is a lightweight, open-source, and cross-platform version of the Entity Framework.

Here are the essential differences between the two:

- **Cross-platform:** We can use EF Core in cross-platform apps that target .NET Core. EF 6.x targets .NET Framework, so you're limited to Windows.
- **Performance:** EF Core is fast and lightweight. It significantly outperforms EF 6.x.

55. What other libraries or frameworks might you use with ASP.NET Core to build your application, and for what purposes?

Since most applications need to store data, I'd likely reach for Entity Framework Core or Dapper for data access.

In addition, I'm also a big fan of FluentValidation to make validating user input easier to understand.

I'm pretty comfortable in writing unit tests, integration tests with xUnit, FluentAssertions and Moq. xUnit is quite advanced and extensible. FluentAssertions makes it easy to write human-readable & close-to-natural-language assert statements. Moq helps me to mock services.

56. What is SQL injection attack?

A SQL injection attack is an attack mechanism used by hackers to steal sensitive information from the database of an organization. It is the application layer (means front-end) attack which takes benefit of inappropriate coding of our applications that allows a hacker to insert SQL commands into your code that is using SQL statement.

SQL Injection arises since the fields available for user input allow SQL statements to pass through and query the database directly. SQL Injection issue is a common issue with an ADO.NET Data Services query.

57. How to handle SQL injection attacks in Entity Framework?

Entity Framework is injection safe since it always generates parameterized SQL commands which help to protect our database against SQL Injection.

A SQL injection attack can be made in Entity SQL syntax by providing some malicious inputs that are used in a query and in parameter names. To avoid this one, you should never combine user inputs with Entity SQL command text.

58. What are POCO classes?

The term POCO does not mean to imply that your classes are either plain or old. The term POCO simply specifies that the POCO classes don't contain any reference that is specific to the entity framework or .NET framework.

Basically, POCO (Plain Old CLR Object) entities are existing domain objects within your application that you use with Entity Framework.

Eg:

```
public class Model
{
  public type PropertyName1 { get; set; }
  public type PropertyName2 { get; set; }
}
```

59. What is the proxy object?

An object that is created from a POCO class (model class) to support change tracking and lazy loading, is known as a proxy object.

There are some rules for creating a proxy class object:

- The class must be public and not sealed.
- Each navigation property must be marked as virtual.
- Each property must have a public getter and setter.
- Any collection navigation properties must be typed as ICollection <T>.

60. What are the various Entity States in EF?

Each and every entity has a state during its lifecycle which is defined by an enum (EntityState) that have the following values:

- Added
- Modified
- Deleted
- Unchanged
- Detached

61. What are various approaches in Code First for model designing?

- POCO model classes with data annotations
- POCO model classes with FluentAPI in DbContext

In Entity Framework Code First approach, our POCO classes are mapped to the database objects using a set of conventions defined in Entity Framework. If you do not want to follow these conventions while defining your POCO classes, or you want to change the way the conventions work then you can use the fluent API or data annotations to configure and to map your POCO classes to the database tables. There are two approaches, which you can use to define the model in EF Code First:

62. What C# Datatype is mapped with which Datatype in SQL Server?

C# Data Type	SQL server data type
int	int
string	nvarchar(Max)
decimal	decimal(18,2)
float	real
byte[]	varbinary(Max)
datetime	datetime
bool	bit
byte	tinyint
short	smallint
long	bigint
double	float
char	No mapping
sbyte	No mapping
object	No mapping

63. What is Code First Migrations in Entity Framework?

Code First Migrations allow you to create a new database or to update the existing database based on your model classes by using Package Manager Console exist within Visual Studio.

64. What is Migrations History Table?

In EF Core, Migrations history table (__MigrationHistory) is a part of the application database and used by Code First Migrations to store details about migrations applied to a database. This table is created when you apply the first migration to the database. This table stores metadata describing the schema version history of one or more EF Code First models within a given database.

65. How you apply code first migrations through code in EF Core?

Entity Framework Core supports “Migrate()” method, which can be called anywhere either in controller or in any other middleware when you want to perform code first migrations programmatically instead of applying migrations using “Update-Database” command.

Eg:

```
using (var context = new MyDbContext(...))
{
    context.Database.Migrate();
}
```

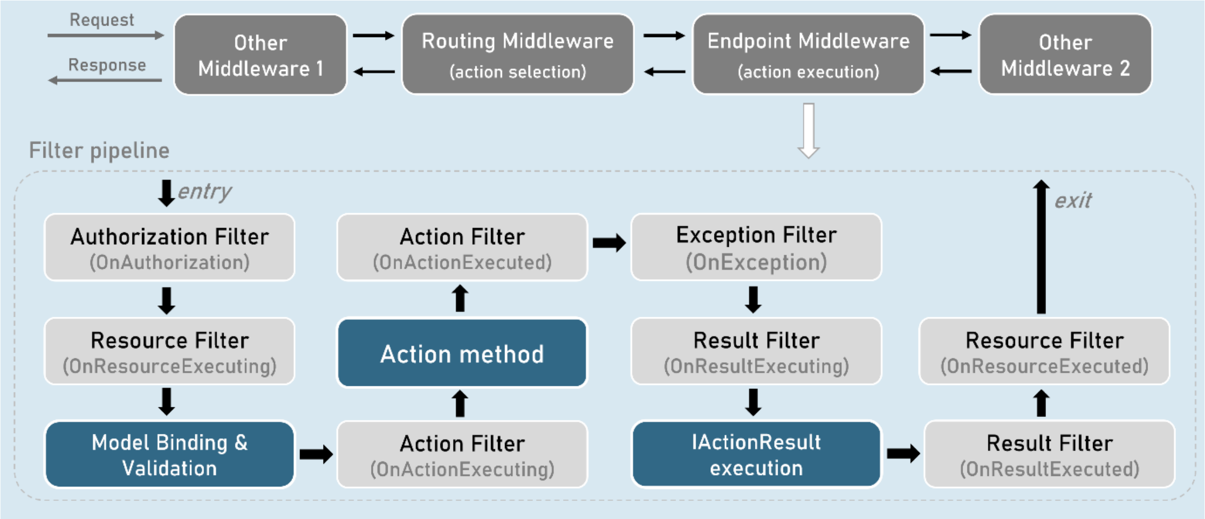
66. Explain different types of filters

Filters provide the capability to run the code before or after the specific stage in request processing pipeline, it could be either MVC app or Web API service. Filters performs the tasks like Authorization, Caching implementation, Exception handling etc. ASP.NET Core also provide the option to create custom filters. There are 5 types of filters supported in ASP.NET Core Web apps or services.

- **Authorization filters** run before all or first and determine the user is authorized or not.
- **Resource filters** are executed after authorization. OnResourceExecuting filter runs the code before rest of filter pipeline and OnResourceExecuted runs the code after rest of filter pipeline.
- **Action filters** run the code immediately before and after the action method execution. Action filters can change the arguments passed to method and can change returned result.
- **Exception filters** used to handle the exceptions globally before writing the response body
- **Result filters** allow to run the code just before or after successful execution of action results.

67. Explain request processing pipeline [or] filter pipeline in asp.net core?

The following diagram explains the complete request processing pipeline (includes with filter pipeline, which is the subset of request pipeline).



Request
Response
Other Middleware 1
Routing Middleware (action selection)
Endpoint Middleware (action execution)
Other Middleware 2
Filter pipeline
entry
exit
Authorization Filter (OnAuthorization)
Resource Filter (OnResourceExecuting)
Model Binding & Validation
Action Filter (OnActionExecuting)
Action method
Action Filter (OnActionExecuted)
Exception Filter (OnException)
Result Filter (OnResultExecuting)
IActionResult execution
Result Filter (OnResultExecuted)
Resource Filter (OnResourceExecuted)

68.How cookies work in asp.net core?

A cookie is a small amount of data that is persisted across requests and even sessions. Cookies store information about the user. The browser stores the cookies on the user's computer. Most browsers store the cookies as key-value pairs.

- At first, server sends a cookie (key/value pair) by using a response header called "Set-Cookie". Then the browser receives it and stores the cookie in the browser memory.
- For each subsequent request, the same cookie will be sent to the server with "Cookie" request header.

Write a cookie in ASP.NET Core:

```
Response.Cookies.Append(key, value);
```

Delete a cookie in ASP.NET Core:

```
Response.Cookies.Delete(somekey);
```

69.How do you short circuit the request in an action filter?

If I don't want the action method to be executed, I'll short-circuit the request in OnActionExecuting() method of Action filter, as it executes before execution of the action method.

```
public class MyActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
       if (condition)
       {
            //short-circuit the request
            context.Result = some_action_result;
       }
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
    }
}
```

70. How do you use dependency injection in action filter?

The ServiceFilterAttribute or TypeFilterAttribute helps me to have constructor injection (DI) in an action filter class.

If the filter class needs arguments to constructor, I would use TypeFilterAttribute; otherwise ServiceFilterAttribute.

```
public class FilterClassName : IActionFilter
{
   private readonly IService _service;

   public FilterClassName(IService service)
   {
       _service = service;
   }
   public void OnActionExecuting(ActionExecutingContext context)
   {
       //TO DO: before logic here
   }
   public void OnActionExecuted(ActionExecutedContext context)
   {
      //TO DO: after logic here
   }
}
```

71. How do you override order of filters?

I would implement an interface called IOrderFilter to have a property called “Order” that assigns to an int value.

The before methods execute in ascending order of “Order” property value; and the after methods execute in descending order.

```
public class FilterClassName : IActionFilter, IOrderedFilter
{
  public int Order { get; set; } //Defines sequence of execution
  public FilterClassName(int order)
  {
    Order = order;
  }
  public void OnActionExecuting(ActionExecutingContext context)
  {
    //TO DO: before logic here
  }
  public void OnActionExecuted(ActionExecutedContext context)
  {
    //TO DO: after logic here
  }
}
```

72. How will you add global filters?

Global filters once added to the “Filters” collection in the Startup (Program) class, will be applied to all action methods of all controllers in the application.

```
builder.Services.AddControllersWithViews(options => {
   options.Filters.Add<FilterClassName>(); //add by type
   //or
   options.Filters.Add(new FilterClassName()); //add filter instance
});
```

73. How do you handle errors in asp.net core application?

Errors arising in controller / view / result / middleware execution can be best handled with exception middleware.

```
public class ExceptionHandlingMiddleware : IMiddleware
{
  public async Task InvokeAsync(HttpContext context, RequestDelegate next)
  {
   try
   {
    await next(context);
   }
   catch (Exception ex)
   {
    //log errors
   }
  }
}

Apply the exception handling middleware before all custom middleware:

if (builder.Environment.IsDevelopment())
{
  app.UseDeveloperExceptionPage(); //enables developer exception page on exception
in “Development” environment
}
else
{
  app.UseMiddleware<ExceptionHandlingMiddleware>(); //adds
ExceptionHandlingMiddleware in case of other than “Development” environment
}
```

Status code pages

To enable custom error pages on specific status codes:

```
app.UseStatusCodePagesWithRedirects("url”); //it redirects to the specified url
when exception occurs with specific status code such as 400, 500, 404 etc.
```

74. How do you choose between Exception Middleware and Exception filter?

Though we can also create Exception filter by implementing IExceptionFilter interface, it’s not recommended unless you actually need it.

Exception filter handles unhandled exceptions that occur in controller creation, model binding, action filters or action methods.

Exception filter doesn't handle the unhandled exceptions that occur in authorization filters, resource filters, result filters or IActionResult execution.

Exception filters are recommended to be used only when you want a different error handling and generate different result for specific controllers; otherwise, ErrorHandlingMiddleware is recommended over Exception Filters.

75. Can you explain the concept of ASP.NET Core Identity and its role in building secure web applications?

ASP.NET Core Identity is a membership system that provides authentication and authorization functionalities for building secure web applications. It enables developers to manage user accounts, including user registration, login, password management, and user profile customization.

Additionally, ASP.NET Core Identity integrates with external login providers like Microsoft, Facebook, and Google, allowing users to log in using their existing accounts on these platforms. It also supports role-based authorization, allowing developers to control access to different parts of the application based on user roles.

76. What are the key components of ASP.NET Core Identity and how do they work together?

ASP.NET Core Identity consists of the following key components:

User: Represents a registered user in the system. It contains properties such as username, password hash, email, and other customizable user attributes.

Role: Defines a set of permissions or responsibilities. Users can be assigned to one or more roles, which determine their access rights within the application.

UserManager: Provides APIs for managing user-related operations, such as creating new users, deleting users, and resetting passwords.

SignInManager: Handles the authentication process, including user login, logout, and session management. It also supports features like two-factor authentication and external login providers.

Claims: Represent additional information about the user, such as user roles or custom attributes. Claims are used for authorization purposes and can be customized based on application requirements.

77. How can you customize ASP.NET Core Identity to meet specific application requirements?

ASP.NET Core Identity offers various customization points to adapt to specific application needs.

Here are a few customization options:

User and Role Models: You can extend the default User and Role models by adding additional properties or associating them with other entities in your application's domain.

Validation: You can modify the validation rules for user registration, password complexity, and other user-related operations by configuring the IdentityOptions.

Storage: ASP.NET Core Identity supports multiple data storage providers such as SQL Server, SQLite, and in-memory databases. You can choose the appropriate provider based on your application's requirements.

Authentication and Authorization: You can configure external login providers, implement custom authentication schemes, and define fine-grained authorization policies using the built-in policy-based authorization system.

Views and UI: ASP.NET Core Identity provides default views and UI components for user registration, login, and account management. You can customize these views or create your own to match the application's visual style.

78. How does ASP.NET Core Identity handle authentication and authorization in a web application?

ASP.NET Core Identity uses authentication and authorization middleware to handle these processes. When a user logs in, the SignInManager validates the user's credentials and creates an authentication ticket (token), which is stored as a cookie or other authentication token. Subsequent requests from the user contain this ticket, allowing the application to identify the user.

For authorization, ASP.NET Core Identity provides a flexible role-based system. Developers can assign users to specific roles, and then use the Authorize attribute or the policy-based authorization system to protect actions and resources. The framework checks the user's assigned roles and authorizes or denies access based on the defined rules.

79. How can you handle user registration and password management in ASP.NET Core Identity?

ASP.NET Core Identity provides APIs and features to handle user registration and password management. Here's how it can be done:

User Registration: To enable user registration, you can create a registration form that collects user information such as username, email, and password. In the registration action, you use the UserManager to create a new user by providing the user details. The UserManager takes care of hashing the password and storing the user in the underlying user store.

Password Management: ASP.NET Core Identity provides various features for password management, including password hashing, password reset, and password change. When a user registers or changes their password, the password is automatically hashed and stored securely.

Password Policies: ASP.NET Core Identity allows you to configure password policies to enforce password complexity requirements. You can customize the password length, required characters, and other criteria by configuring the IdentityOptions in the application's startup code.

80. How can you implement role-based authorization in ASP.NET Core Identity?

ASP.NET Core Identity provides built-in support for role-based authorization. Here's how you can implement it:

Define Roles: First, you need to define roles that represent different levels of access or responsibilities within your application. You can create roles using the RoleManager provided by ASP.NET Core Identity.

Assign Roles: Once roles are defined, you can assign roles to individual users or groups of users. This can be done using the UserManager by associating the appropriate roles with user accounts.

Protect Resources: In your application's controllers or actions, you can use the Authorize attribute with role-based policies to restrict access to specific resources. For example, you can decorate an action with [Authorize(Roles = "Admin")] to allow only users in the "Admin" role to access it.

Check Role-Based Authorization: Within the controller or action, you can use the User property to check the currently authenticated user's assigned roles and perform additional authorization logic based on the user's role membership.

By implementing role-based authorization, you can control access to various parts of your application based on the user's assigned roles

.

81. What are some common security considerations when using ASP.NET Core Identity?

When working with ASP.NET Core Identity, there are several important security considerations to keep in mind:

Password Storage: Ensure that passwords are securely stored by using proper hashing and salting techniques. ASP.NET Core Identity handles this automatically, but it's essential to choose a secure password hashing algorithm and keep passwords hashed and protected from unauthorized access.

Authentication Security: Implement secure authentication practices such as enforcing strong password policies, enabling two-factor authentication for sensitive accounts, and using secure protocols (HTTPS) to protect authentication credentials during transit.

Authorization and Role-Based Access Control (RBAC): Carefully define and manage roles and permissions to ensure that only authorized users have access to specific resources and actions within your application. Avoid granting excessive privileges to user roles and regularly review and update role assignments as needed.

Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF): Implement measures to prevent and mitigate common web vulnerabilities like XSS and CSRF attacks. Use appropriate input validation and output encoding techniques to protect against malicious user input and ensure that request forgery prevention mechanisms are in place.

Account Lockout and Brute Force Protection: Protect user accounts from brute force attacks by implementing account lockout policies and rate limiting mechanisms. Set limits on the number of failed login attempts and implement measures to detect and respond to suspicious or malicious activities.

User Input Validation: Always validate and sanitize user input to prevent security vulnerabilities such as SQL injection and other forms of code injection attacks. Use proper input validation techniques and parameterized queries when interacting with the database.

Error Handling and Logging: Implement secure error handling and logging practices to avoid exposing sensitive information or system details in error messages. Ensure that exception details are properly handled and logged without revealing potentially sensitive data.

Regular Updates and Patches: Keep your ASP.NET Core Identity framework and its dependencies up to date by regularly applying security patches and updates. Stay informed about any security vulnerabilities or best practices related to the framework and follow recommended guidelines for secure application development.

By considering these security aspects and implementing the necessary measures, you can enhance the overall security of your ASP.NET Core Identity-based application.

82. What are different managers in ASP.NET Core?

In ASP.NET Core, managers are classes that provide APIs and functionalities for managing various aspects of an application. In the context of ASP.NET Core Identity, the following managers are commonly used:

UserManager: The UserManager class provides operations for managing user-related functionalities, such as creating new users, updating user information, validating user credentials, and managing user roles and claims.

RoleManager: The RoleManager class is responsible for managing roles within the application. It allows you to create new roles, update role information, assign or remove roles from users, and perform role-related operations.

SignInManager: The SignInManager class handles user authentication and sign-in processes. It provides methods for signing in a user, signing out a user, and managing user sessions. Additionally, it supports features like two-factor authentication and external login providers.

UserStore and RoleStore: While not managers themselves, the UserStore and RoleStore classes are implementations of the store interfaces used by the UserManager and RoleManager. These store classes handle data persistence and provide methods for interacting with the underlying data storage, such as a database or other data repositories.

83. What architecture is used in ASP.NET Core Identity (with store and managers)?

ASP.NET Core Identity follows a layered architecture with separation of concerns. The architecture consists of the following components:

User Interface (UI) Layer: This layer represents the user-facing part of the application, which includes views, controllers, and client-side code. The UI layer interacts with the application's business logic layer and utilizes ASP.NET Core Identity functionalities through the managers and services.

Business Logic Layer: The business logic layer contains the application's core logic and is responsible for implementing application-specific rules and workflows. It uses the managers provided by ASP.NET Core Identity, such as UserManager and RoleManager, to perform user and role-related operations.

Data Access Layer: The data access layer is responsible for interacting with the underlying data storage, such as a database. ASP.NET Core Identity uses the concept of stores, such as UserStore and RoleStore, which encapsulate the operations for reading and writing data related to users and roles. These stores abstract away the specific data storage implementation and provide a consistent interface for the managers to access and manipulate the data.

Identity Services: ASP.NET Core Identity provides a set of services that are registered in the application's dependency injection container. These services include UserManager, RoleManager, SignInManager, and other related services. The managers and services encapsulate the core functionality of ASP.NET Core Identity and are consumed by the business logic layer and the UI layer.

Overall, the architecture of ASP.NET Core Identity follows a layered approach with clear separation of concerns, allowing for modular and extensible application development.

84. What is Cross-Site Request Forgery (XSRF) and how does it impact web applications?

Cross-Site Request Forgery (XSRF), also known as CSRF or session riding, is a web security vulnerability. It occurs when an attacker tricks a user's browser into performing an unwanted action on a target website on behalf of the user. This can lead to unauthorized actions, such as making changes, submitting forms, or performing transactions, without the user's knowledge or consent. XSRF attacks exploit the trust between a website and a user's authenticated session, potentially resulting in data breaches, unauthorized access, or unintended modifications.

85. How does ASP.NET Core protect against XSRF attacks?

ASP.NET Core provides built-in protection against XSRF attacks through anti-forgery tokens. Here's how it works:

Anti-Forgery Tokens: ASP.NET Core generates unique anti-forgery tokens for each user session. These tokens are included in HTML forms or Ajax requests as hidden fields or headers.

Token Validation: When a form or Ajax request is submitted, ASP.NET Core automatically validates the anti-forgery token. It compares the token sent by the client with the token stored in the server's session or cookie. If the tokens match, the request is considered valid; otherwise, it is rejected as a potential XSRF attack.

Automatic Integration: ASP.NET Core automatically adds anti-forgery tokens to forms generated by Razor views, making it easier to implement XSRF protection without explicit configuration.

86. How can you implement XSRF protection in ASP.NET Core manually?

In ASP.NET Core, you can implement XSRF protection manually by following these steps:

Generating Anti-Forgery Tokens: Generate and include anti-forgery tokens in your HTML forms or Ajax requests. This can be done using the @Html.AntiForgeryToken() helper in Razor views or by manually adding the token as a hidden field or header.

Validating Anti-Forgery Tokens: In your server-side code, validate the anti-forgery token for each submitted form or Ajax request. Use the [ValidateAntiForgeryToken] attribute on controller actions or manually validate the token using the ValidateAntiForgeryToken attribute or Request.Form property.

Configuring Anti-Forgery Options: Customize the anti-forgery options in the ASP.NET Core configuration. You can set properties like cookie name, header name, and token lifespan to match your application's requirements.

87. What is ASP.NET Web API?

ASP.NET Web API is a framework provided by Microsoft for building HTTP services that can be consumed by various clients, such as web browsers, mobile applications, and desktop applications. It allows developers to create RESTful APIs using the .NET platform.

88. How do you define a Web API controller in ASP.NET Web API?

To define a Web API controller, you need to create a new class that inherits from the ApiController class. This class represents an HTTP service and contains action methods that handle different HTTP requests.

89. Explain the basic syntax of a Web API controller.

A Web API controller consists of a class that inherits from the ApiController class and contains action methods. Action methods are public methods decorated with attributes such as [HttpGet], [HttpPost], [HttpPut], or [HttpDelete], depending on the HTTP verb they handle. These methods are responsible for processing incoming requests and returning responses.

90. What are Action Results in ASP.NET Web API?

Action Results in ASP.NET Web API represent the result returned by an action method. They encapsulate the data that needs to be sent back to the client. Action Results derive from the IActionResult interface.

91. What is the difference between IActionResult and ActionResult<T>?

IActionResult is an interface that represents the result of an action method, whereas ActionResult<T> is a generic class that derives from IActionResult and is used when you want to return a specific type of data as the result of an action method. For example, ActionResult<string> can be used to return a string result.

92. Explain the usage of the ProblemDetails class in ASP.NET Web API.

The ProblemDetails class is a part of the ASP.NET Core framework and is used to provide consistent error responses from a Web API. It allows you to return structured error information in the form of a JSON response, including details like the error message, status code, and additional properties.

93. Can you create a custom base class for Web API controllers? If so, how?

Yes, you can create a custom base class for Web API controllers. To do this, you need to create a new class that derives from the ControllerBase class. This custom base class can contain common functionality, utility methods, or custom behavior that you want to share across multiple Web API controllers.

94. How do you integrate Entity Framework Core with ASP.NET Web API?

To use Entity Framework Core (EF Core) with ASP.NET Web API, you need to follow these steps:

1. Install the required NuGet packages for EF Core.
2. Define your data model classes as entities.
3. Create a DbContext class that represents the database context.
4. Configure the DbContext and entity relationships.
5. Inject the DbContext into your Web API controllers or services using dependency injection.
6. Use the DbContext in your controllers or services to perform database operations such as querying, inserting, updating, or deleting data.

95. How can you return data from EF Core queries in Web API controller actions?

You can return data from EF Core queries in Web API controller actions by using the Ok method of the ApiController. For example, you can use return Ok(data) to return an HTTP 200 OK response along with the data retrieved from the database.

96. What is the purpose of the ActionResult class in Web API?

The ActionResult class is the base class for all action results in Web API. It provides a convenient way to create and return different types of HTTP responses. It encapsulates both the data and the HTTP status code that should be returned to the client.

97. What is Swagger/OpenAPI?

Swagger, now known as OpenAPI, is a specification and set of tools for designing, building, documenting, and consuming RESTful APIs. It provides a standardized way to describe APIs using JSON or YAML, enabling developers to understand and interact with APIs more easily.

98. What is the purpose of Swagger/OpenAPI?

The main purpose of Swagger/OpenAPI is to enhance the development process of RESTful APIs by providing a machine-readable contract that describes the API's structure, endpoints, request/response formats, authentication requirements, and more. This contract can be used to generate client SDKs, server stubs, and interactive API documentation, simplifying API consumption and integration.

99. How can you enable Swagger in ASP.NET Core?

To enable Swagger in ASP.NET Core, you need to follow these steps:

1. Install the "Swashbuckle.AspNetCore" NuGet package.

2. In the Startup.cs file, add the following code within the ConfigureServices method:

```
builder.Services.AddSwaggerGen(c =>
{
  c.SwaggerDoc("v1", new OpenApiInfo { Title = "Your API Name", Version = "v1" });
});
```

3. In the Configure method, add the following code:

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
  c.SwaggerEndpoint("/swagger/v1/swagger.json", "Your API Name v1");
});
```

4. Run your application and navigate to the Swagger UI at "/swagger" to view and interact with the API documentation.

100. How can you version your API using Swagger in ASP.NET Core?

To version your API using Swagger in ASP.NET Core, you can follow one of the common versioning approaches like URI versioning or query string versioning. Once you have implemented the versioning mechanism, you can generate separate Swagger documents for each API version. Here's an example of using URI versioning:

1. Install the "Microsoft.AspNetCore.Mvc.Versioning" NuGet package.

2. Configure the API versioning in the Startup.cs file within the ConfigureServices method:

```
builder.Services.AddApiVersioning(options =>
{
  options.DefaultApiVersion = new ApiVersion(1, 0);
  options.AssumeDefaultVersionWhenUnspecified = true;
});
```

3. Modify the Swagger configuration in ConfigureServices to generate versioned Swagger documents:

```
builder.Services.AddSwaggerGen(c =>
{
  c.SwaggerDoc("v1", new OpenApiInfo { Title = "Your API Name", Version = "v1" });
  c.SwaggerDoc("v2", new OpenApiInfo { Title = "Your API Name", Version = "v2" });
});
```

4. Configure Swagger UI in the Configure method:

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
  c.SwaggerEndpoint("/swagger/v1/swagger.json", "Your API Name v1");
  c.SwaggerEndpoint("/swagger/v2/swagger.json", "Your API Name v2");
});
```

Now, you can access the different API versions in the Swagger UI.

101. What is content negotiation in the context of ASP.NET Core Web APIs?

Content negotiation in the context of ASP.NET Core Web APIs refers to the process of selecting the appropriate response format (such as JSON, XML, or others) based on the client's preferences and the available options from the server. The negotiation process involves matching the client's requested media types (specified in the Accept header) with the server's supported media types (specified in the Produces attribute or configuration).

102. How can you implement content negotiation in ASP.NET Core Web APIs?

Content negotiation in Web APIs can be implemented using the Produces attribute or the AddControllers method configuration.

Here's how you can do it:

1. Using the Produces attribute:

```
[ApiController]
[Route("api/[controller]")]
public class MyController : ControllerBase
{
  [HttpGet]
  [Produces("application/json", "application/xml")]
  public IActionResult Get()
  {
    // Action implementation...
  }
}
```

In this example, the Produces attribute is applied to the Get action method of the MyController API. It specifies that the action can produce responses in either JSON or XML format. The appropriate format is selected based on the client's Accept header.

2. Using the AddControllers method configuration in the Startup.cs file:

```
builder.Services.AddControllers(options =>
{
  options.OutputFormatters.Add(new XmlSerializerOutputFormatter());
})
.AddXmlDataContractSerializerFormatters();
```

In this example, the AddControllers method is used to configure the content negotiation for the Web API. The ReturnHttpNotAcceptable property is set to true to return a 406 Not Acceptable status code when the requested media type is not supported. The XmlSerializerOutputFormatter is added to the OutputFormatters collection to support XML serialization. Additionally, the AddXmlDataContractSerializerFormatters method is called to enable XML serialization using the DataContractSerializer.

What is CORS (Cross-Origin Resource Sharing)?

Cross-Origin Resource Sharing (CORS) is a security mechanism implemented in web browsers that allows restricted resources on a web page to be requested from a different domain. The same-origin policy enforced by web browsers restricts web pages from making requests to a different domain. However, CORS provides a way to relax this restriction and enable controlled access to resources from different origins.

When a web page makes a cross-origin request, the browser sends an HTTP OPTIONS preflight request to the server to check if the server allows the actual request. The server responds with CORS headers that specify the allowed origins, methods, headers, and other parameters for incoming requests. If the server allows the request based on the specified CORS policies, the browser allows the cross-origin request and the requested resource is accessible.

103. Why is CORS necessary?

CORS (Cross-Origin Resource Sharing) is necessary because web browsers enforce a security policy known as the same-origin policy. The same-origin policy restricts web pages from making requests to a different domain, which helps protect users from potential security vulnerabilities.

However, there are legitimate scenarios where web applications need to access resources or consume APIs from different domains. This is where CORS comes into play. CORS allows controlled and secure access to resources from different origins while still maintaining the fundamental security provided by the same-origin policy.

Here are a few reasons why CORS is necessary:

1. Cross-Domain Communication: Many modern web applications are built using a client-server architecture, where the client-side code (such as JavaScript) runs in a web browser and needs to interact with APIs or services hosted on different domains. CORS enables the client-side code to make cross-domain requests to retrieve data or perform actions on behalf of the user.

2. API Consumption: Web applications often consume data or services from third-party APIs. These APIs might be hosted on different domains or subdomains. CORS allows the web application to make requests to these APIs and retrieve the data it needs to display or process.

3. Microservices and Distributed Systems: In complex systems composed of multiple microservices or distributed components, each component may have its own domain or endpoint. CORS allows the components to communicate and exchange data securely across different domains, enabling a scalable and modular architecture.

4. Third-Party Integration: Web applications may integrate with external services or embed content from other websites. CORS enables these integrations by allowing the web application to make cross-origin requests to fetch data, embed iframes, or interact with the integrated services.

By enabling CORS, web developers can selectively specify which domains or origins are allowed to access their resources or APIs. CORS provides a mechanism to define fine-grained policies that determine the origins, HTTP methods, headers, and other parameters that are permitted for cross-origin requests, ensuring controlled access and maintaining security while facilitating seamless integration between different web resources.

104. How CORS works internally?

Internally, CORS (Cross-Origin Resource Sharing) works as a series of steps involving the web browser and the server to facilitate controlled cross-origin requests. Here's a high-level overview of how CORS works:

1. Origin Determination:

When a web page makes a request to a different domain, the browser considers the origin of the request. The origin consists of the combination of the protocol (e.g., HTTP, HTTPS), domain (e.g., example.com), and port number (if specified). The origin is derived from the URL of the web page making the request.

2. Preflight Request:

Before making certain types of cross-origin requests (e.g., requests with HTTP methods other than GET, POST, or HEAD, or requests with custom headers), the browser sends an HTTP OPTIONS preflight request to the server. This preflight request includes additional headers, such as "Access-Control-Request-Method" and "Access-Control-Request-Headers", to gather information about the intended cross-origin request.

3. Server Response and CORS Headers:

The server receives the preflight request and generates a response. It determines whether the actual request should be allowed based on its CORS policies. The server includes specific CORS response headers in the preflight response to inform the browser of its decision. These headers include:

1. "Access-Control-Allow-Origin": Specifies the allowed origin(s) that can access the resource.
2. "Access-Control-Allow-Methods": Specifies the allowed HTTP methods for the actual request.
3. "Access-Control-Allow-Headers": Specifies the allowed request headers for the actual request.
4. "Access-Control-Allow-Credentials": Indicates whether the actual request can include credentials like cookies or authorization headers.

4. Browser Evaluation:

The browser receives the preflight response and evaluates the CORS headers. It checks if the origin of the web page making the request is allowed by the server's CORS policy. It also verifies if the requested method, headers, and credentials (if any) are permitted. If the browser determines that the request is allowed based on the CORS headers, it proceeds with the actual cross-origin request.

5. Actual Request:

If the browser confirms that the actual request is permitted, it sends the actual cross-origin request to the server, including the necessary headers and payload (if any). The server processes the request and generates the corresponding response.

6. Response Handling:

The browser receives the response from the server. The response may include additional CORS headers indicating the server's policies and restrictions. The browser then allows or restricts access to the response based on the evaluation of these headers.

By following this process, CORS ensures controlled and secure cross-origin communication between web browsers and servers. It allows servers to define policies that dictate which origins, methods, headers, and credentials are allowed to access their resources, thereby protecting users from potential security threats while enabling legitimate cross-origin interactions.

105. How do you enable CORS in an ASP.NET Core Web API?

To enable CORS in an ASP.NET Core Web API, you can follow these steps:

1. Add CORS service to the service collection

```
builder.Services.AddCors(options => {
  options.AddDefaultPolicy(builder =>
  {
    builder.WithOrigins("http://localhost:4200");
  });
});
```

2. Add CORS middleware to the middleware chain / request pipeline.

```
app.UseCors();
```

106. What are CORS policies?

CORS policies define the rules and restrictions for allowing or restricting cross-origin requests to an API. CORS policies are implemented on the server side and specify which origins, methods, headers, and other parameters are allowed to access the API resources.

CORS policies help control and secure access to resources exposed by an API, ensuring that only trusted and authorized sources can make cross-origin requests.

By defining CORS policies, API developers can specify the level of flexibility and control they want to allow when it comes to accessing their resources from different origins.

Here are some key elements that CORS policies can define:

1. Allowed Origins:

CORS policies specify the origins or domains that are allowed to access the API resources. An origin consists of the combination of the protocol (e.g., HTTP, HTTPS), domain (e.g., example.com), and port number (if specified). Developers can specify one or more allowed origins or use the wildcard "*" to allow requests from any origin.

2. Allowed Methods:

CORS policies define the HTTP methods (such as GET, POST, PUT, DELETE, etc.) that are allowed for cross-origin requests. This helps restrict unauthorized methods from being used for accessing or modifying resources.

3. Allowed Headers:

CORS policies specify the headers that are allowed in cross-origin requests. This allows API developers to control which request headers are accepted and processed by the server.

4. Exposed Headers:

CORS policies can define the headers that are exposed in the response to cross-origin requests. Exposed headers are additional response headers that the server allows the browser to access and read in the client-side code.

5. Credentials:

CORS policies can indicate whether cross-origin requests can include credentials, such as cookies or authorization headers. By default, browsers do not send credentials in cross-origin requests, but CORS policies can allow or restrict this behavior.

6. Preflight Requests:

CORS policies determine whether preflight requests (HTTP OPTIONS requests) are required for certain cross-origin requests. Preflight requests are used to check if the actual request is allowed by the server based on its CORS policies.

By defining and configuring CORS policies, API developers can ensure controlled access to their resources, protect against unauthorized cross-origin requests, and maintain the security and integrity of their API endpoints.

107. How can you create and apply multiple CORS policies in ASP.NET Core Web API?

In ASP.NET Core Web API, you can specify multiple CORS policies by defining and registering multiple instances of the CorsPolicy class. Each CORS policy can have its own set of allowed origins, methods, headers, and other parameters.

Steps to create multiple CORS policies in an ASP.NET Core Web API project:

1. Open the Program.cs file in your ASP.NET Core Web API project.

2. Add the CORS services by calling the AddCors method on the IServiceCollection object. This registers the CORS services in the dependency injection container.

```
builder.Services.AddCors(options =>
{
  options.AddPolicy("Policy1", builder =>
  {
    builder.WithOrigins("http://example1.com")
      .AllowAnyMethod()
      .AllowAnyHeader();
  });

  options.AddPolicy("Policy2", builder =>
  {
    builder.WithOrigins("http://example2.com")
      .AllowMethods("GET")
      .AllowHeaders("Authorization");
  });

  // Add more policies as needed...
});
```

3. Use the UseCors method with the name of the desired CORS policy to apply the policy to the request pipeline. You can specify the CORS policy at the global level or apply it to specific routes or controllers.

```
app.UseCors("Policy1");
```

Alternatively, you can apply the CORS policy at the controller or action level by using the [EnableCors] attribute and specifying the policy name as an argument.

```
[EnableCors("Policy2")]
public class MyController : ControllerBase
{
  // Controller code...
}
```

By following these steps, you can specify and apply multiple CORS policies in your ASP.NET Core Web API. Each policy can have its own set of allowed origins, methods, headers, and other parameters, providing granular control over cross-origin requests in your API.

108. What is JWT (JSON Web Token)?

JWT (JSON Web Token) is a compact and self-contained way to transmit information between parties as a JSON object. It is a standardized format for token-based authentication and authorization in web applications.

A JWT consists of three parts separated by dots: the header, the payload, and the signature.

Header: The header contains metadata about the token, such as the type of token and the algorithm used for signing the token. It is base64Url encoded.

Payload: The payload contains the claims or statements about the user or entity, such as the user's identity, roles, and additional data. Claims can include standard claims defined by the JWT specification or custom claims specific to the application. The payload is also base64Url encoded.

Signature: The signature is created by combining the base64Url encoded header and payload, along with a secret key known only to the server. It is used to verify the integrity of the token and ensure that it has not been tampered with.

JWTs are commonly used for authentication and authorization in web applications. When a user logs in or authenticates, the server generates a JWT and sends it back to the client. The client includes the JWT in subsequent requests as an authorization mechanism. The server can then verify the token's signature and extract the claims to determine the identity and permissions of the user.

One of the advantages of JWTs is that they are self-contained, meaning the server can validate and extract the necessary information without needing to store the token on the server-side. This makes JWTs suitable for stateless authentication systems and distributed environments.

109. How can you configure JWT authentication in ASP.NET Core Web API?

To configure JWT authentication in ASP.NET Core Web API with top-level statements, modify the Program.cs file as follows:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;

// Add the following code after builder.Services.AddIdentity()
var jwtSettings = builder.Configuration.GetSection("JwtSettings");
var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtSettings["Key"]));
var tokenValidationParameters = new TokenValidationParameters
{
 ValidateIssuer = true,
 ValidateAudience = true,
 ValidateIssuerSigningKey = true,
 ValidIssuer = jwtSettings["Issuer"],
 ValidAudience = jwtSettings["Audience"],
 IssuerSigningKey = key
};

builder.Services.AddAuthentication(options =>
{
 options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
 options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
 options.TokenValidationParameters = tokenValidationParameters;
});
```

This code configures JWT authentication by using the JwtBearerDefaults.AuthenticationScheme and sets the token validation parameters, including the issuer, audience, and signing key.

110. How can you generate a JWT token in ASP.NET Core Web API?

To generate a JWT token in ASP.NET Core Web API, you can use the System.IdentityModel.Tokens.Jwt package. Modify the Program.cs file as follows:

```
using System.IdentityModel.Tokens.Jwt;

// Inside a method or controller action
var tokenHandler = new JwtSecurityTokenHandler();
var key = Encoding.UTF8.GetBytes(jwtSettings["Key"]);
var tokenDescriptor = new SecurityTokenDescriptor
{
 Subject = new ClaimsIdentity(new[]
 {
  new Claim(ClaimTypes.Name, "John Doe"),
  // Add additional claims as needed
 }),
 Expires = DateTime.UtcNow.AddHours(1),
 SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key),
SecurityAlgorithms.HmacSha256Signature)
};

var token = tokenHandler.CreateToken(tokenDescriptor);
var tokenString = tokenHandler.WriteToken(token);
```

This code creates a new instance of JwtSecurityTokenHandler and generates a JWT token by specifying the claims, expiration time, signing credentials, and other necessary information.

111. How can you authorize API endpoints using JWT tokens?

To authorize API endpoints using JWT tokens in ASP.NET Core Web API, you can use the [Authorize] attribute on API endpoints or controllers. Modify the Program.cs file as follows:

```
// Add the following code after app.UseAuthorization()
app.MapControllers().RequireAuthorization();
```

This code ensures that all API endpoints require authorization. You can also apply the [Authorize] attribute selectively on individual controllers or action methods for more granular control.

Remember to update the appsettings.json file with the required JWT settings, such as the issuer, audience, and signing key.

112. How can you implement role-based authorization using Identity in ASP.NET Core Web API?

To implement role-based authorization using Identity in ASP.NET Core Web API, follow these steps:

Define the roles in the database using the RoleManager provided by Identity.

Assign roles to users during user registration or through an admin interface.

Add the `[Authorize(Roles = "RoleName")]` attribute to API endpoints or controllers that should be accessible only to users in specific roles.

In the Program.cs file, configure the authorization policy by adding the following code:

```
builder.Services.AddAuthorization(options =>
{
 options.AddPolicy("RoleNamePolicy", policy =>
  policy.RequireRole("RoleName"));
});
```

Apply the policy to the desired endpoints by using the [Authorize(Policy = "RoleNamePolicy")] attribute.

113. What is refresh token and what is its purpose in asp.net core?

A refresh token is a type of token used in token-based authentication systems, to obtain a new access (JWT) token without requiring the user to reauthenticate. It is an additional token issued alongside the access token and serves a specific purpose:

Longer-lived sessions: Refresh tokens have a longer expiration time compared to access tokens. While access (JWT) tokens are typically short-lived and have a limited validity period, refresh tokens are designed to be used for a longer duration, often lasting days, weeks, or even months.

Renewing access tokens: When an access token expires, the refresh token can be used to request a new access token from the authentication server without prompting the user for their credentials again. This allows the user to maintain their session and continue accessing protected resources seamlessly.

Revoking access: Refresh tokens can be revoked independently of access tokens. This provides better control and security if a refresh token is suspected to be compromised or if a user wants to revoke access to their account.

Reduced reliance on authentication server: By using refresh tokens, client applications can minimize the frequency of interactions with the authentication server for token renewal. This reduces the load on the authentication server and enhances the overall performance of the system.

Overall, the purpose of a refresh token is to provide a long-lived, secure mechanism for obtaining new access tokens without requiring the user to reauthenticate. It improves user experience, security, and reduces the need for frequent authentication requests, making it a valuable component of token-based authentication systems.

113. How to implement refresh tokens with JWT tokens in asp.net core?

To implement refresh token, you can modify the ApplicationUser model to store refresh token and its details inside the ApplicationUser model itself (in the AspNetUsers tabe).

Steps:

1. Modify the ApplicationUser model: Add properties for the refresh token and its details inside the ApplicationUser class.

```
public class ApplicationUser : IdentityUser
{
  // Other user properties

  public string RefreshToken { get; set; }
  public DateTime RefreshTokenExpiration { get; set; }
}
```

2. Update the registration/login process: During the registration or login process, generate a refresh token and assign it to the RefreshToken property of the ApplicationUser model. Set the RefreshTokenExpiration property to the appropriate expiration date.

3. Store and retrieve the ApplicationUser: When saving or retrieving the ApplicationUser from the database, ensure that the refresh token and its details are persisted along with the user data.

4. Token generation and validation: During the generation of the JWT access token, include the refresh token and its expiration details in the token payload. When validating the JWT, make sure to also validate the refresh token details and expiration.

5. Token refresh: When a client sends a request with an expired access token, check if the associated ApplicationUser has a valid refresh token. If the refresh token is valid, generate a new access token and update the RefreshToken and RefreshTokenExpiration properties of the ApplicationUser model. Return the new access token to the client.

By adding the refresh token and its details directly inside the ApplicationUser model, you can maintain the token information alongside the user data, simplifying the process of managing and associating refresh tokens with users.

114. What is ASP.NET Core Minimal API, and how does it differ from traditional Web API?

ASP.NET Core Minimal API is a lightweight and simplified approach to building web APIs in ASP.NET Core. It aims to reduce the amount of boilerplate code required to create APIs by leveraging the new minimalistic programming model. Unlike traditional Web API, which typically involves multiple files and configurations, Minimal API allows developers to define the API routes and handlers in a single file, making it easier to understand and maintain.

115. How to implement CRUD operations using Asp.Net Core Minimal API? Explain with sample code.

To create minimal API CRUD operations in ASP.NET Core 6, follow these steps:

1. Create a new ASP.NET Core 6 project in Visual Studio:

- Open Visual Studio.
- Click on "Create a new project."
- Select "ASP.NET Core Empty" template.
- Configure other project details and click on "Create."

2. Define your data model:

Create a class that represents the entity you want to perform CRUD operations on. For example, a "Product" class with properties like Id, Name, Price, etc.

3. Implement the API endpoints:

Open the project's Program.cs file.

Add the necessary services for your application. For example, you'll typically need to add a database context using builder.Services.AddDbContext.

Here's an example of creating minimal API CRUD operations for a "Product" entity:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services
builder.Services.AddDbContext<ApplicationDbContext>(options =>
{

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"
));
});

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
 c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API", Version = "v1" });
});

// Create database and migrate
using (var app = builder.Build())
{
using (var scope = builder.Services.CreateScope())
{
  var services = scope.ServiceProvider;
  var dbContext = services.GetRequiredService<ApplicationDbContext>();
  dbContext.Database.Migrate();
}

// Enable middleware to serve generated Swagger as a JSON endpoint.
app.UseSwagger();

// Specify the Swagger JSON endpoint.
app.UseSwaggerUI(c =>
{
 c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
 c.RoutePrefix = string.Empty;
});
```

4. Define API endpoints for GET, POST, PUT and DELETE operations

```
app.MapGet("/api/products", async (ApplicationDbContext dbContext) =>
{
  var products = await dbContext.Products.ToListAsync();
  return Results.Ok(products);
});

app.MapGet("/api/products/{id}", async (int id, ApplicationDbContext dbContext) =>
{
  var product = await dbContext.Products.FindAsync(id);
  if (product != null)
  {
    return Results.Ok(product);
  }
  else
  {
    return Results.NotFound();
  }
});

app.MapPost("/api/products", async (Product product, ApplicationDbContext
dbContext) =>
{
  dbContext.Products.Add(product);
  await dbContext.SaveChangesAsync();
  return Results.Created($"/api/products/{product.Id}", product);
});

app.MapPut("/api/products/{id}", async (int id, Product updatedProduct,
ApplicationDbContext dbContext) =>
{
  var product = await dbContext.Products.FindAsync(id);
  if (product != null)
  {
    product.Name = updatedProduct.Name;
    product.Price = updatedProduct.Price;
    // Update other properties as needed

    await dbContext.SaveChangesAsync();
    return Results.Ok(product);
  }
  else
  {
  return Results.NotFound();
  }
});

app.MapDelete("/api/products/{id}", async (int id, ApplicationDbContext
dbContext) =>
{
  var product = await dbContext.Products.FindAsync(id);
  if (product != null)
  {
    dbContext.Products.Remove(product);
    await dbContext.SaveChangesAsync();
    return Results.NoContent();
  }
  else
  {
    return Results.NotFound();
  }
});

app.Run();
}
```

In this example, the endpoints for listing products, retrieving a product by id, creating a product, updating a product, and deleting a product are defined using app.MapGet, app.MapPost, app.MapPut, and app.MapDelete respectively. The endpoints use the ApplicationDbContext to access the database and perform CRUD operations. Additionally, Swagger is configured to generate documentation for the API.

116. How do you handle request routing and parameter binding in ASP.NET Core Minimal API?

In ASP.NET Core Minimal API, request routing and parameter binding can be handled using the MapMethods extension method within the Configure method. The MapMethods method allows you to specify the HTTP verb, route pattern, and handler function for each API endpoint.

The MapMethod can be MapGet, MapPost, MapPut and MapDelete.

For example, to handle a GET request with a route parameter, you can define a route pattern with a placeholder for the parameter and bind it to the handler function using lambda syntax:

```
app.MapGet("/api/users/{id}", new[] { "GET" }, (int id) =>
{
  // Handle the GET request with the specified route parameter
  // Access the 'id' parameter in the handler function
  // Perform necessary operations and return a response
});
```

ASP.NET Core Minimal API uses a built-in route parameter binding feature that automatically maps the route parameter to the corresponding parameter in the handler function. You can use various parameter types (such as int, string, DateTime, etc.) depending on the type of data expected.

117. How do you perform model validation in ASP.NET Core Minimal API?

To perform model validation in ASP.NET Core Minimal API, you can utilize the model binding and validation features provided by the framework. Follow these steps:

Define a request model class with the desired properties and validation attributes. For example:

```
public class UserRequest
{
  [Required]
  public string Name { get; set; }

  [Range(18, 99)]
  public int Age { get; set; }
}
```

In the handler function for the API endpoint, add a parameter of the defined request model type. ASP.NET Core will automatically bind and validate the request data against the model.

Check the ModelState.IsValid property to determine if the model validation passed or not. If the model is invalid, you can return a BadRequest response with the validation errors.

Here's an example of a POST endpoint with model validation:

```
app.MapPost("/api/users", new[] { "POST" }, (UserRequest user) =>
{
  if (!ModelState.IsValid)
  {
    return Results.BadRequest(ModelState);
  }

  // Process the valid user request and return a response
});
```

ASP.NET Core will automatically perform validation based on the defined attributes, such as Required or Range, and populate the ModelState with any validation errors. Returning a BadRequest result with the ModelState will include the validation errors in the response

.

118. How can you implement authentication and authorization in ASP.NET Core Minimal API?

To implement authentication and authorization in ASP.NET Core Minimal API, you can leverage the authentication and authorization middleware provided by the framework. Here's a high-level overview of the steps involved:

Install the required NuGet packages for the authentication and authorization middleware. For example, Microsoft.AspNetCore.Authentication and Microsoft.AspNetCore.Authorization.

Configure the desired authentication scheme(s) in the services collection. This typically involves setting up authentication options, such as JwtBearer or cookies, and specifying the authentication provider details.

Use the UseAuthentication and UseAuthorization middleware in the request pipeline. Ensure that the UseAuthentication middleware is placed before the routing middleware to authenticate incoming requests.

Apply the necessary authorization attributes to the API endpoints or handler functions to restrict access. For example, [Authorize] attribute can be used to specify that only authenticated users are allowed to access an endpoint.

Here's an example of configuring JWT authentication and applying authorization to an endpoint:

```
// Services
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
.AddJwtBearer(options =>
{
  // Configure JWT authentication options
});

builder.Services.AddAuthorization();

// Request pipeline
var app = builder.Build();

app.UseAuthentication();
app.UseAuthorization();

app.MapMethods("/api/protected", new[] { "GET" }, () =>
{
  // This endpoint requires authorization
}).RequireAuthorization();

app.Run();
```

In this example, the JWT authentication scheme is configured in ConfigureServices, and the UseAuthentication and UseAuthorization middleware are applied in Configure. The /api/protected endpoint requires authorization, and the [Authorize] attribute can also be applied to the handler function for the same effect.