**Top 51 .NET Core Interview Questions**

Here are 51 top .NET Core interview questions:

**What is the purpose of the ConfigureServices and Configure methods in Startup.cs?**

It defines the HTTP pipeline, how the application will respond to request. We can change the sequence of middleware configuration, it accepts IApplicationBuilder as a parameter and two optional parameters: IHostingEnvironment and ILoggerFactory.  
  
Using this method, we can configure built-in middleware such as authentication, error handling and also any custom middleware or third-party middleware.

**Configure Service():** It is used to add services to the container and configure those services. basically, service is a component that is intended for common consumption in the application. There is framework service like MVC, EF core, identity so on. but there are also application services that are application-specific like send mail services.

**Configure():** it is used to specify how the asp.net core application will respond to individual requests. Through this, we actually build the HTTP request pipeline

1. Explain Tag Helpers in ASP.NET Core.
2. Explain the purpose of the appsettings.json file in ASP.NET Core.
3. How can you implement logging in ASP.NET Core?
4. Explain the concept of Middleware Authentication in ASP.NET Core.
5. What is ASP.NET Core Web API and how is it used?

**How can you implement versioning in ASP.NET Core Web API?**

There are several ways we can Implement API versioning in core API.

1. **URL-Based Versioning**: In this approach, we can specify the version of API in URL. For Example, In URL we need to include the version number just like, api/v1/Products

2. **Header-Based Versioning**: In this technique, we must use a request header to specify the version number in header of the API.

3. **Query String Based Versioning**: As the name suggesting we can use query string to specify the version number.

4. **Media type versioning**: This option uses the media type to specify the version of the API. For example, “application/vnd.yourcompany.product-v1+json” in the Accept header.

**1. What’s the difference between .NET and .NET Framework?**

.NET is an open-source, cross-platform framework with core libraries (in NuGet packages) for building modern, cloud-based, and microservices-based applications. It supports development on Linux, macOS, and Windows and provides a modular lightweight runtime deployed as a self-contained executable or a shared library.

.NET Framework, on the other hand, is a Windows-only framework for developing classes and libraries and running code on web services and traditional desktop applications. It provides a larger set of libraries and features than .NET Core, but it’s limited to Windows and is not open-source.

**2. How does ASP.NET Core handle dependency injection?**

Dependency injection is a design pattern of ASP.NET Core that’s handled by the built-in dependency injection container. This container can register and resolve dependencies, typically defined as interfaces implemented by concrete classes.

There are several ways to configure the container, including the ConfigureServices method in the Startup class (the entry point of a .NET application), attributes on classes and properties, and the service provider itself. ASP.NET Core supports constructor, property, and method injection, allowing dependencies to be dynamically injected into methods at runtime.

However, a more up-to-date way to handle dependency injection in ASP.NET Core focuses on singleton, transient, and scoped service lifetimes. You can read more about this [here](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-7.0).

**3. What is Kestrel and how does it differ from IIS?**

Kestrel is a cross-platform, lightweight web server used by default in ASP.NET Core applications. It can run on Linux, macOS, and Windows and provides a fast, scalable, and efficient platform for handling HTTP requests.

Kestrel is designed to be used with a reverse proxy server, such as IIS or Nginx, which handles load balancing and SSL termination tasks.

On the other hand, IIS is a web server specific to Windows that provides more advanced features than Kestrel, such as support for HTTP/2 and WebSocket protocols and integration with Windows authentication and SSL.

**4. What is the purpose of middleware in ASP.NET Core?**

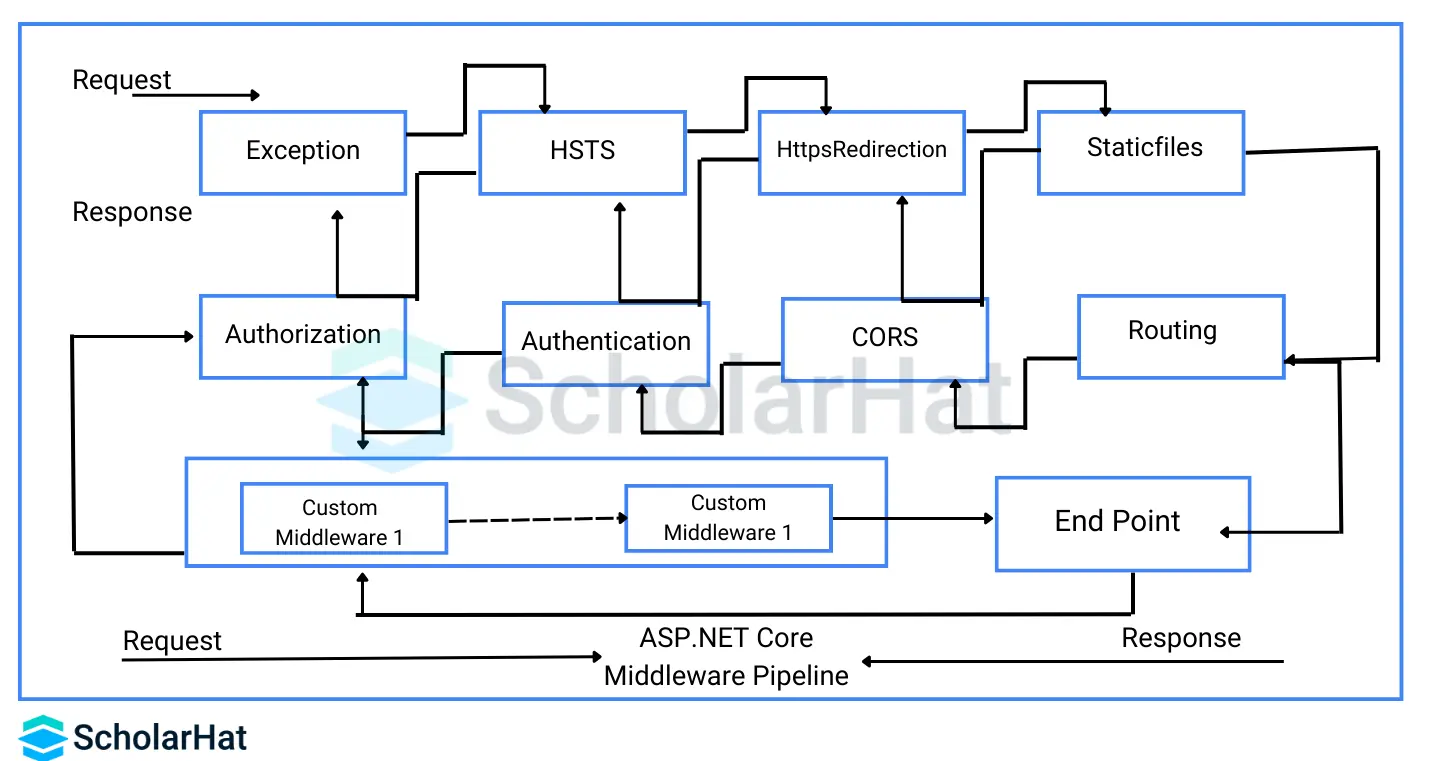
Middleware in ASP.NET Core is a software component responsible for processing requests and generating responses in the web application pipeline. It sits between the server side and the application and is designed to handle cross-cutting concerns, such as authentication, caching, logging, and routing.

The primary purpose of middleware is to provide a modular way of processing HTTP requests and responses, allowing developers to add, remove, or reorder middleware components in the pipeline based on their specific needs. This makes it easy to customize the web application's behavior without modifying the core application logic.

In addition, middleware can perform various tasks, such as modifying request or response headers, handling errors and exceptions, and executing asynchronous code. Middleware can also perform custom processing of requests and responses, such as generating dynamic content or formatting data.

Overall, middleware plays a critical role in the architecture of ASP.NET Core applications, allowing developers to write modular, flexible, and extensible web applications that can be easily customized and scaled.

* In ASP.NET Core, [**middleware**](https://www.scholarhat.com/tutorial/aspnet/middleware-custom-pipeline) is a powerful and flexible concept for processing incoming HTTP requests and generating responses.
* It essentially acts like a pipeline of components, where each component performs a specific task on the request or response before passing it to the next one.
* Middleware is essentially software code that gets plugged into the ASP.NET Core application pipeline.
* Each middleware component is a class that implements the IMiddleware interface.
* Each component gets executed sequentially on every HTTP request and response.



What is the role of middleware in ASP.NET Core?

* Intercepts HTTP requests and responses
* Processes requests in a configurable pipeline
* Offers built-in features like authentication, logging, and compression
* Allows custom middleware for specific application needs
* Extends functionality without modifying the core framework

**5. How does ASP.NET Core handle garbage collection?**

Garbage collection in ASP.NET Core automatically manages the allocation and deallocation of memory that an ASP.NET Core application uses. The garbage collector is responsible for identifying and reclaiming memory no longer needed by the application, thus freeing up resources and improving the application's performance.

The garbage collector in ASP.NET Core uses a generational garbage collection algorithm that divides the heap into gen0, gen1, and gen2, each generation representing a different stage of the object's life cycle. New objects are allocated to the youngest generation, and as they survive longer, they are moved to older generations. The garbage collector collects and frees memory from the youngest generation first and only collects the older generations when necessary.

ASP.NET Core provides several options for configuring and tuning the garbage collector, including setting the maximum size of the heap, the size of the individual generations, and the frequency of garbage collection. These options can be configured using environment variables or application configuration files depending on the needs of the application.

In addition, ASP.NET Core provides several tools and APIs for monitoring and diagnosing garbage collection behavior, including the **GC.Collect()** method, which can force a garbage collection cycle, and the **GC.GetTotalMemory()** method, which returns the total amount of memory used by the application.

Overall, garbage collection in ASP.NET Core is a critical component of the runtime, ensuring efficient memory use and improving the performance and stability of ASP.NET Core applications.

**What is the role of Startup class?**

**Startup class** is responsible for configuration related things as below.

* It configures the services which are required by the app.
* It defines the app's request handling pipeline as a series of middleware components.
* But You can emit this class and Program.cs file contains all startup code.

// Startup class example

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

services.AddRazorPages();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

// other middleware components

}

}

Startup class is specified inside the 'CreateHostBuilder' method when the host is created.  
Multiple Startup classes can also be defined for different environments, At run time appropriate startup classes are used.

**Explain the difference between app.Run and app.Use in ASP.NET Core.**

* **app.Use** method adds a middleware delegate to the application's request pipeline. When you want to pass the context to the next middleware then prefer app.Use method.
* **app.Run** method adds a terminal middleware delegate to the application's request pipeline. When you want to terminate the pipeline then prefer to use the app.Run method.

app.Use((context, nextMidWare) => { context.Response.Body.Write("Hello app.Use"); nextMidWare(context);});

app.Run((context) => context.Response.Body.Write("Hello app.Run"));

app.Use((context, nextMidWare) => context.Response.Body.Write("Hello , again app.Use"));

Output:

Hello app.Use

Hello app.Run

**9. How does ASP.NET Core handle concurrency and parallelism?**

ASP.NET Core provides several mechanisms for handling concurrency and parallelism depending on the application's specific requirements. Some common mechanisms used in ASP.NET Core applications are:

* **Asynchronous programming:** ASP.NET Core supports asynchronous programming by using the async and awaits keywords. Asynchronous programming allows multiple tasks to be executed concurrently without blocking the main thread, improving the application's responsiveness.
* **Parallel programming:** ASP.NET Core supports parallel programming using the Parallel class and the Task Parallel Library (TPL). Parallel programming allows multiple tasks to be executed concurrently across multiple processors, improving the application's performance.
* **Locking and synchronization:** ASP.NET Core provides several mechanisms for locking and synchronization, including the lock keyword, the Interlocked class, and the Monitor class. These mechanisms allow multiple threads to access shared resources safely and prevent race conditions.
* **Concurrency control:** ASP.NET Core supports concurrency control through transactional memory and the optimistic concurrency control (OCC) pattern. Concurrency control ensures that multiple threads can access and modify shared resources without interfering with each other.

Using these mechanisms, developers can build ASP.NET Core applications that are more responsive, scalable, and efficient, handling multiple requests and tasks concurrently and in parallel. However, using these mechanisms carefully and appropriately is important, as concurrency and parallelism can introduce new challenges, such as race conditions, deadlocks, and thread starvation.

**10. How do you implement caching in ASP.NET Core?**

Response caching in ASP.NET Core is a technique used to improve the performance and scalability of web applications by caching the ASP.NET Core MVC responses returned by the server for a specific period. Caching the response can help reduce the number of requests made to the server, as clients can reuse the cached response instead of requesting the same resource again.

Response caching works by adding a caching layer between the client and the server. When a client requests a resource, the caching layer checks whether the response for the request has been cached. If the response is cached, the caching layer returns the cached response to the client. If the response is not cached, the request is forwarded to the server, and the server generates the response and caches it for future use.

In ASP.NET Core, response caching can be implemented using the [ResponseCache] attribute, which can be applied to an action method in a controller. The attribute allows developers to specify the caching behavior, such as the duration of the cache, the location of the cache, and the cache key. By default, the caching location is on the client side, but it can also be set to a distributed or proxy cache depending on the needs of the application.

Response caching can significantly impact the performance and scalability of web applications, particularly for resources that are expensive to generate, such as database queries or API calls. However, it’s important to use response caching judiciously, as caching can also lead to stale data being returned to clients. Therefore, setting appropriate caching policies and ensuring the cache is invalidated when the underlying data changes are crucial.

**11. What’s the difference between middleware and a filter in ASP.NET Core?**

In ASP.NET Core, middleware and filters are two mechanisms used for processing requests and responses.

Middleware is a software component between the web server (like Apache) and the application and processes requests and responses during the application development. Middleware can be used for various tasks, such as authentication, logging, and error handling. Middleware is executed in a pipeline, and each middleware component can modify the request or response before passing it to the next component in the pipeline.

Conversely, filters are used to perform cross-cutting concerns on controllers and actions in an MVC application. Filters can be used for authorization, validation, and caching tasks. Filters are executed before and after the action method, and they can modify the request or response or short-circuit the request processing if necessary.

The main difference between middleware and filters is their scope and the way they are executed. Middleware is executed globally and can be used for any request or response. In contrast, filters are executed only for specific controllers or actions and can be used to modify the request or response before or after the action method.

**12. What is Core CLR?**

CoreCLR (Common Language Runtime, now renamed to .NET Runtime) is the runtime environment executing ASP.NET Core applications. It is the open-source implementation of the .NET runtime, developed by Microsoft and available on multiple platforms, including Windows, Linux, and macOS.

CoreCLR provides a managed execution environment for ASP.NET Core applications, including memory management, garbage collection, type safety, and security. It also supports just-in-time (JIT) compilation, which compiles code at runtime to native machine code, allowing for faster execution.

CoreCLR is designed to be modular, with various components such as the garbage collector, JIT compiler, and primitive data type system implemented as separate modules. This modularity allows for more flexibility and customization in building and deploying .NET Core applications.

CoreCLR is a critical component of the .NET platform, providing the necessary runtime infrastructure for developing and executing .NET applications across different platforms.

**13. Have you worked with Docker on ASP.NET Core projects?**

The Docker platform allows developers to package and deploy applications in lightweight, portable containers. In the context of ASP.NET Core, Docker provides a way to package and deploy ASP.NET Core applications and their dependencies in a self-contained, isolated container that can run on any platform that supports Docker.

Using Docker in ASP.NET Core, developers can create Docker images of their applications, which can be deployed to any environment that supports Docker. This makes it easy to deploy ASP.NET Core applications consistently and reliably, without worrying about differences in the underlying infrastructure.

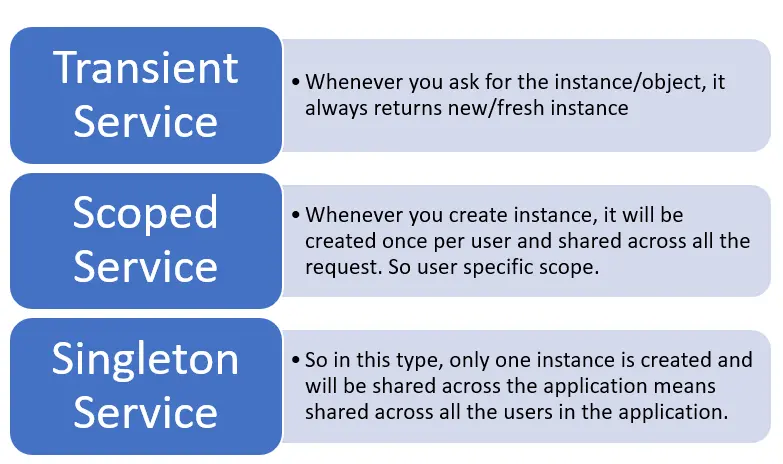
Docker also provides a way to manage and orchestrate containers in a distributed system, allowing developers to scale their applications up or down as needed.

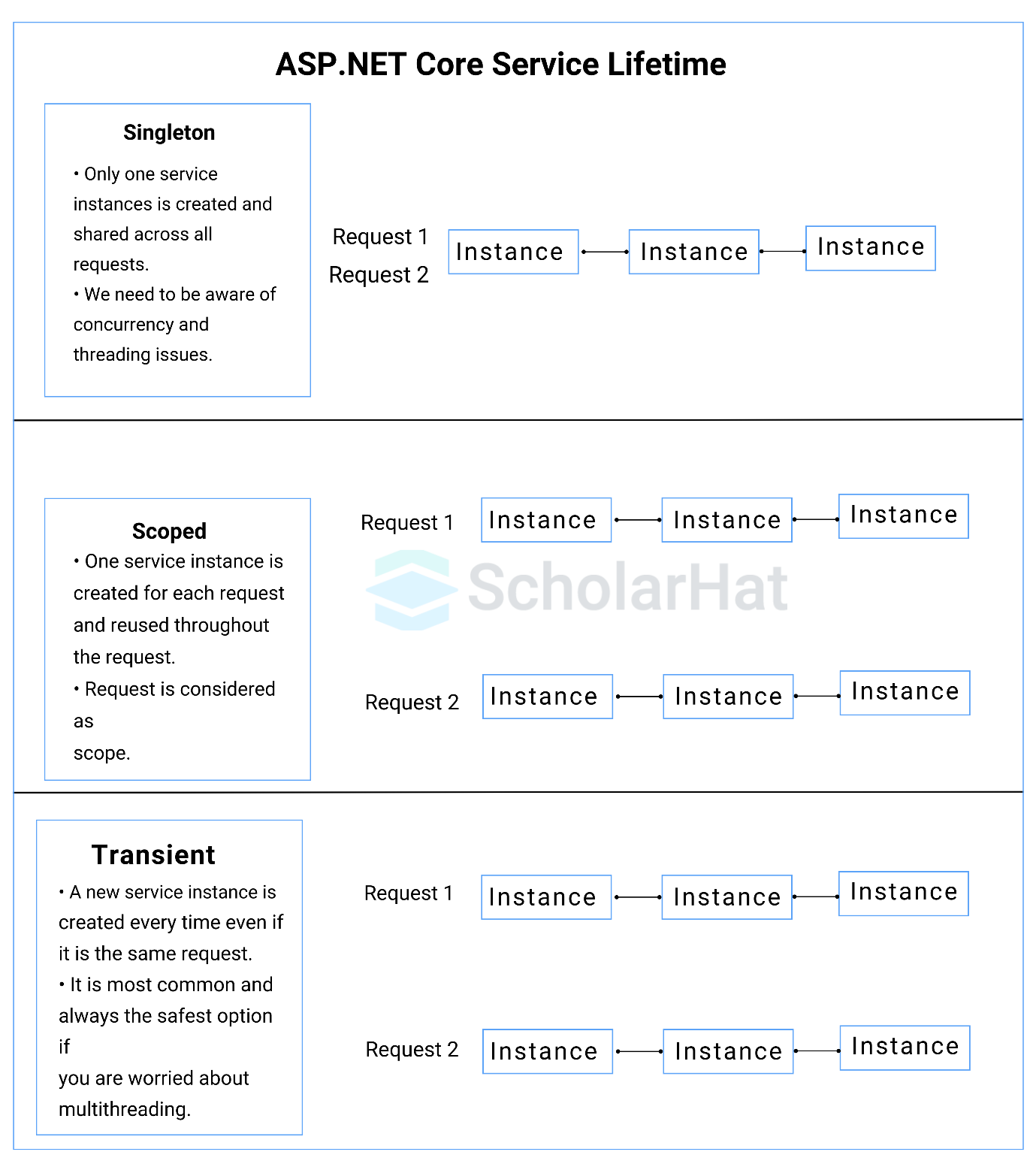
Overall, Docker is a powerful tool for developing, deploying, and managing ASP.NET Core applications, providing a portable, flexible, and scalable environment for building modern applications.

**Service Lifetimes in ASP.NET Core.**

crucial for optimizing performance, managing resources, and preventing memory leaks. Here are the three primary service lifetimes:

* **Transient Service:** Instance is created each time it is requested.
* **Scoped Service:** User-specific instance is created once per user and shared across all the requests.
* **Singleton Service:** Single Instance is created once a lifetime of the application.





1. **Transient Lifetime:** A new instance of the service is created every time it's injected.
2. **Scoped Lifetime:** A single instance of the service is created per request scope (e.g., per HTTP request).
3. **Singleton Lifetime:** A single instance of the service is created for the entire lifetime of the application.

**How to configure and manage multiple environments in ASP.NET Core applications?**

Managing multiple environments in ASP.NET Core is crucial for a smooth development and deployment process. Here's how to configure and manage them effectively:

Configuration:

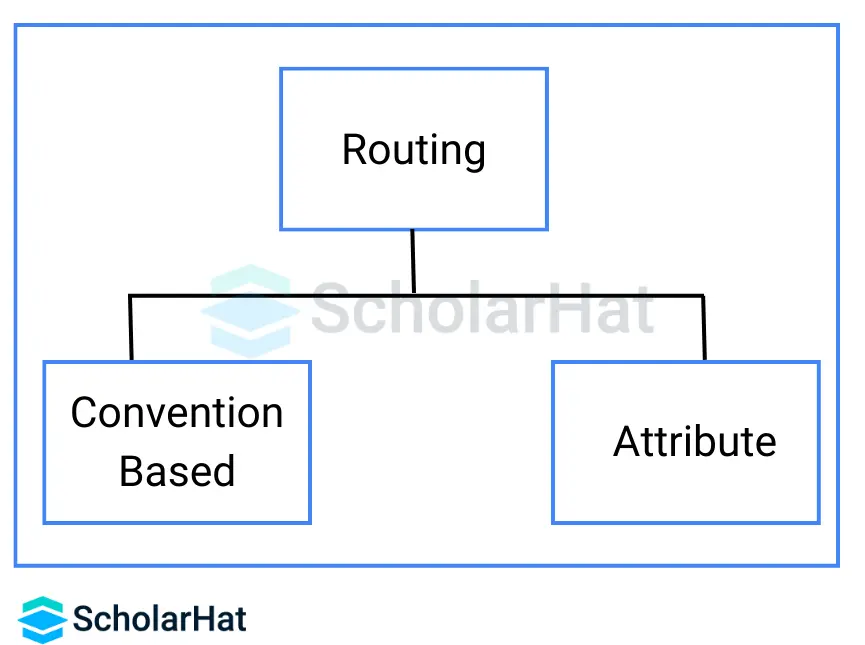
* **Environment Variable:** This is the most common method. Set an environment variable like ASPNETCORE\_ENVIRONMENT to values like Development, Staging, or Production. Your code can then react to this variable to adjust settings.
* **Multiple appsettings.json files:** Create separate appsettings.json files for each environment, named appsettings.Development.json, appsettings.Staging.json, etc. Your application can then read the specific file based on the environment variable.
* **Other sources:** You can also use other configuration sources like Azure Key Vault, command-line arguments, or custom providers.

Managing Environments:

* **Tools:** Consider using tools like dotnet CLI or deployment platforms like Azure DevOps to manage environment-specific configuration and deployment processes.
* **Code isolation:** Separate code specific to different environments (e.g., database connection strings) into different assemblies or modules.
* **Secret management:** Use secure methods like Azure Key Vault to store sensitive information like passwords or API keys, ensuring they are not exposed in configuration files.

**Describe how Routing works in ASP.NET Core and its different types.**

* [**Routing in ASP.NET Core**](https://www.scholarhat.com/tutorial/aspnet/routing) acts like a map, directing incoming requests to the right destination.
* It matches the URL path against predefined templates in two main ways: convention-based (like "/Products/{id}" for product details) and attribute-based (using [Route] annotations on controllers).
* These routes can be named for easier navigation and URL generation.
* Convention-based routing kicks in first, followed by attribute-based, ensuring flexibility and control.
* This dynamic system lets you build clean, intuitive URLs for your users.

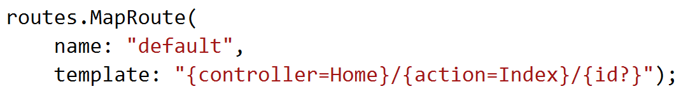


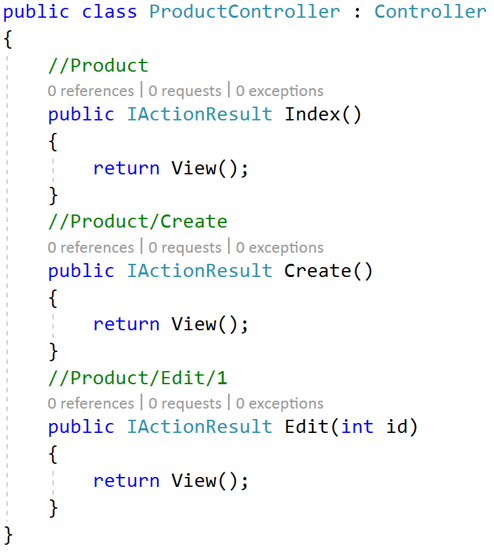
There are two main ways to define routes in ASP.NET Core:

1. **Convention-Based Routing**

It creates routes based on a series of conventions representing all the possible routes in your system. Convention-based is defined in the Startup.cs file.

Convention-Based Routing Configuration & Mapping





1. **Attribute Routing**

It creates routes based on attributes placed on the controller or action level. Attribute routing provides us more control over the URL generation patterns which helps us in SEO.



Attribute Routing Tokens

One of the cool things about [**ASP.NET Core**](https://www.scholarhat.com/tutorial/aspnet) routing is its flexibility as compared to ASP.NET MVC5 routing since it provides tokens for [area], [controller], and [action]. These tokens get replaced by their values in the route table.



**Explain how to enable Cross-Origin Requests (CORS) in ASP.NET Core for API access from different domains.**

* Install Microsoft.AspNetCore.Cors NuGet package.
* Register CORS middleware in ConfigureServices of Startup.cs.
* Define CORS policy with allowed origins, methods, and headers.
* Use the [EnableCors] attribute on controllers or apply globally.
* Optionally, set exposed response headers for client access.

**What is the best way to manage errors in .NET Core?**

There are mainly four ways to manage errors in .NET Core for web APIs.

* Developer Exception Page
* Exception Handler Page
* Exception Handle Lambda
* UseStatusCodePages

But, in all these four, the best way is "Developer Exception Page" as it provides detailed information (stacks, query string parameters, headers, cookies) about unhandled request exceptions. You can easily enable this page by running your applications in the development environment. This page runs early in the middleware pipeline, so you can easily catch the exception in middleware.

**How do you implement authentication in ASP.NET Core?**

Authentication in ASP.NET Core is implemented using the Authentication middleware. You configure it in the Startup class, specifying the authentication scheme(s) your application uses. ASP.NET Core supports various authentication mechanisms, such as cookies, JWT bearer tokens, and external authentication providers like Google, Facebook, etc. You set up these schemes in the ConfigureServices method and then apply them to your application using attributes or policies.

**Explain the differences between JWT, OAuth, and OpenID Connect.**

* **JWT (JSON Web Token):** A compact, URL-safe means of representing claims to be transferred between two parties. It’s a token format used in authentication and information exchange.
* **OAuth:** An authorization framework that enables a third-party application to obtain limited access to an HTTP service. It’s about delegation of authorization.
* **OpenID Connect:** A simple identity layer on top of OAuth 2.0, which allows clients to verify the identity of the end-user and to obtain basic profile information in an interoperable and REST-like manner.

**How do you configure authorization policies?**

Authorization policies in ASP.NET Core are configured using the ConfigureServices method of the Startup class by using the AddAuthorization method. You can define policies that incorporate various requirements, such as user roles, claims, or custom requirements. These policies are then applied to controllers or actions within your application through attributes (like [Authorize]) or by using the policy name directly if more complex rules are needed.

**What is the role of ASP.NET Core Identity in authentication and authorization?**

ASP.NET Core Identity is a membership system that adds login functionality to ASP.NET Core applications. It supports authentication (verifying who a user is) and authorization (determining what resources a user is allowed to access). ASP.NET Core Identity allows for easily integrating user profiles and managing user accounts, passwords, roles, and security tokens.

**What is Entity Framework Core?**

Entity Framework Core (EF Core) is an open-source, lightweight, extensible, and cross-platform version of Entity Framework, Microsoft’s Object Relational Mapper (ORM) for .NET. It enables developers to work with a database using .NET objects, eliminating the need for most of the data-access code that developers usually need to write.

**How do you configure EF Core in ASP.NET Core Projects?**

To configure EF Core in an ASP.NET Core project, you typically:

* Install the necessary NuGet packages for EF Core and the database provider you’re using (e.g., Microsoft.EntityFrameworkCore.SqlServer for SQL Server).
* Define your DbContext and entity classes to represent your database schema.
* Register the DbContext with the dependency injection container in the Startup.cs file using the services.AddDbContext method.
* Configure the connection string in the appsettings.json file and read it in Startup.cs to set up the database connection.

**Explain the differences between Code First and Database First approaches.**

* Code First: Developers write C# classes to define the database model; then, EF Core migrations are used to generate the database schema based on these classes. It’s suitable for new projects where the database schema is developed alongside the application.
* Database First: Begins with an existing database, and EF Core scaffolding is used to generate the entity classes and DbContext based on the schema of the existing database. It’s suitable for projects that need to work with an existing database.

**How do you handle database migrations?**

Database migrations in EF Core are handled through the dotnet ef migrations command-line tool or the Package Manager Console in Visual Studio. To handle migrations, you typically:

* Create a migration using the Add-Migration command, providing a name for the migration.
* Apply the migration to the database using the Update-Database command, which updates the database schema to match the current model by applying the necessary changes.

**How do you secure ASP.NET Core Applications?**

Securing ASP.NET Core applications involves multiple strategies, including:

* Implementing authentication and authorization (e.g., using ASP.NET Core Identity).
* Using HTTPS to encrypt data in transit.
* Implementing data protection to secure sensitive data.
* Using anti-forgery tokens to prevent Cross-Site Request Forgery (CSRF) attacks.
* Validating and sanitizing input to prevent Cross-Site Scripting (XSS) attacks.

**Explain Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) attacks and how to mitigate them.**

* **XSS:** An attacker injects malicious scripts into content that is then served to other users. Mitigation includes validating and encoding user input and using Content Security Policy (CSP) headers.
* **CSRF:** An attacker tricks a user’s browser into executing unauthorized actions on a web application in which they’re authenticated. Mitigation involves using anti-forgery tokens that validate that the requests to the server are legitimate and originated from the site itself.

**What are some best practices for securing APIs?**

Best practices for securing APIs include:

* Implementing authentication and authorization, often using tokens (such as JWT).
* Using HTTPS to secure data in transit.
* Validating and sanitizing input to prevent injection attacks.
* Limiting request rates to prevent abuse.
* Applying the principle of least privilege to API access.

**How do you handle sensitive data in ASP.NET Core applications?**

Handling sensitive data securely involves:

* Encrypting sensitive data at rest and in transit (using HTTPS).
* Using ASP.NET Core’s Data Protection API to encrypt data in your application.
* Ensuring that sensitive data is not exposed in logs or error messages.
* Implementing proper access controls to limit who can access sensitive data.

**How do you deploy ASP.NET Core applications?**

Deploying ASP.NET Core applications can be done in several ways, including:

* To a web server, like IIS, using Web Deploy or FTP.
* To cloud services, like Azure App Service, directly from Visual Studio or using CI/CD pipelines.
* Using containers, deploying as a Docker container to a container orchestration service like Kubernetes.
* Creating self-contained deployments (SCD) or framework-dependent deployments (FDD) for hosting on any platform that supports .NET.

**What are some deployment options available for ASP.NET Core?**

* **IIS (Internet Information Services):** A flexible, secure, and manageable Web server for hosting anything on the Web.
* **Kestrel:** A cross-platform web server for ASP.NET Core.
* **Docker Containers:** Package applications with all of their dependencies and services.
* **Cloud Services: A**zure App Service, AWS Elastic Beanstalk, and Google Cloud App Engine are popular cloud hosting options.
* **Linux or Windows Virtual Machines:** For full control over the hosting environment.

**Explain the role of Docker and Kubernetes in ASP.NET Core deployment.**

* **Docker:** Provides a way to package ASP.NET Core applications with all their dependencies into containers, ensuring consistency across environments and simplifying deployment.
* **Kubernetes:** An orchestration tool for Docker containers, managing aspects like scaling, load balancing, and self-healing of containers in cluster environments, facilitating microservices architecture.

**How do you implement continuous integration and continuous deployment (CI/CD) pipelines for ASP.NET Core?**

Utilize tools like Azure DevOps, Jenkins, or GitHub Actions to automate the build, test, and deployment process of ASP.NET Core applications.

Set up pipelines to include steps for code compilation, running tests, and deploying to various environments (development, staging, production) based on triggers like code commits or manual approvals.

**What are microservices?**

* **Definition:** An architectural style that structures an application as a collection of loosely coupled, independently deployable services.
* **Design in ASP.NET Core:** Use ASP.NET Core’s lightweight, modular nature to develop individual microservices. Leverage APIs for communication between services and Docker containers for isolation and deployment.

**Explain the role of messaging queues and service buses in distributed systems.**

* **Messaging Queues (e.g., RabbitMQ, Azure Queue Storage):** Enable asynchronous communication between services, enhancing decoupling and scalability.
* **Service Buses (e.g., Azure Service Bus, MassTransit):** Provide more complex routing, message sequencing, and transaction management, facilitating sophisticated integration patterns across services.

**What challenges do you face when developing distributed systems with ASP.NET Core?**

* **Complexity:** Increased complexity in managing multiple services, inter-service communication, and data consistency.
* **Deployment:** Coordinating deployment across multiple services.
* **Monitoring and Logging:** Centralizing logs and monitoring from disparate services.
* **Latency:** Increased latency due to network calls between services.

**How do you optimize performance in ASP.NET Core applications?**

* **Response Caching:** Use response caching to reduce the load on the server and speed up responses.
* **Asynchronous Programming:** Leverage async/await to improve scalability and responsiveness.
* **Minimize Resource Usage:** Optimize database queries, minimize the use of blocking calls, and use efficient algorithms.

**What are some techniques to reduce latency in web applications?**

* **Content Delivery Networks (CDNs):** Use CDNs to serve static files closer to the user’s location.
* **Load Balancing:** Distribute requests across multiple servers to reduce load and improve response times.
* **Optimize Assets:** Minify and bundle CSS and JavaScript files, compress images.

**Explain caching strategies in ASP.NET Core.**

* **In-Memory Caching:** Stores data in the memory of the web server for quick access. Suitable for single-server or ephemeral data.
* **Distributed Caching:** Distributed cache systems like Redis or Memcached can be used to share cache data across multiple servers, which is beneficial for scalable applications.
* **Response Caching:** Cache the entire response or parts of it to serve repeated requests quickly.

**How do you identify and resolve performance bottlenecks?**

To identify performance bottlenecks, I use tools like Visual Studio Diagnostic Tools, Application Insights, or third-party profilers. I focus on areas like slow database queries, inefficient memory use, or CPU-intensive operations. Once identified, I resolve these bottlenecks by optimizing the code, implementing caching, and using asynchronous programming models to improve response times and resource utilization.

**What are the different types of tests you can write for ASP.NET Core applications?**

In ASP.NET Core applications, we can write unit tests, integration tests, and functional tests. Unit tests focus on testing individual components or methods for correctness. Integration tests verify the interaction between components or systems, such as database access and API calls. Functional tests, or end-to-end tests, validate the application as a whole, ensuring that the user experience is as expected.

**How do you unit test controllers and services?**

To unit test controllers and services, I use a testing framework like xUnit or NUnit, along with a mocking library like Moq. For controllers, I mock the services they depend on to isolate the controller logic. For services, I mock external dependencies like database contexts or external APIs. This approach allows me to test the behavior of my code in isolation from its dependencies.

**Explain integration testing in ASP.NET Core.**

Integration testing in ASP.NET Core involves testing the application’s components as a whole, ensuring they work together as expected. This includes testing interactions with databases, file systems, and external services. I use the ASP.NET Core TestHost package to run the application in a test environment, allowing me to send requests to the application and assert the responses and side effects.

**What are some popular testing frameworks used with ASP.NET Core?**

Popular testing frameworks for ASP.NET Core include xUnit, NUnit, and MSTest for writing test cases. For mocking dependencies, libraries like Moq, NSubstitute, and FakeItEasy are commonly used. For integration testing, the ASP.NET Core provides built-in support through Microsoft.AspNetCore.TestHost package, which is often combined with SpecFlow for behavior-driven development (BDD) scenarios.

**How do you create RESTful APIs in ASP.NET Core?**

To create RESTful APIs in ASP.NET Core, I define controllers inheriting from ControllerBase and use attributes to map HTTP verbs to action methods. I adhere to REST principles, designing endpoints around resources and using HTTP verbs (GET, POST, PUT, DELETE) semantically. For content negotiation, I leverage ASP.NET Core’s built-in support to automatically handle JSON, XML, or custom formats based on the Accept header in the request.

**What is the role of controllers and actions?**

Controllers in ASP.NET Core serve as the entry point for handling HTTP requests and returning responses. Each controller contains actions, which are methods that handle requests for a specific route or URL. Actions read data from the request, perform operations (such as calling a service), and return a response, which can be a view, data, or status code.

**Explain content negotiation in ASP.NET Core Web API.**

Content negotiation in ASP.NET Core Web API involves selecting the appropriate format for the response content based on the client’s request. ASP.NET Core automatically handles this through the Accept header, where the client specifies the desired media type(s). The framework then uses formatters to serialize the response data into the requested format, such as JSON or XML.

**How do you handle routing and versioning in Web APIs?**

For routing, I use attribute routing in ASP.NET Core to define routes directly on controllers and actions, providing clear and customizable URL patterns. For versioning, I implement URL path, query string, or header-based versioning strategies using built-in services or third-party libraries. This approach allows me to maintain multiple versions of the API, ensuring backward compatibility while introducing new features.