# ASP .Net Core-MVC Interview Questions

### What is ASP.NET Core?

ASP.NET Core is an open-source, cross-platform framework for building modern, cloud-based, and internet-connected applications, including web applications, APIs, and microservices. It is a significant redesign of ASP.NET, providing a modular and high-performance framework that is optimized for developer productivity.

### How does ASP.NET Core differ from previous versions of ASP.NET?

ASP.NET Core differs from previous versions primarily in terms of its architecture, cross-platform support, performance improvements, and modularity. It is designed to be lightweight, modular, and highly extensible, making it suitable for various applications and deployment scenarios.

### Differentiate between ASP.NET Core MVC and ASP.NET Core Web API.

ASP.NET Core MVC is a framework for building web applications following the Model-View-Controller architectural pattern, primarily used for creating UI-based applications. ASP.NET Core Web API, on the other hand, is used to build HTTP-based APIs that clients can use for web applications, mobile apps, and other services. While MVC deals with views, controllers, and models, Web API focuses on endpoints that return data in various formats like JSON or XML.

### Discuss the benefits of using ASP.NET Core over ASP.NET Framework.

Some benefits of ASP.NET Core over ASP.NET Framework include:

- **Cross-platform support**: ASP.NET Core can run on Windows, macOS, and Linux, providing more flexibility in deployment environments.
- **Improved performance**: ASP.NET Core is optimized for performance, resulting in faster response times and better scalability.
- **Modularity**: ASP.NET Core is designed to be modular, allowing developers to include only the necessary components, reducing the application's overall footprint.
- **Easier deployment with Docker and cloud platforms**: ASP.NET Core applications can be easily containerized with Docker and deployed to cloud

platforms like Azure, AWS, and Google Cloud.

- **Modern development practices**: ASP.NET Core supports modern development practices such as dependency injection and middleware pipelines, promoting better code organization and maintainability.
- **Smaller footprint**: ASP.NET Core has a smaller footprint compared to the ASP.NET Framework, making it more suitable for microservices and lightweight applications.

### Explain the design principles behind ASP.NET Core. How does it differ from previous versions of ASP.NET?

ASP.NET Core was designed with the principles of modularity, cross-platform functionality, and performance in mind. It represents a significant departure from previous versions of ASP.NET by:

- **Modularity**: ASP.NET Core allows developers to include only the components they need, reducing application size and improving performance.
- **Cross-Platform**: It can run on Windows, Linux, and macOS, making it more versatile than its predecessor.
- **Performance**: ASP.NET Core is optimized for modern web applications, offering improved performance due to its lightweight and modular nature.

### What is Kestrel in the context of ASP.NET Core?

Kestrel is a lightweight, cross-platform web server that comes bundled with ASP.NET Core. It's the default web server used by ASP.NET Core applications and is designed for high performance. Kestrel can also be used as an edge server or a reverse proxy in combination with other web servers like Nginx or IIS.

### What is the Model–View–Controller (MVC) pattern?

Model–View–Controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts to separate the internal representation of information from the way that information is presented to or accepted from the user.

### What are the components of MVC in web applications?

MVC is a framework for building web applications using an MVC (Model View Controller) design:

- **The Model** represents the application core (for instance, a list of database records).
- **The View** displays the data (the database records).
- **The Controller** handles the input (to the database records).
- The MVC model also provides full control over HTML, CSS, and JavaScript.

### What are the logic layers defined by the MVC model?

The MVC model defines web applications with 3 logic layers:

- **The Business Layer (Model logic)**: The part of the application that handles the logic for the application data, often retrieving and storing data in a database.
- **The Display Layer (View logic)**: The part of the application that handles the display of the data, typically created from the model data.
- **The Input Control (Controller logic)**: The part of the application that handles user interaction, reading data from a view, controlling user input, and sending input data to the model.

### How does MVC help in managing complex applications?

The MVC separation helps manage complex applications by allowing focus on one aspect at a time. For example, developers can focus on the view without depending on the business logic, making it easier to test the application as well.

### What are the advantages of MVC?

The Model–View–Controller (MVC) framework offers several advantages:

- **Multiple view support**:
  Due to the separation of the model from the view, the user interface can display multiple views of the same data simultaneously.
- **Change accommodation**:
  User interfaces tend to change more frequently than business rules (e.g., different colors, fonts, screen layouts, and levels of support for new devices such as cell phones or PDAs). Since the model does not depend on the views, adding new types of views to the system generally does not affect the model. As a result, the scope of change is confined to the view.
- **Separation of Concerns (SoC)**:
  Separation of Concerns is one of the core advantages of ASP.NET MVC. The

MVC framework provides a clean separation of the UI, business logic, model, or data.

- **More control**:
  The ASP.NET MVC framework provides more control over HTML, JavaScript, and CSS than traditional Web Forms.
- **Testability**:
  The ASP.NET MVC framework provides better testability of the web application and good support for test-driven development.
  **Full features of ASP.NET**:
  One of the key advantages of using ASP.NET MVC is that it is built on top of the ASP.NET framework. Hence, most of the features of ASP.NET, such as membership providers, roles, etc., can still be used.

### What is the purpose of the appsettings.json file in an ASP.NET Core application?

The appsettings.json file in an ASP.NET Core application is used for configuration. It stores settings like connection strings, application settings, and environment-specific configurations. ASP.NET Core's configuration system can read settings from various sources, and appsettings.json serves as a convenient place to store application-level configurations that can be easily accessed throughout the application.

### What is the role of the wwwroot folder in an ASP.NET Core application?

The wwwroot folder in an ASP.NET Core application is designated as the root web directory. It contains static files like HTML, CSS, JavaScript, and image files. These files are served directly to clients and are accessible via a path relative to the web root. ASP.NET Core applications use the Static Files Middleware (app.UseStaticFiles()) to serve static files from the wwwroot folder.

### Explain the concept of Model Binding in ASP.NET Core.

Model Binding in ASP.NET Core automatically maps data from HTTP requests to action method parameters. When a request is made, model binding goes through the incoming data (from the form values, query string, route data, and JSON POST body), and attempts to bind it to the parameters of the action method being called. This simplifies the code for handling requests by abstracting the manual extraction of data.

### Explain the concept of Razor syntax in ASP.NET Core.

Razor syntax is a markup syntax that blends C# code with HTML. It allows developers to generate web content with an ASP.NET Core view dynamically. Razor minimizes the number of characters and keystrokes required in a file and enables a fast, fluid coding workflow. With Razor, you can easily incorporate C# logic directly within an HTML file, making it powerful for developing dynamic web pages efficiently.

### What are the different hosting options available for ASP.NET Core applications?

ASP.NET Core applications can be hosted in several environments, including:

- **Kestrel**: A cross-platform web server for ASP.NET Core.
- **IIS**: As a reverse proxy server.
- **HTTP.sys**: For Windows-based internet services without using IIS.
- **Docker containers**: Providing a way to package applications with their dependencies and deploy them in a containerized environment.
- **Cloud services like Azure App Service**: Which offers a fully managed platform for building, deploying, and scaling web apps.

### Explain the concept of environment-specific configuration in ASP.NET Core.

ASP.NET Core supports environment-specific configuration, allowing developers to have different configurations (e.g., for development, staging, and production) without changing the code. This is achieved using multiple appsettings files (e.g., appsettings.Development.json, appsettings.Production.json) and environment variables. The framework automatically loads the appropriate settings based on the current environment, which can be set through the ASPNETCORE_ENVIRONMENT environment variable. This feature simplifies managing application behavior across different deployment environments, improving the development workflow and deployment process.

### 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.

### Explain the concept of Middleware pipeline in ASP.NET Core.

The Middleware pipeline in ASP.NET Core is a mechanism for how HTTP requests are processed by the web application. Each middleware component in the pipeline is responsible for invoking the next middleware in the sequence or short-circuiting the pipeline. Middleware can perform a variety of tasks, such as authentication, routing, session management, and more. The order in which middleware components are added to the pipeline defines the order of execution for request processing and response generation.

### Explain the concept of Middleware.

In ASP.NET Core, middleware 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

### How to write custom ASP.NET Core middleware for specific functionalities?

To create custom ASP.NET Core middleware, you need two key things: a class implementing IMiddleware or an extension method.

- The class constructor takes a RequestDelegate which defines your next step in the request pipeline.
- In the Invoke method, access the HttpContext to perform your desired functionality.
- Remember to call _next(httpContext) to continue processing the request.
- You can then modify the request/response objects, and log information, or even short-circuit the pipeline.
  This opens up endless possibilities for custom tasks like authentication, logging, or request manipulation in your ASP.NET Core application.

### What is the IActionResult interface in ASP.NET Core?

The IActionResult interface is used to represent the result of an action method in ASP.NET Core. It provides a way to encapsulate different types of action results into a single return type. Implementations of IActionResult can represent various HTTP responses such as status codes, JSON data, views, file downloads, and more. This abstraction allows for flexible and maintainable code within controller actions.

### What are Tag Helpers in ASP.NET Core?

Tag Helpers in ASP.NET Core are a new feature that simplifies the process of creating and working with HTML elements in Razor views. They allow developers to use HTML-like syntax with server-side logic to generate HTML markup. Tag Helpers make writing and maintaining views easier by reducing the amount of inline C# code and improving the readability of the markup.

### What is the difference between TempData, ViewBag, and ViewData in ASP.NET Core?

TempData, ViewBag, and ViewData are mechanisms for passing data between controllers and views in ASP.NET Core. TempData persists data for the duration of an HTTP request and subsequent redirect, ViewBag is a dynamic property used to pass data from controllers to views during the current request, and ViewData is similar to ViewBag but uses a dictionary to pass data from controllers to views.

### Explain the concept of Content Negotiation in ASP.NET Core:

Content Negotiation is the process of determining the best content type (e.g., JSON, XML, HTML) to return to a client based on its preferences and capabilities. In ASP.NET Core, Content Negotiation is handled automatically by the framework through MediaTypeFormatters, which serialize and deserialize data based on the content type requested by the client.

### How do you handle errors and exceptions in ASP.NET Core?

ASP.NET Core provides middleware for handling errors and exceptions globally or at the application level. Developers can use built-in middleware like UseExceptionHandler to catch unhandled exceptions and return appropriate error responses. Additionally, custom middleware can be implemented to handle specific error scenarios.

### Explain strategies for handling errors and exceptions in ASP.NET Core applications.

Error and exception handling are crucial for building robust and reliable ASP.NET Core applications. Here are some key strategies to consider:

- **Middleware:** Global error handling with specific responses, logging, and custom error pages.
- **Try/Catch:** Localized error handling for specific scenarios and resource cleanup.
- **Exception Filters:** Intercept and handle exceptions before reaching the middleware.
- **Descriptive Errors:** Throw specific exceptions with clear messages for better debugging.
- **Logging:** Capture all exceptions for analysis and later troubleshooting.

### How do you handle logging in ASP.NET Core?

ASP.NET Core provides built-in logging capabilities through the Microsoft.Extensions.Logging framework. Developers can configure logging providers such as Console, Debug, EventSource, File, or third-party providers to log messages at different levels of severity. Logging can be configured in the Startup.cs file or through configuration settings.

### Explain the concept of Routing in ASP.NET Core.

Routing in ASP.NET Core is the process of mapping incoming requests to the appropriate controllers and actions. It is configured in the Program.cs file and enables the application to understand URLs, thereby determining how requests are handled. ASP.NET Core supports both conventional routing, which uses predefined patterns, and attribute routing, which allows for more granular control by decorating controllers and actions with attributes that define routes.

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

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

Routing in ASP.NET Core 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:**

**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**
- **Conventions based Routing Configuration**

- **Conventions based Routing Mapping**

```
public class ProductController : Controller
{
    //Product
    0 references | 0 requests | 0 exceptions
    public IActionResult Index()
    {
        return View();
    }
    //Product/Create
    0 references | 0 requests | 0 exceptions
    public IActionResult Create()
    {
        return View();
    }
    //Product/Edit/1
    0 references | 0 requests | 0 exceptions
    public IActionResult Edit(int id)
    {
        return View();
    }
}
```

**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.

```
[Route("Home")]
0 references
public class HomeController : Controller
{
    [Route("")]      // "Home"
    [Route("Index")] // "Home/Index"
    [Route("/")]     //  ""
    0 references | 0 requests | 0 exceptions
    public IActionResult Index()
    {
        return View();
    }
    [Route("About")] // "Home/About"
    0 references | 0 requests | 0 exceptions
    public IActionResult About()
    {
        return View();
    }
}
```

**Attribute Routing Tokens**

- One of the cool things about ASP.NET Core 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.

```
[Route("[controller]/[action]")]
0 references
public class ProductsController : Controller
{
    [HttpGet] // Matches '/Products/List'
    0 references | 0 requests | 0 exceptions
    public IActionResult List()
    {
        // ...
        return View();
    }
    [HttpGet("{id}")] // Matches '/Products/Edit/{id}'
    0 references | 0 requests | 0 exceptions
    public IActionResult Edit(int id)
    {
        // ...
        return View();
    }
}
```

### Explain the concept of CORS (Cross-Origin Resource Sharing) in ASP.NET Core. How do you configure it?

CORS is a security feature that allows or restricts web applications from making requests to resources hosted on a domain different from the one the application was served from. In ASP.NET Core, CORS can be configured using middleware. You configure it by adding the CORS services in the ConfigureServices method of the Startup class and then enabling CORS with the desired policy in the Configure method. This setup allows specifying which origins, headers, and methods are allowed for cross-origin requests.

### Describe strongly typed views and their benefits in ASP.NET Core MVC.

In ASP.NET Core MVC, a strongly typed view is a view that's explicitly associated with a specific data type or model class. This means the view is aware of the model's properties and methods, offering several benefits compared to "dynamic" views.

**Benefits of strongly typed views:**

- **Compile-time type checking.**
- **Improved tooling support.**

- **Reduced runtime errors.**
- **Cleaner and more readable code.**
- **Improved maintainability.**

### Explain Partial Views and their use cases in ASP.NET Core MVC.

Partial views are reusable Razor components in ASP.NET Core MVC applications. They're essentially mini-views, built as .cshtml files but without the @page directive (used in Razor Pages). Instead, they're incorporated into other views to render specific sections of the UI.

**Use Cases for Partial Views:**

- **Navigation menus:** Render a common navigation menu across multiple pages using a single partial view.
- **Product lists:** Display dynamic product listings on different pages by dynamically loading the partial view with different product data.
- **Comments sections:** Implement a reusable comment section component across various blog posts or articles.
- **Modals and popups:** Create reusable modals or popup windows for different functionalities.
- **Form sections:** Break down complex forms into smaller, reusable sections for better organization and validation.

### Explain how to access the HttpContext object within an ASP.NET Core application.

In ASP.NET Core, there are two main ways to access the HttpContext object:

**Dependency Injection:** This is the preferred approach. Inject the IHttpContextAccessor service into your class through the constructor. Then, use _httpContextAccessor.HttpContext to access the current request context.

**Direct Access:** In controllers and Razor Pages, you can directly access HttpContext as a property. However, this is less flexible and tightly couples your code to the request context.

### DIP vs IOC vs DI

1. **Dependency Inversion Principle (DIP):**

- DIP is one of the SOLID principles of object-oriented design, proposed by Robert C. Martin. It states that high-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.
- In simpler terms, this principle suggests that classes should depend on abstractions rather than concrete implementations. This promotes loose coupling and makes the system easier to maintain and extend.

2. **Inversion of Control (IoC):**

- IoC is a design principle where the control of object creation or flow of control is inverted from the calling class to the framework or container. Instead of the application code controlling the flow, it delegates the control to an external system.
- IoC is often implemented using design patterns like the factory pattern, service locator pattern, or through frameworks like Spring in Java or AngularJS in JavaScript.

3. **Dependency Injection (DI):**

- DI is a technique used to implement IoC. It's a design pattern where the dependencies of a class are provided from the outside rather than being created within the class itself. This can be done through constructor injection, setter injection, or interface injection.
- By using DI, classes become more reusable, testable, and easier to maintain because they are decoupled from their dependencies.

In summary, the Dependency Inversion Principle is a design principle focusing on the abstraction level of dependencies, while Inversion of Control is a broader design principle that delegates control to an external system or framework. Dependency Injection is a specific technique used to implement Inversion of Control, where dependencies are provided externally rather than created internally. These concepts often work together to create flexible, modular, and maintainable software systems.

**app.Run vs. app.Use in ASP.NET Core Middleware Configuration**

| Feature | app.Run | app.Use |
|---|---|---|
| **Purpose** | Adds a terminal middleware delegate to the pipeline. | Adds a non-terminal middleware delegate to the pipeline. |
| **Execution** | Processes the request and ends the pipeline. | Processes the request and passes it to the next middleware in the pipeline. |
| **Use Case** | Suitable for simple applications with no further processing needed. | Suitable for complex applications requiring |

| Feature | app.Run | app.Use |
| --- | --- | --- |
| | | multiple processing steps. |
| **Example** | `app.Run(async context => await context.Response.WriteAsync("Hello World!"));` | `app.UseAuthentication(); app.UseAuthorization();` |
| **Result** | Sends the response directly to the client. | Modifies the request context and passes it to the next middleware. |
| **Flexibility** | Limited, only handles the request directly. | More flexible, and allows chaining multiple middleware components. |
| **Testability** | Can be challenging to test due to its terminal nature. | Easier to test in isolation as it is part of a larger pipeline. |

### How does ASP.NET Core handle static file serving?

ASP.NET Core doesn't serve static files like images, HTML, or CSS by default. Instead, it relies on the UseStaticFiles middleware to handle this task.

You configure this middleware to point to your static file folder, typically wwwroot.

Then, the middleware intercepts requests for these files and delivers them directly to the client, bypassing the entire ASP.NET Core pipeline.

This keeps things fast and efficient. Additionally, you can control caching and authorization for static files to further optimize your application.

### Explain Session and State Management options in ASP.NET Core.

In ASP.NET Core, Session and State Management refers to techniques for storing and maintaining data across multiple user requests. This data can be user-specific (like shopping cart items) or application-wide (like configuration settings).

Session state uses cookies to track users, while other options like cache or database can hold global state. Choosing the right approach depends on the type and persistence needs of your data.

### Explain Model Validation and how to perform custom validation logic.

Model validation ensures that data submitted to your application meets certain criteria before being processed. It's crucial for maintaining data integrity and preventing invalid or incomplete data from entering your system. ASP.NET Core provides built-in features

and libraries for model validation, but you can also implement custom logic for specific scenarios.

**Performing custom validation logic:**

Here are some ways to perform custom validation logic in ASP.NET Core:

- **Using IValidatableObject:** Inside the `Validate` method, you can perform your custom checks and add `ValidationResult` objects to the errors list. These results are then used to display error messages to the user.
- **Creating custom validation attributes:** You can inherit from the `ValidationAttribute` class and implement your custom logic in the `IsValid` method. This allows you to define reusable validation rules that can be applied to multiple model properties.
- **Using validation libraries:** Libraries like Fluent Validation provide an intuitive syntax for defining validation rules and error messages. You can create validation classes specific to your models and integrate them with the framework's validation system.

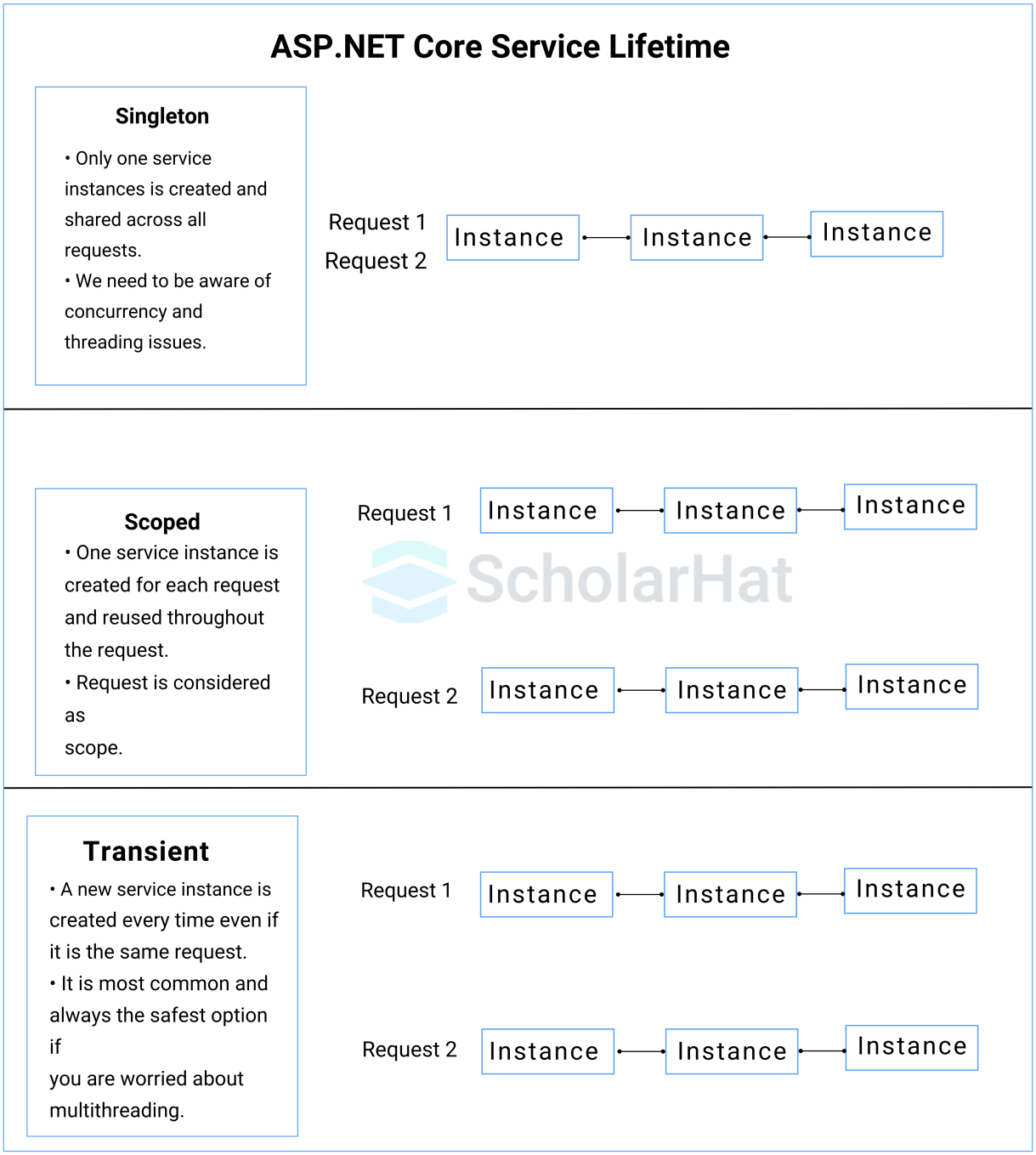
### Describe the concept of view models in ASP.NET Core MVC development.

In ASP.NET Core MVC development, ViewModels play a crucial role in separating data and presentation concerns. They act as a bridge between your domain models (entities representing business data) and the views (user interface).

ViewModels are custom classes specifically designed to represent the data required by a particular view. Unlike domain models, they are not directly tied to the database or business logic. They are lightweight and hold only the data relevant to that specific view, often combining information from multiple domain models or adding formatting or calculations for display purposes.

### Describe the different Service Lifetimes in ASP.NET Core.

In ASP.NET Core, service lifetimes define how long a particular instance of a service will be managed by the dependency injection (DI) container. Choosing the right lifetime for your services is crucial for optimizing performance, managing resources, and preventing memory leaks. Here are the three primary service lifetimes:




**Transient Lifetime:** A new instance of the service is created every time it's injected.

**Scoped Lifetime:** A single instance of the service is created per request scope (e.g., per HTTP request).

**Singleton Lifetime:** A single instance of the service is created for the entire lifetime of the application.

### Explain how Dependency Injection is implemented for controllers in ASP.NET Core MVC.

In ASP.NET Core MVC, controllers request their dependencies (like data access or service classes) explicitly through their constructors. This dependency injection

happens via a built-in Inversion of Control (IoC) container.

The container manages the creation and lifetime of these dependencies, injecting them into the controllers when needed. This keeps controllers loose-coupled, testable, and easily adaptable to changes.

Simply put, controllers tell the container what they need, and the container provides it automatically.

### What is Forgery Attacks and How to combat it ?

Cross-Site Request Forgery (CSRF) is a type of attack where unauthorized commands are transmitted from a user that the web application trusts. In ASP.NET MVC, this typically involves an attacker tricking a user into unknowingly submitting a request, such as changing their password or making a purchase, while authenticated on a different site.

**To combat CSRF attacks in MVC, you can implement anti-forgery tokens. Here's how it works:**

**Generate Tokens:** Include an anti-forgery token in your forms. This token is unique per session and per form, and it's validated on the server-side upon form submission.

**Use @Html.AntiForgeryToken():** In your view, include `@Html.AntiForgeryToken()` inside your `<form>` tag. This will generate a hidden input field with the token value.

**Validate the Token:** In your controller action method, decorate it with the `[ValidateAntiForgeryToken]` attribute. This attribute ensures that the token sent with the form matches the one stored in the server's session. If they don't match, the server rejects the request.

**Example:**

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult SubmitForm(FormViewModel model)
{
    // Handle form submission
}
```