1. Introduction to ASP.NET Core MVC

Basics of ASP.NET Core MVC

- **Question:** What is ASP.NET Core MVC, and how does it differ from ASP.NET MVC?
  Answer: ASP.NET Core MVC is a modern, lightweight, and modular framework for building web applications and APIs using the Model-View-Controller pattern. It differs from the classic ASP.NET MVC by being cross-platform, having better performance, and being fully integrated with ASP.NET Core, which supports cloud-based, microservices, and containerized environments.

Core Features of ASP.NET Core MVC

- **Question:** What are the key features of ASP.NET Core MVC?
  Answer: Key features include:
  - Cross-platform support (Windows, Linux, macOS).
  - Built-in dependency injection.
  - Razor Pages for simplified page-focused development.
  - Tag Helpers for cleaner HTML generation.
  - Built-in routing and middleware.
  - Strong integration with modern front-end frameworks and tools.

---

2. Middleware and Request Pipeline

Middleware in ASP.NET Core MVC

- **Question:** What is middleware in ASP.NET Core, and how does it work in the request pipeline?
  Answer: Middleware is a component that processes HTTP requests and responses in the ASP.NET Core request pipeline. Each middleware component can either handle a request, modify it, or pass it to the next middleware in the pipeline. Common middleware components include authentication, authorization, logging, and error handling.

Adding Middleware

- **Question:** How do you add middleware to the request pipeline in ASP.NET Core MVC?
  Answer: Middleware is added in the `Startup.Configure` method using extension methods such as `UseRouting`, `UseAuthentication`, `UseAuthorization`, and `UseEndpoints`. Middleware components are executed in the order they are added.

---

3. Routing in ASP.NET Core MVC

Conventional Routing

- **Question:** What is conventional routing in ASP.NET Core MVC, and how is it configured?
  Answer: Conventional routing defines URL patterns that map to controllers and actions. It is typically configured in the `Startup.Configure` method using `UseEndpoints()` and `MapControllerRoute()`.

```csharp
app.UseEndpoints(endpoints => { endpoints.MapControllerRoute( name: "default",
pattern: "{controller=Home}/{action=Index}/{id?}"); });
```

Attribute Routing

- **Question:** What is attribute routing, and how does it work in ASP.NET Core MVC?
  Answer: Attribute routing allows developers to define routes directly on controller actions using attributes. This provides more control and flexibility over route definitions and is useful for RESTful APIs.

```csharp
[Route("products/{id}")] public IActionResult ProductDetails(int id) { // Retrieve product details }
```

---

4. Controllers and Actions

Controllers in ASP.NET Core MVC

- **Question:** What is the role of a controller in ASP.NET Core MVC?
  Answer: A controller in ASP.NET Core MVC handles incoming HTTP requests, processes them (usually by interacting with a model), and returns an appropriate response. The response can be a view, JSON data, or an HTTP status code.

Action Results

- **Question:** What are the different types of action results in ASP.NET Core MVC?
  Answer: Action results represent the response from a controller action. Common action results include:
  - **ViewResult:** Renders a view to the response.
  - **JsonResult:** Returns JSON data.
  - **ContentResult:** Returns plain text or other content.
  - **RedirectResult:** Redirects to a different action or URL.
  - **FileResult:** Returns a file download.

Asynchronous Actions

- **Question:** How do you create asynchronous controller actions in ASP.NET Core MVC?
  Answer: Asynchronous actions are created using the `async` keyword and `await` keyword. This allows the action to handle I/O-bound operations without blocking the main thread.

```csharp
public async Task<IActionResult> GetData() { var data = await _dataService.GetDataAsync(); return View(data); }
```

---

5. Views and Razor Pages

Razor View Engine

- **Question:** What is the Razor view engine in ASP.NET Core MVC, and why is it used?
  Answer: The Razor view engine is a syntax for embedding C# code into HTML in a concise and readable way. It uses the `@` symbol to switch between HTML and C# code. Razor is commonly used to build dynamic views in ASP.NET Core MVC applications.

Strongly Typed Views

- **Question:** What are strongly typed views in ASP.NET Core MVC?
  Answer: Strongly typed views are views that are bound to a specific model type. This allows for IntelliSense, type checking, and more maintainable code when accessing model properties in the view.

```csharp
@model MyApp.Models.Product <h1>@Model.ProductName</h1>
```

Razor Pages

- **Question:** What are Razor Pages in ASP.NET Core, and how do they differ from MVC?
  Answer: Razor Pages are a page-based development model in ASP.NET Core, similar to MVC but with a simpler structure. Each page has a corresponding `.cshtml` file and a page model class. Razor Pages are more suited for scenarios where the focus is on pages rather than complex application logic and routing.

---

6. Model Binding and Validation

Model Binding

- **Question:** How does model binding work in ASP.NET Core MVC?
  Answer: Model binding automatically maps data from HTTP requests (form data, query strings, route data, etc.) to action method parameters or model properties. ASP.NET Core MVC handles type conversion and validation as part of the binding process.

Data Annotations

- **Question:** How do data annotations work for model validation in ASP.NET Core MVC?
  Answer: Data annotations are attributes applied to model properties to define validation rules. Common attributes include `[Required]`, `[StringLength]`, `[Range]`, and `[EmailAddress]`. ASP.NET Core MVC automatically validates models based on these attributes when an action method is invoked.

Custom Validation

- **Question:** How do you implement custom validation in ASP.NET Core MVC?
  Answer: Custom validation can be implemented by creating a custom attribute that inherits from `ValidationAttribute` and overriding the `IsValid` method.

```csharp
public class CustomAgeAttribute : ValidationAttribute { protected override
ValidationResult IsValid(object value, ValidationContext validationContext) { int
age = (int)value; return age >= 18 ? ValidationResult.Success : new
ValidationResult("Age must be 18 or older."); } }
```

---

7. Tag Helpers and HTML Helpers

Tag Helpers

- **Question:** What are Tag Helpers in ASP.NET Core MVC?
  Answer: Tag Helpers are server-side components that generate HTML dynamically. They provide an easier and more readable syntax compared to HTML Helpers, allowing developers to work with HTML and Razor syntax in a seamless way. Common Tag Helpers include `asp-action`, `asp-controller`, and `asp-for`.

HTML Helpers

- **Question:** What are HTML Helpers in ASP.NET Core MVC, and how do they work?
  Answer: HTML Helpers are methods used to generate HTML elements in Razor views. They are used to generate form elements like text boxes, checkboxes, and dropdowns. Examples include `Html.TextBoxFor()`, `Html.DropDownListFor()`, and `Html.ValidationMessageFor()`.

---

8. Dependency Injection in ASP.NET Core MVC

Built-in Dependency Injection

- **Question:** How is dependency injection implemented in ASP.NET Core MVC?
  Answer: ASP.NET Core MVC has built-in support for dependency injection (DI). Services are registered in the `Startup.ConfigureServices` method, and dependencies are injected into controllers or services using constructor injection.

```csharp
services.AddTransient<IDataService, DataService>();
```

Service Lifetimes

- **Question:** What are the different service lifetimes available in ASP.NET Core MVC?
  Answer: The three main service lifetimes in ASP.NET Core are:
  - **Transient:** A new instance is created every time the service is requested.
  - **Scoped:** A new instance is created per HTTP request.
  - **Singleton:** A single instance is created and shared across the application.

---

9. Authentication and Authorization

Authentication in ASP.NET Core MVC

- **Question:** How does authentication work in ASP.NET Core MVC?
  Answer: ASP.NET Core MVC uses middleware for authentication. Common authentication schemes include cookies, JWT Bearer tokens, and OAuth. You configure authentication in `Startup.ConfigureServices` using `services.AddAuthentication()` and define authentication schemes like `CookieAuthenticationDefaults.AuthenticationScheme`.

Authorization

- **Question:** How is role-based authorization implemented in ASP.NET Core MVC?
  Answer: Role-based authorization is implemented using the `[Authorize]` attribute. You can restrict access to certain actions or controllers based on user roles by specifying roles in the attribute.

```csharp
[Authorize(Roles = "Admin")] public IActionResult AdminDashboard() { // Only accessible to users with the "Admin" role }
```

---

10. Error Handling and Logging

Global Error Handling

- **Question:** How do you implement global error handling in ASP.NET Core MVC?
  Answer: Global error handling can be implemented using the `UseExceptionHandler()` middleware in the `Startup.Configure` method. This middleware catches unhandled exceptions and redirects them to a custom error page.

```
app.UseExceptionHandler("/Home/Error");
app.UseStatusCodePagesWithReExecute("/Home/StatusCode", "?code={0}");
```

Logging in ASP.NET Core MVC

- **Question:** How is logging implemented in ASP.NET Core MVC?
  Answer: Logging is built into ASP.NET Core and is implemented using the `ILogger` interface. You can configure logging providers (e.g., console, file, Azure Application Insights) in the `Startup.ConfigureServices` method. Logging can be injected into controllers and services via constructor injection.

```
public class HomeController { private readonly ILogger<HomeController> _logger;
public HomeController(ILogger<HomeController> logger) { _logger = logger; } public
IActionResult Index() { _logger.LogInformation("Index page accessed"); return
View(); } }
```

---

11. Asynchronous Programming in ASP.NET Core MVC

Asynchronous Action Methods

- **Question:** How do asynchronous action methods work in ASP.NET Core MVC?
  Answer: Asynchronous action methods use `async` and `await` to perform non-blocking operations, allowing the server to handle more requests. This is particularly useful for I/O-bound operations like database access or API calls.

```
public async Task<IActionResult> GetData() { var data = await
_dataService.GetDataAsync(); return View(data); }
```

Task-Based Asynchronous Pattern (TAP)

- **Question:** What is the Task-Based Asynchronous Pattern (TAP) in ASP.NET Core?
  Answer: TAP is the standard model for writing asynchronous code in .NET, where asynchronous methods return `Task` or `Task<T>`. It allows for better scalability by freeing up threads when waiting for I/O operations, enabling the server to handle more requests concurrently.

---

12. **Unit Testing in ASP.NET Core MVC**

Unit Testing Controllers

- **Question:** How do you unit test controllers in ASP.NET Core MVC?
  Answer: Unit testing controllers involves testing the action methods to ensure they return the correct `ActionResult`. You can use mocking frameworks like Moq to mock dependencies and isolate the controller's behavior.

```
[Fact] public void Index_ReturnsViewResult() { var mockService = new Mock<IDataService>(); var controller = new HomeController(mockService.Object); var result = controller.Index() as ViewResult; Assert.NotNull(result); }
```

Mocking Dependencies

- **Question:** How do you mock dependencies in ASP.NET Core MVC for unit testing?
  Answer: You can mock dependencies using a mocking framework like Moq. Mocking allows you to simulate the behavior of services and repositories, isolating the logic in the controller for unit testing.

```
var mockService = new Mock<IDataService>(); mockService.Setup(service => service.GetDataAsync()).ReturnsAsync(new List<Data>());
```

---

13. **Performance Optimization**

Caching in ASP.NET Core MVC

- **Question:** How do you implement caching in ASP.NET Core MVC?
  Answer: ASP.NET Core MVC supports caching through response caching, in-memory caching, and distributed caching. You can use the `[ResponseCache]` attribute to cache HTTP responses or `IMemoryCache` for in-memory caching of data.

```
[ResponseCache(Duration = 60)] public IActionResult Index() { return View(); }
```

Using `AsNoTracking`

- **Question:** What is `AsNoTracking()` in ASP.NET Core MVC, and when should it be used?
  Answer: `AsNoTracking()` is used in Entity Framework Core to improve performance for read-only queries by disabling change tracking. It is ideal for scenarios where data is fetched without the intention to modify it.

```
var products = context.Products.AsNoTracking().ToList();
```

---

14. **Globalization and Localization**

Localization in ASP.NET Core MVC

- **Question:** How do you implement localization in ASP.NET Core MVC?
  Answer: Localization is implemented by adding resource files for different languages and configuring the localization services in `Startup.ConfigureServices`. You can use `IStringLocalizer` and `IViewLocalizer` to retrieve localized strings in views and controllers.

```
public void ConfigureServices(IServiceCollection services) {
services.AddLocalization(options => options.ResourcesPath = "Resources");
services.AddControllersWithViews() .AddViewLocalization()
.AddDataAnnotationsLocalization(); }
```

Globalization

- **Question:** How do you enable globalization in ASP.NET Core MVC?
  Answer: Globalization involves configuring the `RequestLocalizationMiddleware` to handle culture-specific data like date formats, numbers, and currencies. You can define supported cultures in the `Startup.Configure` method.

```
app.UseRequestLocalization(new RequestLocalizationOptions { DefaultRequestCulture =
new RequestCulture("en-US"), SupportedCultures = new List<CultureInfo> { new
CultureInfo("en-US"), new CultureInfo("fr-FR") }, SupportedUICultures = new
List<CultureInfo> { new CultureInfo("en-US"), new CultureInfo("fr-FR") } });
```

---

15. API Development with ASP.NET Core MVC

Building REST APIs

- **Question:** How do you build a REST API using ASP.NET Core MVC?
  Answer: REST APIs are built using controllers that return data (typically in JSON format) instead of views. You use `JsonResult` or `Ok()` methods to return responses, and routing is configured with attribute routing to match HTTP verbs (GET, POST, PUT, DELETE).

```
[HttpGet] public IActionResult GetAllProducts() { var products =
_productService.GetAll(); return Ok(products); }
```

API Versioning

- **Question:** How do you implement API versioning in ASP.NET Core?
  Answer: API versioning can be implemented using the `Microsoft.AspNetCore.Mvc.Versioning` package. You define API versions using attributes like `[ApiVersion("1.0")]` and `[Route("api/v{version:apiVersion}/[controller]")]`, and configure versioning in `Startup.ConfigureServices`.

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