ChatGPT

1. What is ASP.NET Core? Explain any four features of ASP.NET Core.
2. Draw the .NET Core architecture and explain any four components.
3. What is .NET Core? Explain any four characteristics of .NET Core.
4. Write and explain any five advantages of ASP.NET Core.
5. Write two examples of server rendered UI. Explain advantages and limitations of server rendered UI.
6. Explain client rendered UI. Discuss advantages and limitations of it.

ANSWER VERY DESCRIPTIVELY

1. What is ASP.NET Core? Explain any four features of ASP.NET Core.

ASP.NET Core is an open-source, cross-platform web framework developed by Microsoft. It is designed for building modern, scalable web applications and APIs. ASP.NET Core is a significant redesign of the original ASP.NET framework, with improvements in performance, flexibility, and portability. It can be run on Windows, macOS, and Linux, making it ideal for cloud-based applications and microservices architectures.

Four Features of ASP.NET Core:

1. **Cross-Platform:**
   ASP.NET Core can run on multiple platforms (Windows, Linux, and macOS) due to its decoupling from the traditional Windows-based ASP.NET runtime. This makes it a versatile choice for developers who want to build applications that are portable and scalable across different operating systems.
2. **Performance:**
   ASP.NET Core is optimized for performance and is one of the fastest web frameworks in the market. It is lightweight, and its architecture has been designed to reduce memory overhead and CPU usage, improving the overall speed of web applications.
3. **Modular and Lightweight:**
   ASP.NET Core is modular, allowing developers to use only the features they need. It doesn't come with unnecessary overhead, which ensures a more lightweight and faster application. The modularity allows easy inclusion of features like dependency injection, logging, and configuration.
4. **Unified Development Model:**
   ASP.NET Core unifies the development model for building web applications, APIs, and microservices. It supports MVC (Model-View-Controller), Razor Pages, Web API, SignalR, and more under a single framework. This unification streamlines development and reduces complexity.

---

2. Draw the .NET Core architecture and explain any four components.

The **.NET Core architecture** is based on a modular, cross-platform design, with an emphasis on performance and flexibility. Here's a basic overview of its architecture:

```
sql


  +-----------------------------------------------------+
  |                .NET Core Applications               |
  |   (Web apps, Desktop apps, Mobile apps, etc.)       |
  +-------------------+---------------------------------+
                      |
          +-----------+-------------+
          |                         |
     +---+---+                  +---+---+
     | ASP.NET Core  |          |  .NET Core Runtime  |
     +-------+-------+          +---------------------+
             |                          |
  +----------+----------+      +-------+------+
  | Core Libraries      |      | CLR/Execution Engine |
  | (e.g., System.IO,   |      | (JIT Compiler, Garbage|
  | System.Net, etc.)   |      | Collector, etc.)      |
  +---------------------+      +-----------------------+
                         |                  |
                    +----------------------+  +--------------------+
                    |  .NET Standard APIs |  |   .NET Core SDK    |
                    | (Shared across all  |  | (Development Tools|
                    | .NET platforms)     |  | e.g., CLI tools,  |
                    |                     |  | Debugger, etc.)   |
                    +---------------------+  +--------------------+
```

Four Key Components of .NET Core Architecture:

1. **.NET Core Applications:** .NET Core applications include web applications (ASP.NET Core), desktop applications, console applications, and microservices. These applications target different platforms such as Windows, Linux, and macOS.
2. **.NET Core Runtime:** The runtime includes the **Common Language Runtime (CLR)**, which is responsible for executing .NET applications. It provides essential services such as garbage collection, exception handling, and just-in-time (JIT) compilation. It also ensures that .NET Core applications can run on different platforms.
3. **ASP.NET Core:** ASP.NET Core is the web framework for building scalable, high-performance web applications and APIs. It provides components like MVC, Web API, SignalR for real-time communication, and Razor Pages for web pages.
4. **Core Libraries:** .NET Core includes a set of foundational libraries for common application needs like **I/O operations**, **networking**, **data serialization**, **security**, and **database connectivity**. These libraries provide a robust framework to build applications in .NET Core.

---

3. What is .NET Core? Explain any four characteristics of .NET Core.

.NET Core is an open-source, cross-platform framework for building modern applications. It is a lightweight, high-performance runtime that enables the development of web, desktop, mobile, cloud, and IoT applications. .NET Core is the foundation of the .NET ecosystem and supports multiple programming languages, including C#, F#, and Visual Basic.

Four Characteristics of .NET Core:

1. **Cross-Platform:** One of the primary advantages of .NET Core is its cross-platform nature. It runs on multiple operating systems, including Windows, Linux, and macOS, which allows developers to build applications that can be deployed across diverse environments.
2. **Modular and Lightweight:** .NET Core is designed to be modular. Developers can include only the necessary components for their applications, reducing the overall size and increasing performance. Its modularity ensures that applications are lightweight and efficient.
3. **High Performance:** .NET Core is optimized for performance. It includes features such as Just-In-Time (JIT) compilation, garbage collection, and memory management, which help to ensure applications run efficiently. Benchmark studies have shown that .NET Core outperforms many other frameworks in terms of speed.
4. **Open Source:** .NET Core is open-source and has a large, active community. This allows developers to contribute to the framework, ensuring that it evolves with the needs of modern development. The source code is available on GitHub, and Microsoft actively maintains it, making it more reliable and transparent.

---

4. Write and explain any five advantages of ASP.NET Core.

Five Advantages of ASP.NET Core:

1. **Cross-Platform Support:** ASP.NET Core can be deployed and run on Windows, macOS, and Linux. This cross-platform support enables developers to build applications that can run on various operating systems without rewriting the code.
2. **Improved Performance:** ASP.NET Core is designed for performance. The framework is lightweight and optimized for speed. By reducing the overhead compared to traditional ASP.NET, ASP.NET Core applications typically perform faster, which is especially beneficial in cloud-based environments or for handling high traffic.
3. **Unified Development Model:** ASP.NET Core unifies different aspects of web development under one framework, such as MVC (Model-View-Controller), Razor Pages, Web API, and SignalR for real-time web applications. Developers can use a single framework for building different types of applications, streamlining development.
4. **Dependency Injection (DI):** ASP.NET Core has built-in support for Dependency Injection, which promotes loose coupling and makes the application easier to test and maintain. This is a key feature for building scalable, maintainable, and testable applications.
5. **Built-in Support for Cloud-Based Development:** ASP.NET Core is designed with cloud-native applications in mind. It supports integration with cloud services like Azure, and features like app configuration, logging, and monitoring are easily configurable for cloud environments.

---

5. Write two examples of server-rendered UI. Explain advantages and limitations of server-rendered UI.

Two Examples of Server-Rendered UI:

1. **ASP.NET Core MVC:** In ASP.NET Core MVC, views are rendered on the server using the Razor templating engine. The server generates HTML content and sends it to the browser.
2. **Server-Side Blazor:** Blazor is a framework from Microsoft for building interactive web UIs. In **Server-Side Blazor**, the UI components are executed on the server, and only UI updates are sent to the client over a SignalR connection.

Advantages of Server-Rendered UI:

1. **SEO Optimization:** Server-rendered UIs can be indexed by search engines because the content is generated on the server and delivered as fully-rendered HTML. This is important for applications where SEO is a concern.
2. **Faster Initial Page Load:** Since the server generates the full HTML page, the browser can start rendering the page immediately, leading to a faster initial load compared to client-side rendering.

Limitations of Server-Rendered UI:

1. **Higher Latency:** Every user interaction requires a round trip to the server, leading to increased latency, especially in scenarios where the network connection is slow or unreliable.
2. **Scalability Concerns:** Since each user request requires a server round-trip for rendering, this can put a strain on server resources. For applications with high traffic, this can affect performance and scalability.

---

6. Explain client-rendered UI. Discuss advantages and limitations of it.

Client-Rendered UI refers to a technique where the web application generates the user interface directly in the browser using JavaScript frameworks such as React, Angular, or Vue.js. Instead of the server generating the HTML content, the client (browser) requests data from the server (usually via an API) and renders the UI dynamically on the user's machine.

Advantages of Client-Rendered UI:

1. **Faster Interactivity:** Once the client-side JavaScript has loaded, UI updates can be rendered locally, without needing to communicate with the server. This provides a snappy, responsive user experience for actions like clicking buttons or navigating between pages.
2. **Reduced Server Load:** Since the client handles rendering and interactions, the server only needs to provide data via APIs, reducing the processing load on the server and allowing it to handle more users.
3. **Better User Experience (UX):** Client-rendered UIs often offer smoother transitions, animations, and more interactive features, leading to a better user experience overall. This is why Single Page Applications (SPAs) are so popular for modern web apps.

Limitations of Client-Rendered UI:

1. **SEO Challenges:** Client-rendered UIs can be difficult for search engines to index properly since the content is often generated dynamically using JavaScript. This can negatively impact SEO unless techniques like server-side rendering (SSR) or static rendering are used.

2. **Longer Initial Load:** The browser needs to download and execute all the necessary JavaScript, which can increase the initial loading time, especially for larger apps or slower network connections.
3. **JavaScript Dependency:** The functionality of client-rendered UIs depends heavily on JavaScript. If the user disables JavaScript or encounters compatibility issues, the application may fail to load or behave incorrectly.

7. What are Blazor server and Blazor Web Assembly? Explain.
8. Write and explain commands to create ASP.NET Core MVC project.
9. Write and explain content of launchSettings.json file.
10. With necessary code snippet, explain endpoint and HTML rendering.
11. Step-by-step explain ASP.NET code snippet for controller and view to demonstrate use of “asp-action

7. What are Blazor Server and Blazor WebAssembly? Explain.

Blazor is a framework from Microsoft that allows you to build interactive web UIs using C# instead of JavaScript. Blazor runs on top of the .NET platform, enabling developers to write client-side logic in C# rather than JavaScript. Blazor has two main hosting models: **Blazor Server** and **Blazor WebAssembly**.

Blazor Server:

Blazor Server is a hosting model where the application’s components run on the server, and the UI is rendered on the client in the browser via a SignalR connection. In this model, only UI updates and events (like clicks or form submissions) are sent over the network to the server, which then processes the logic and sends back HTML to update the UI in the browser.

How it works:

- The Blazor Server app runs on the server and maintains a real-time WebSocket connection to the client.
- The UI is rendered on the server, and changes to the UI (like button clicks or form submissions) are sent to the server over a SignalR connection.
- The server computes the UI changes, and only the HTML is sent back to the client to update the DOM.

Advantages:

- Faster startup because only a small Blazor runtime needs to be downloaded.
- It can be used with older browsers because it doesn't rely on WebAssembly.

Limitations:

- Dependent on the network; if the connection is slow or interrupted, it can affect the user experience.
- Heavy reliance on server resources as the app’s logic is executed on the server.

Blazor WebAssembly (WASM):

Blazor WebAssembly allows running Blazor applications directly in the browser using WebAssembly (WASM). In this model, the entire application, including .NET runtime and all required libraries, is downloaded to the client's browser. Once downloaded, the app executes in the browser without requiring a connection to the server, except for backend API calls.

How it works:

- The entire application, including .NET runtime, is compiled into WebAssembly.
- The Blazor WebAssembly app runs directly in the browser, with the logic and UI rendering happening client-side.
- The client sends requests (e.g., for data) to the server via APIs, but UI interactions and logic are processed locally.

Advantages:

- No dependency on the server after the initial load; users can interact with the application without a persistent connection.
- Works offline after loading (with proper offline support).

Limitations:

- Larger download size for the initial page load since the .NET runtime and app code need to be downloaded to the browser.
- The performance is limited by the browser's WebAssembly support, which may not be as fast as native execution in some cases.

---

8. Write and explain commands to create an ASP.NET Core MVC project.

To create an ASP.NET Core MVC project, you can use the **.NET CLI (Command Line Interface)**. Below are the steps and commands to create an MVC application.

1. **Open Command Line/Terminal** and navigate to the directory where you want to create the project.
2. **Create the project** using the following command:

   ```
   bash


     dotnet new mvc -n MyMvcApp
   ```

   Explanation:

   - `dotnet new`: This is the command to create a new project.
   - `mvc`: This specifies the template for an MVC project. MVC stands for Model-View-Controller, which is a design pattern for building web applications.
   - `-n MyMvcApp`: This specifies the name of the project folder. In this case, it creates a folder called `MyMvcApp` and places the project inside it.
3. **Change to the project directory**:

```bash
cd MyMvcApp
```

4. **Restore dependencies** (if required):

```bash
dotnet restore
```

This will restore any NuGet packages required by the project.

5. **Run the application**:

```bash
dotnet run
```

This command will build and start the web server. By default, it will run on `http://localhost:5000` (unless otherwise configured).

---

9. Write and explain the content of `launchSettings.json` file.

The `launchSettings.json` file is a configuration file used by ASP.NET Core to define settings related to how the application is launched and debugged during development. It is located in the **Properties** folder of an ASP.NET Core project.

Example content of `launchSettings.json`:

```json
{
  "iisSettings": {
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 44300
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "MyMvcApp": {
      "commandName": "Project",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000"
    }
```

```
    }
}
```

Explanation of the fields:

- `iisSettings`: Contains settings for IIS Express, a lightweight web server used during development.
  - `applicationUrl`: The base URL where the application will be hosted when running under IIS Express (e.g., `http://localhost:5000`).
  - `sslPort`: The port used for HTTPS (if configured).
- `profiles`: Contains profiles for different ways to launch the application.
  - `IIS Express`: Specifies the settings for launching the app using IIS Express, which is the default option in Visual Studio.
  - `MyMvcApp`: Defines the profile for running the application directly with the .NET Core development server (Kestrel).
    - `commandName`: Can be `IISExpress` or `Project`. `Project` means running the project using the Kestrel server.
    - `environmentVariables`: Defines the environment variable `ASPNETCORE_ENVIRONMENT` to set the environment to `Development`.
    - `applicationUrl`: Specifies the URL where the app will be available (e.g., `http://localhost:5000`).

---

10. With necessary code snippet, explain endpoint and HTML rendering.

In ASP.NET Core, **endpoints** define the routes that the application responds to. They are configured in the `Startup.cs` file, within the `Configure` method, and map HTTP requests to appropriate logic, such as MVC actions or Razor Pages.

Code snippet for configuring an endpoint in `Startup.cs`:

```csharp


public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews(); // Register MVC services
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        // Configure MVC endpoints
        app.UseRouting();
```

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

Explanation:

- `app.UseRouting()`: This enables the routing system to match incoming HTTP requests to the appropriate controller action.
- `app.UseEndpoints()`: This defines the endpoints in your app. The `MapControllerRoute` method maps incoming requests to a controller action. In this case, the pattern specifies that requests with the path `/Home/Index` will be routed to the `Index` action of the `HomeController`.

HTML Rendering in MVC:

In the `HomeController`, you might have an action method that returns a view (HTML) like this:

```
csharp


  public class HomeController : Controller
  {
      public IActionResult Index()
      {
          return View(); // Renders the Index.cshtml view
      }
  }
```

The `View()` method in the controller looks for a corresponding Razor view file (e.g., `Index.cshtml`) in the `Views/Home` folder. The `Index.cshtml` file contains HTML markup, and Razor syntax for dynamic content.

```
html


  @{
      ViewData["Title"] = "Home Page";
  }

  <div>
      <h1>@ViewData["Title"]</h1>
      <p>Welcome to the ASP.NET Core MVC application!</p>
  </div>
```

This Razor view will generate the following HTML when rendered:

```
html


  <!DOCTYPE html>
  <html>
  <head>
      <title>Home Page</title>
  </head>
  <body>
```

```
    <h1>Home Page</h1>
    <p>Welcome to the ASP.NET Core MVC application!</p>
</body>
</html>
```

---

11. Step-by-step explain ASP.NET code snippet for controller and view to demonstrate use of “asp-action”

In ASP.NET Core MVC, the `asp-action` tag helper is used to generate the correct action URL for a specific action method in a controller. This helps to create links or form actions dynamically based on the controller and action names.

Step 1: Define a Controller Action

Create a controller (e.g., `HomeController`) with an action method called `Contact`:

```csharp
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }

    public IActionResult Contact()
    {
        return View();
    }
}
```

Step 2: Create a View with `asp-action`

In the `Index.cshtml` view, you can use the `asp-action` tag helper to link to the `Contact` action:

```html
<a asp-action="Contact">Contact Us</a>
```

The `asp-action="Contact"` generates a link to the `Contact` action method of the `HomeController`.

Step 3: Running the Application

When you run the application and navigate to the `Index` view, the link "Contact Us" will point to `/Home/Contact`, and clicking on it will navigate to the `Contact` action method in the `HomeController`.

This demonstrates the use of the `asp-action` tag helper to dynamically generate URLs for actions, improving maintainability and ensuring correct URLs are always generated.

13. Create necessary classes and demonstrate use of @model. Also discuss required ASP.NET code.
14. With suitable example code snippet of controller, model, and view, explain validation.
15. Develop ASP.NET Core MVC view which uses Bootstrap. With code snippet, explain any five classes of bootstap.css.
16. Create sample Program.cs file for ASP.NET Core MVC project and explain any three statements from it.

13. Create necessary classes and demonstrate the use of `@model`. Also, discuss the required ASP.NET code.

In ASP.NET Core MVC, the `@model` directive in a Razor view is used to specify the type of data that the view will work with. It links the view to a specific model, enabling the view to display data from that model.

Step-by-step Example:

1. Create the Model Class:

A model class defines the data structure that you pass to a view.

```csharp
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

This `Product` class represents a product with an `Id`, `Name`, and `Price`.

2. Create the Controller:

The controller fetches the data and passes it to the view.

```csharp
public class ProductController : Controller
{
    public IActionResult Index()
    {
        // Example product data
        var product = new Product
        {
            Id = 1,
            Name = "Laptop",
            Price = 999.99m
        };

        return View(product);
```

```
    }
}
```

In the `Index` action, a `Product` object is created and passed to the view.

3. Create the View:

In the view (`Index.cshtml`), you use the `@model` directive to specify the model type for the data being passed.

```
html


 @model Product

 <!DOCTYPE html>
 <html>
 <head>
     <title>Product Details</title>
 </head>
 <body>
     <h2>Product Information</h2>
     <p><strong>ID:</strong> @Model.Id</p>
     <p><strong>Name:</strong> @Model.Name</p>
     <p><strong>Price:</strong> @Model.Price</p>
 </body>
 </html>
```

The `@model` directive specifies that the view expects a `Product` object. In the body, the `@Model` is used to access the properties of the `Product` object passed from the controller.

ASP.NET Core MVC Code Recap:

- **Model (`Product` class):** Defines the data structure.
- **Controller (`ProductController`):** Retrieves the data and passes it to the view.
- **View (`Index.cshtml`):** Displays the data using the `@model` directive to bind the data.

---

14. With suitable example code snippet of controller, model, and view, explain validation.

In ASP.NET Core MVC, **validation** is typically done using **data annotations** in the model class. These annotations define rules for validating user input, and the validation occurs automatically when the form is submitted.

Step-by-step Example:

1. Create the Model Class with Validation:

You can use data annotations to validate properties in the model.

```
csharp


 using System.ComponentModel.DataAnnotations;

 public class User
 {
```

```
    public int Id { get; set; }

    [Required(ErrorMessage = "Name is required.")]
    [StringLength(50, ErrorMessage = "Name cannot be longer than 50 characters.")]
    public string Name { get; set; }

    [EmailAddress(ErrorMessage = "Invalid email address.")]
    public string Email { get; set; }

    [Range(18, 120, ErrorMessage = "Age must be between 18 and 120.")]
    public int Age { get; set; }
}
```

In this example:

- The `Required` attribute ensures that the `Name` field is not empty.
- The `StringLength` attribute limits the length of `Name` to 50 characters.
- The `EmailAddress` attribute ensures that the email is valid.
- The `Range` attribute restricts the `Age` field to values between 18 and 120.

2. Create the Controller:

The controller will handle the POST request, perform validation, and return the view with validation results.

```
csharp


public class UserController : Controller
{
    [HttpGet]
    public IActionResult Create()
    {
        return View();
    }

    [HttpPost]
    public IActionResult Create(User user)
    {
        if (ModelState.IsValid)
        {
            // Proceed with saving data
            return RedirectToAction("Success");
        }

        // If validation fails, return the same view with validation messages
        return View(user);
    }

    public IActionResult Success()
    {
        return View();
    }
}
```

- The `Create` GET action renders the form.
- The `Create` POST action validates the form data using `ModelState.IsValid`. If the data is valid, it proceeds to the `Success` view; otherwise, it returns the form with validation errors.

3. Create the View:

The view will render the form and display validation errors.

```
html


  @model User

  <form asp-action="Create" method="post">
      <div>
          <label>Name</label>
          <input type="text" asp-for="Name" />
          <span asp-validation-for="Name" class="text-danger"></span>
      </div>
      <div>
          <label>Email</label>
          <input type="text" asp-for="Email" />
          <span asp-validation-for="Email" class="text-danger"></span>
      </div>
      <div>
          <label>Age</label>
          <input type="number" asp-for="Age" />
          <span asp-validation-for="Age" class="text-danger"></span>
      </div>
      <button type="submit">Submit</button>
  </form>

  @section Scripts {
      <partial name="_ValidationScriptsPartial" />
  }
```

- `asp-for` binds the form input fields to the model properties.
- `asp-validation-for` displays the validation error messages associated with each field.
- The `_ValidationScriptsPartial` is used to include JavaScript for client-side validation.

Key Concepts in Validation:

- **Data Annotations:** Used to specify validation rules on model properties.
- **ModelState:** Used in the controller to check if the data submitted by the user is valid.
- **Client-Side Validation:** Enabled automatically when you include the validation scripts.

---

15. Develop ASP.NET Core MVC view which uses Bootstrap. With code snippet, explain any five classes of Bootstrap.css.

Bootstrap is a popular CSS framework that makes it easy to build responsive web applications. Below is an example of an ASP.NET Core MVC view using Bootstrap for styling.

Example View with Bootstrap:

```
html


  @{
      ViewData["Title"] = "Bootstrap Example";
  }

  <!DOCTYPE html>
  <html lang="en">
  <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <title>@ViewData["Title"]</title>
      <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.c
  ss" rel="stylesheet" />
```

```
</head>
<body>
    <div class="container mt-5">
        <h1 class="text-center">Welcome to Bootstrap Example</h1>

        <div class="row">
            <div class="col-md-6 offset-md-3">
                <div class="card">
                    <div class="card-body">
                        <h5 class="card-title">Card Title</h5>
                        <p class="card-text">This is a simple Bootstrap card with a body of
content.</p>
                        <a href="#" class="btn btn-primary">Go somewhere</a>
                    </div>
                </div>
            </div>
        </div>
    </div>

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/js/bootstrap.bundl
e.min.js"></script>
</body>
</html>
```

Explanation of Five Bootstrap Classes Used:

1. `container`:
   This class creates a responsive fixed-width container. It centers the content and applies appropriate margins and padding. It adapts to different screen sizes.
2. `mt-5`:
   This class applies a top margin of `5`, which corresponds to Bootstrap's spacing scale. It helps to create space above the element.
3. `row`:
   The `row` class is used as a container for columns. It ensures that columns are aligned properly and have appropriate spacing between them.
4. `col-md-6`:
   This class is used to define a column that takes up 6 out of 12 columns on medium-sized devices (tablet and above). This makes the content responsive.
5. `card`:
   The `card` class is used to create a flexible and extensible content container with a default style. Cards are useful for displaying content in a structured layout.

---

16. Create sample `Program.cs` file for ASP.NET Core MVC project and explain any three statements from it.

In ASP.NET Core 6 and later, the `Program.cs` file has a streamlined structure to configure services and middleware. Below is an example `Program.cs` file.

Sample `Program.cs`:

```csharp
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

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

app.Run();
```

Explanation of Three Statements:

1. **`var builder = WebApplication.CreateBuilder(args);`**:
   This line initializes a new **`WebApplicationBuilder`**, which is used to configure the application's services and middleware. It is the starting point for building the web application in ASP.NET Core 6+.
2. **`builder.Services.AddControllersWithViews();`**:
   This line adds the necessary services for an MVC application. It registers the MVC services that are needed for controller and view functionality in your application.
3. **`app.UseRouting();`**:
   This middleware component is used to route incoming HTTP requests to the appropriate route handler, such as controllers or Razor Pages. It is necessary to enable routing functionality in the application.