- **What is model?**

    - The Model component is responsible for managing the application's data and business logic.

    - It encapsulates the data and provides methods and operations for interacting with that data.

    - In the context of a .NET MVC application, the Model typically consists of classes and objects that represent the application's data structures and business rules.

        - ✓ For example, if you are building a web application to manage a list of products, the Model might include classes that define the structure of a product (e.g., Product class) and methods to perform operations on products (e.g., add, update, delete).

- **What is routing in MVC?**

    - Routing in MVC (Model-View-Controller) refers to the process of mapping incoming HTTP requests to specific controller actions or methods in an MVC web application.

    - It plays a crucial role in determining how a web application responds to different URLs and requests from clients (typically web browsers).

    - In MVC, routing is used to define URL patterns and map them to specific controller actions.

- **What is the difference between Temp data, View, and View Bag?**

ViewData	ViewBag	TempData
It is Key-Value Dictionary collection	It is a type object	It is Key-Value Dictionary collection
ViewData is a dictionary object and it is property of ControllerBase class	ViewBag is Dynamic property of ControllerBase class.	TempData is a dictionary object and it is property of controllerBase class.
ViewData is Faster than ViewBag	ViewBag is slower than ViewData	NA
ViewData is introduced in MVC 1.0 and available in MVC 1.0 and above	ViewBag is introduced in MVC 3.0 and available in MVC 3.0 and above	TempData is also introduced in MVC1.0 and available in MVC 1.0 and above.
ViewData also works with .net framework 3.5 and above	ViewBag only works with .net framework 4.0 and above	TempData also works with .net framework 3.5 and above
Type Conversion code is required while enumerating	In depth, ViewBag is used dynamic, so there is no need to type conversion while enumerating.	Type Conversion code is required while enumerating
Its value becomes null if redirection has occurred.	Same as ViewData	TempData is used to pass data between two consecutive requests.
It lies only during the current request.	Same as ViewData	TempData only works during the current and subsequent request.

- **What is difference between MVC and Web Forms?**

ASP.NET MVC	Web Forms
MVC focuses on separation of concern, i.e., there is no fixed code behind page for every view. A view can be called from multiple action.	Web form based on functions and page behind code, i.e., there is code behind page for each view. You have to write code in that class related to this view only.
First request comes to controller and action, then view gets called.	First request comes to Page (View) then it will go to code behind page.
MVC provides HTML Helpers to create form controls. This is optional. You can use simple HTML controls also.	For everything in webforms, you have a server control.
There is no *ViewState* for state management in View.	*Viewstate* is used to maintain state of form in view. This also makes page heavy.
Good for SEO friendly urls. No need to map to existing physical files.	Earlier, this feature was not available in Webforms but now it is available. Although it is not that easy and it is optional.
We create partial views for reusable views.	We create user controls for reusable view or control.

It is very easy to use jquery and JavaScripts. Using CSS is also easy	It is a little difficult to use jquery and JavaScripts in web forms. It provides themes and it is difficult to manage design of server controls.
Maintaining Id of form controls are easy and you can fully control them when working with Jquery.	It is difficult to manage Id of server controls, you don't know object id of server controls most of the time. Especially working with user controls.

- **What is session? What is the default time for session?**
    - In the context of ASP.NET MVC (Model-View-Controller), a session refers to a way to store and manage user-specific data across multiple requests from the same user during their interaction with a web application.
    - Sessions are commonly used to maintain user state and store information that needs to persist throughout a user's visit to a website.
    - This can include user authentication status, shopping cart contents, and other user-specific data.
    - The default time for a session in ASP.NET MVC is typically set to 20 minutes.
    - This means that if a user does not send a request to the server within 20 minutes, their session data will expire, and they will need to log in again or reestablish their session if applicable.
    - The session timeout can be configured in the web.config file of your ASP.NET MVC application using the `<sessionState>` element within the `<system.web>` section.

- **What is Partial View in MVC? With example**

  - In ASP.NET MVC, a Partial View is a reusable and self-contained portion of a web page that can be rendered within a parent view.

  - Partial Views are a way to modularize and encapsulate parts of your user interface, making your code more maintainable and allowing you to reuse common UI components across different views.

  - Here's an example of how to create and use a Partial View in ASP.NET MVC:

1. Create a Partial View:
   - ✓ Let's say you want to create a Partial View for displaying a list of products. First, create a new Partial View file (e.g., "_ProductList.cshtml") in your Views folder.
   - ✓ The name of the file typically starts with an underscore to indicate that it's a partial view.

_ProductList.cshtml:

```
html
@model List<Product>

<ul>
   @foreach (var product in Model)
   {
      <li>@product.Name - $@product.Price</li>
   }
</ul>
```

   - ✓ In this example, the Partial View takes a list of `Product` objects as its model and displays them in an unordered list.

2. Use the Partial View in a Parent View:
   - ✓ Now, you can use the Partial View in a parent view (e.g., "Index.cshtml") by rendering it using the `Html.Partial` or `Html.RenderPartial` helper methods.

Index.cshtml:
html
<h1>Product List</h1>
@Html.Partial("_ProductList", Model.Products)

- ✓ In this example, the `Html.Partial` method is used to render the "_ProductList" Partial View, passing it the list of products from the model.

3. Controller and Model:
   - ✓ Make sure you have a controller action that populates the model with the necessary data.
   - ✓ For example:

csharp
```
public class ProductController : Controller
{
    public ActionResult Index()
    {
        var products = GetProductListFromDatabase();
        return View(products);
    }

    private List<Product> GetProductListFromDatabase()
    {
        // Logic to retrieve product data from the database
        // This is just a placeholder; you should implement your data retrieval logic here.
        var productList = new List<Product>
        {
            new Product { Id = 1, Name = "Product 1", Price = 10.99 },
            new Product { Id = 2, Name = "Product 2", Price = 19.99 },
            new Product { Id = 3, Name = "Product 3", Price = 5.99 }
        };

        return productList;
    }
```

}

4. Routing:
   - ✓ Ensure that your routing is set up properly so that when you navigate to the "Product/Index" URL, it maps to the "Index" action in your `ProductController`.
   - ✓ With these steps, when you navigate to the "Product/Index" URL, the "Index" view will be displayed, including the Partial View "_ProductList" that shows the list of products.
   - ✓ This approach allows you to reuse the same product list rendering logic in multiple views throughout your application.

- **What is the difference between View and Partial View?**

View	**Partial View**
View contains the layout page	Partial view does not contain the layout page
_viewstart page is rendered before any view is rendered	Partial view does not check for a _viewstart.cshtml. We cannot place any common code for a partial view within the _viewStart.cshtml page.
View may have markup tags like html, body, head, title, meta etc.	The Partial view is specially designed to render within the view and as a result it does not contain any mark up.
Partial view is more lightweight than the view. We can also pass a regular view to the RenderPartial method.	
If there is no layout page specified in the view, it can be considered as a partial view. In razor, there is no distinction between views and partial views as in the ASPX view engine (aspx and ascx).	

- **Explain the concept of MVC Scaffolding?**

    - MVC scaffolding, often referred to as ASP.NET MVC scaffolding, is a code generation technique provided by ASP.NET MVC that simplifies and speeds up the process of creating basic CRUD (Create, Read, Update, Delete) operations for a data model in a web application.

    - It generates the necessary controller and views to perform these operations based on your data model.

    - Scaffolding is a valuable tool for quickly setting up the basic structure of your application when working with databases or data models.

    - Here's how MVC scaffolding works and the key components involved:

1. Data Model:
    - ✓ You start with a data model, typically represented as a C# class. This model defines the structure of your data, including properties and relationships.
    - ✓ For example, you might have a `Product` class representing products in an e-commerce application.

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

2. Scaffolding:
    - ✓ When you want to create CRUD functionality for this data model, you can use scaffolding.
    - ✓ In Visual Studio, you can right-click on your Controllers folder, choose "Add," and then select "Controller" with scaffolding options.

3. Controller Generation:
   - ✓ When you select scaffolding, ASP.NET MVC generates a controller for your data model.
   - ✓ This controller contains actions for Create, Read, Update, and Delete operations. It also includes the necessary logic to interact with your data source, typically a database.

   - ✓ For example, if you scaffold a controller for the `Product` model, it would generate actions like `Index` (for listing products), `Create` (for adding new products), `Edit` (for updating products), and `Delete` (for removing products).

4. View Generation:
   - ✓ Along with the controller, scaffolding generates views for each of the CRUD operations.
   - ✓ These views are HTML templates that provide the user interface for interacting with the data.
   - ✓ For instance, the "Create" view includes input fields for entering product information.

5. Routing:
   - ✓ Scaffolding also configures the necessary routes in your application's routing system.
   - ✓ These routes map URLs to the corresponding controller actions.
   - ✓ For example, a URL like "/Products/Create" would map to the "Create" action in the `ProductsController`.

6. Customization:
   - ✓ While scaffolding provides a basic set of views and actions, you can customize them as needed.
   - ✓ You can modify the generated views to match your application's styling or add validation logic to the controller actions.

7. Data Source:
   - ✓ You need to connect the scaffolding-generated controller to your data source, which is typically a database. You can use Entity Framework or another data access technology to handle data operations.

8. Security and Authorization:
   - ✓ Scaffolding often provides basic functionality, but you should also implement security and authorization checks to ensure that only authorized users can perform CRUD operations.
   - ✓ This may involve adding authentication and authorization attributes to your controller actions.

- MVC scaffolding is a powerful tool for quickly setting up the foundation of a web application, especially when dealing with database-backed models.
- It saves time and reduces repetitive coding tasks by generating the boilerplate code needed for CRUD operations, allowing developers to focus on more complex aspects of their application.

- **How to change time of session?**

  - In ASP.NET, you can change the time of a session by adjusting the session timeout value in your application's configuration.

  - The session timeout determines how long a user's session remains active without any activity before it expires.

  - By default, it's set to 20 minutes, but you can change it to a value that suits your application's needs.

  - Here's how you can change the session timeout in an ASP.NET application:

1. Update `web.config`:

   - ✓ Open the `web.config` file of your ASP.NET application.
   - ✓ This file is typically located in the root directory of your application.

2. Find the `<sessionState>` Element:

   - ✓ Inside the `<system.web>` section of the `web.config` file, look for the `<sessionState>` element.

- ✓ It defines various session-related settings, including the session timeout.
- ✓ Here's an example of what it might look like:

xml
```
<system.web>
  <!-- Other configuration settings -->

  <sessionState mode="InProc" timeout="20" />
</system.web>
```

- ✓ In this example, the `timeout` attribute is set to 20 minutes, which is the default value.

3. Change the Timeout Value:

- ✓ To change the session timeout to a different value, simply update the `timeout` attribute to the desired number of minutes.
- ✓ For example, to set a session timeout of 30 minutes, you would change it like this:

xml
```
<sessionState mode="InProc" timeout="30" />
```

- ✓ You can adjust the timeout value according to your application's requirements.

4. Save the `web.config` File:

- ✓ After making the change, save the `web.config` file.

5. Restart the Application:

- ✓ Depending on your hosting environment, you may need to restart your application for the changes to take effect.
- ✓ This can involve stopping and starting your application pool in IIS or recycling the application in a hosting environment like Azure App Service.

- Keep in mind that increasing the session timeout means that session data will be stored on the server for a longer period, potentially consuming more server resources.

- You should balance the session timeout value based on your application's security and performance requirements.

- Additionally, longer session timeouts can impact the user experience if users are expected to stay active on your site for extended periods without interaction.

- **What is query string? What are disadvantages of query string?**

  - A query string is a part of a URL (Uniform Resource Locator) that is used to pass data to a web server as key-value pairs.

  - It is typically added to the end of a URL and starts with a question mark (`?`), followed by one or more key-value pairs separated by ampersands (`&`).

  - Each key-value pair consists of a parameter name (key) and its corresponding value.

  - For example, consider the following URL with a query string:

    https://www.example.com/search?q=asp.net&page=1

  - In this URL, the query string is `?q=asp.net&page=1`, and it contains two key-value pairs:

  - `q=asp.net`: This pairs the parameter "q" with the value "asp.net," indicating a search query for "asp.net."

- `page=1`: This pairs the parameter "page" with the value "1," indicating the current page number.

- Query strings are commonly used for various purposes in web applications, such as:

1. Filtering and Searching: Passing search queries, filter criteria, or search parameters to a web page or API.

2. Pagination: Specifying the current page number when displaying paginated results.

3. Tracking: Including tracking information or campaign identifiers in URLs.

- While query strings are a convenient way to transmit data, they have some disadvantages:

1. Security:
   - ✓ Query string parameters are visible in the URL, which can pose a security risk.
   - ✓ Sensitive data, such as passwords or authentication tokens, should never be passed in a query string because they can be easily intercepted and exposed.

2. Limited Data Size:
   - ✓ Query strings are limited in size, and there's a practical limit to the amount of data you can pass through them.
   - ✓ Exceeding this limit can lead to URL truncation or data loss.

3. Bookmarking Issues:
   - ✓ If query strings are used for important application state, bookmarking URLs can become problematic.
   - ✓ Users who bookmark a URL with a query string may not get the same result if the application state changes.

4. SEO and User-Friendly URLs:
   - ✓ Query strings can make URLs less human-readable and less SEO-friendly. In contrast, using route parameters or URL segments (e.g.,

`/products/123`) is often preferred for user-friendliness and search engine optimization.

5. Caching:
   - ✓ Some proxy servers or caching mechanisms may not cache URLs with query strings effectively, potentially affecting performance.

6. Maintainability:
   - ✓ As the number of parameters in a query string increases, managing and parsing them in the server-side code can become complex and error-prone.

   - o To address some of these disadvantages, web developers often use other techniques for transmitting data, such as using route parameters, form submissions, or storing data in cookies or session variables. The choice of method depends on the specific requirements and constraints of the application.

- **What is cookie? What are limitations for cookie?**

  - o A cookie is a small piece of data that a web server sends to a user's web browser during their visit to a website.

  - o The browser stores this data and sends it back to the server with subsequent requests.

  - o Cookies are often used to track user sessions, store user preferences, and enable various functionalities on websites.

  - o Here are some key characteristics and components of cookies:

1. Name-Value Pair:
   - ✓ A cookie consists of a name-value pair, where the name is a string identifier, and the value is the associated data.

- ✓ For example, a cookie might have the name "username" and the value "john_doe."

2. Domain and Path:
   - ✓ Cookies can be associated with a specific domain and path on a website.
   - ✓ This allows cookies to be scoped to particular sections of a site or even subdomains.
   - ✓ For example, a cookie set for the domain "example.com" would be accessible to all pages on that domain.

3. Expiration Time:
   - ✓ Cookies can have an expiration time or date, after which they are no longer valid and are automatically deleted by the browser.
   - ✓ Persistent cookies may have a long expiration time, while session cookies are stored only for the duration of the user's session.

4. Secure and HttpOnly Flags:
   - ✓ Cookies can include flags to enhance security.
   - ✓ The "Secure" flag restricts the transmission of cookies to secure (HTTPS) connections only.
   - ✓ The "HttpOnly" flag prevents client-side scripts from accessing the cookie data, enhancing protection against certain types of attacks.

5. Size Limitations:
   - ✓ Cookies have size limitations.
   - ✓ Different browsers have varying maximum sizes for cookies, but in general, cookies are limited to a few kilobytes (usually a few thousand bytes) per cookie.
   - ✓ This limitation can impact the amount of data that can be stored in a single cookie.

6. Quantity Limitations:
   - ✓ Browsers also impose limitations on the total number of cookies that can be stored for a single domain.
   - ✓ This limit is typically in the range of several dozen to a few hundred cookies per domain.

7. Security and Privacy Concerns:

- ✓ Cookies can raise privacy concerns because they can be used to track user behavior across websites.
- ✓ In response to privacy concerns, browsers provide options for users to manage and block cookies.
- ✓ Additionally, various regulations (e.g., GDPR) require websites to inform users about their cookie usage and obtain consent for certain types of cookies.

8. Cross-Origin Restrictions:
   - ✓ Cookies are subject to the Same-Origin Policy, which restricts their access to pages from the same origin (i.e., the same domain, protocol, and port).
   - ✓ This policy prevents one website from reading or modifying cookies set by another website.

9. Data Sensitivity:
   - ✓ Cookies can potentially store sensitive information.
   - ✓ For security reasons, sensitive data such as passwords, credit card numbers, or personal identification should never be stored in cookies. Instead, sensitive data should be stored securely on the server, and only a reference (e.g., a session identifier) should be stored in a cookie.

10. Browser Support:
    - ✓ While cookies are supported by virtually all web browsers, not all users may have cookies enabled.
    - ✓ Therefore, websites should be designed to function properly even when cookies are disabled or blocked.

- o In summary, cookies are a fundamental mechanism for maintaining state and user interactions on the web.

- o However, they have limitations in terms of size, quantity, security, and privacy, which should be considered when designing web applications that rely on them.

- Developers should use cookies responsibly and be mindful of user privacy and security concerns.

- **Create one example to store data in session and show on other view.**

  - To store data in a session and then retrieve and display it on another view in an ASP.NET MVC application, you can follow these steps:

1. Create a New MVC Project:
   - ✓ Start by creating a new ASP.NET MVC project in Visual Studio or your preferred development environment.

2. Create a Controller:
   - ✓ Create a controller that will handle the data storage and retrieval. For this example, let's create a controller named "HomeController."

```
csharp
public class HomeController : Controller
{
    // Action to display the initial view
    public ActionResult Index()
    {
        return View();
    }

    // Action to store data in the session
    [HttpPost]
    public ActionResult StoreData(string data)
    {
        // Store the data in the session
        Session["UserData"] = data;

        return RedirectToAction("DisplayData");
    }
```

```
    // Action to display the data from the session
    public ActionResult DisplayData()
    {
        // Retrieve the data from the session
        string data = Session["UserData"] as string;

        // Pass the data to the view
        ViewBag.UserData = data;

        return View();
    }
}
```

3. Create Views:
   - ✓ Create the necessary views for the controller actions.
   - ✓ You'll need views for the "Index" action (to input data) and the "DisplayData" action (to display the data).

   - o Create a view named "Index.cshtml" for the "Index" action. This view will have a form to input data:

```
html
@model string

<h2>Enter Data</h2>
<form action="@Url.Action("StoreData")" method="post">
    <input type="text" name="data" />
    <input type="submit" value="Submit" />
</form>
```

   - o Create a view named "DisplayData.cshtml" for the "DisplayData" action. This view will display the data retrieved from the session:

```
html
@model string

<h2>Display Data</h2>
<p>Data from the session: @ViewBag.UserData</p>
```

4. Enable Session State:

- ✓ In your application's `web.config` file, make sure that session state is enabled.
- ✓ By default, ASP.NET MVC enables session state. Ensure that your `web.config` includes the following section within the `<system.web>` element:

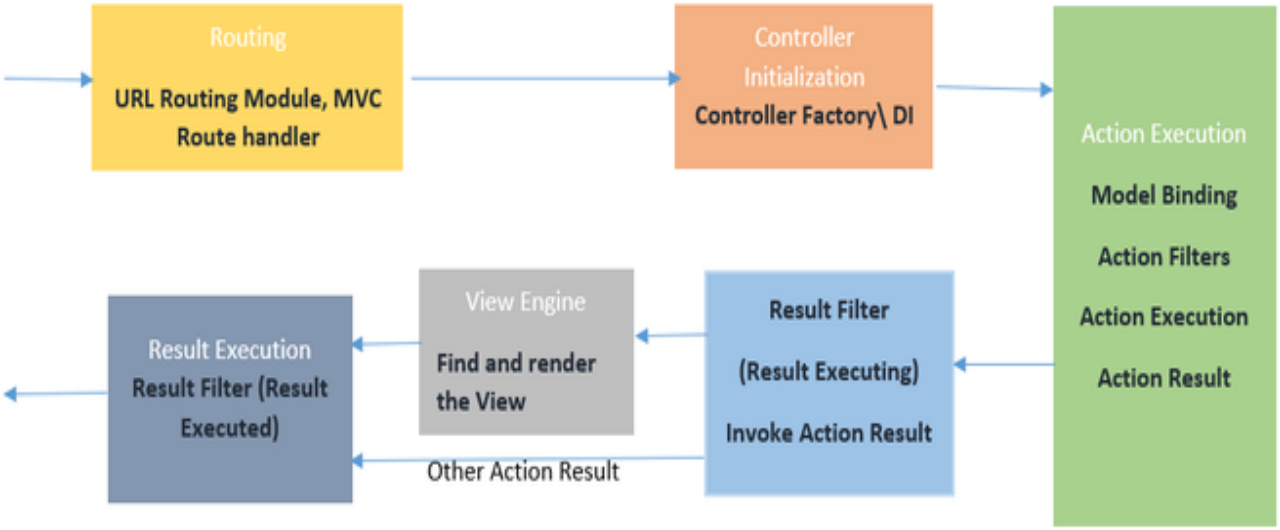
xml
<sessionState mode="InProc" timeout="20" />

- This configuration specifies that session data should be stored "InProc" (in-memory) and that the session timeout is set to 20 minutes.

5. Run the Application:
   - ✓ Build and run your ASP.NET MVC application.
   - ✓ Navigate to the "Index" action, enter some data, submit the form, and then navigate to the "DisplayData" action.
   - ✓ You should see the data you entered displayed on the "DisplayData" view.

   - This example demonstrates how to store data in a session and retrieve it in another view within an ASP.NET MVC application. Remember that session data is temporary and will be available as long as the user's session is active, based on the session timeout configured in the `web.config`.

- **Explain MVC application life cycle.**



- The life cycle of an ASP.NET MVC application refers to the sequence of events and stages that occur from the moment a user makes an HTTP request to your application until the response is sent back to the user.

- Understanding the MVC application life cycle is essential for developers to comprehend how requests are processed and how they can control and customize various aspects of the application's behavior.

- Here's an overview of the typical life cycle stages of an ASP.NET MVC application:

1. Request Routing:
   - ✓ The process begins when a user makes an HTTP request to a specific URL.
   - ✓ The ASP.NET MVC framework uses routing rules defined in the RouteConfig to determine which controller and action method should handle the request.

2. Controller Selection:
   - ✓ Based on the route configuration, the framework identifies the appropriate controller.
   - ✓ The controller class is instantiated.

3. Action Method Execution:
   - ✓ The framework identifies the specific action method to be executed within the selected controller.
   - ✓ Model binding occurs, where incoming data from the request (e.g., form data, query strings) is mapped to action method parameters.
   - ✓ The action method is executed, which typically involves processing the request, interacting with models or services, and preparing data for the view.

4. Result Execution:
   - ✓ The action method returns an instance of an `ActionResult` (e.g., `ViewResult`, `JsonResult`, `RedirectResult`, etc.).

- ✓ The framework executes the `ExecuteResult` method on the selected `ActionResult` to generate the response content.

5. View Rendering (for `ViewResult`):
   - ✓ If the action method returns a `ViewResult`, the framework identifies the associated view (based on conventions or explicit specification).
   - ✓ The view is rendered, and the model data is passed to it.
   - ✓ The view engine (e.g., Razor) processes the view and generates HTML markup.

6. Response Sending:
   - ✓ The generated response (HTML, JSON, etc.) is sent back to the user's browser.
   - ✓ The HTTP status code, headers, and content are assembled and sent as an HTTP response.

7. Application Events and Filters:
   - ✓ Throughout the life cycle, various events and filters (e.g., action filters, authorization filters) can be executed before, during, and after the processing of the request.
   - ✓ These allow developers to add custom logic, such as authentication, logging, or exception handling.

8. Exception Handling:
   - ✓ If an unhandled exception occurs during any stage of the request processing, the framework can execute global error handling mechanisms, like custom error pages or custom exception filters.

9. Response Finalization:
   - ✓ After the response is sent, final cleanup and disposal of resources may occur.
   - ✓ Any objects created during the request can be disposed of.

10. Logging and Monitoring:
    - ✓ Developers can incorporate logging and monitoring to track application behavior and performance.

11. Session Management and State Handling:

- ✓ If session state is being used, the framework manages session data throughout the request life cycle.
- ✓ Session data is typically loaded at the beginning and saved at the end of each request.

12. Caching:
    - ✓ Developers can implement caching mechanisms to improve application performance by storing and serving cached responses for frequently requested content.

13. Application Termination:
    - ✓ When the application is shut down or recycled (e.g., during application pool recycling in IIS), any cleanup tasks are performed.

- Understanding the ASP.NET MVC application life cycle helps developers control the flow of requests, apply custom logic, and optimize the performance and behavior of their web applications.

- It also facilitates the use of various MVC features like routing, filters, and model binding effectively.

• List out different return types of a controller action method

- In ASP.NET MVC, controller action methods can have various return types, depending on the desired behavior and the data to be returned to the client.
- Here are some common return types for controller action methods:

1. ViewResult:
    - ✓ Represents an HTML page or view to be rendered in the client's browser.
    - ✓ Used to return views that display data to the user.
    - ✓ Example: `return View("Index");`

2. PartialViewResult:

- ✓ Similar to ViewResult but returns a partial view, which can be used to render a portion of a page.
- ✓ Example: `return PartialView("_PartialView");`

3. RedirectToActionResult:
   - ✓ Redirects the client's browser to a different action method within the same or a different controller.
   - ✓ Example: `return RedirectToAction("ActionName", "ControllerName");`

4. RedirectToRouteResult:
   - ✓ Redirects the client's browser to a different route defined in the routing configuration.
   - ✓ Example: `return RedirectToRoute(new { controller = "Home", action = "Index" });`

5. JsonResult:
   - ✓ Returns JSON data to the client.
   - ✓ Example: `return Json(data);`

6. ContentResult:
   - ✓ Returns plain text or custom content to the client.
   - ✓ Example: `return Content("Hello, World!");`

7. FileResult:
   - ✓ Used to return files to the client for download.
   - ✓ Example: `return File("path-to-file.pdf", "application/pdf", "DownloadedFile.pdf");`

8. HttpNotFoundResult:
   - ✓ Returns a 404 Not Found HTTP status code.
   - ✓ Example: `return HttpNotFound();`

9. HttpStatusCodeResult:
   - ✓ Returns a custom HTTP status code and an optional status description.

- ✓ Example: `return new HttpStatusCodeResult(403, "Access Denied");`

10. HttpStatusCodeResult:
    - ✓ Returns an HTTP status code indicating success (e.g., 200 OK).
    - ✓ Example: `return new HttpStatusCodeResult(200);`

11. EmptyResult:
    - ✓ Represents no result to be returned.
    - ✓ Example: `return new EmptyResult();`

12. ViewResultBase:
    - ✓ An abstract base class for ViewResult and PartialViewResult.

- These are some of the commonly used return types for controller action methods in ASP.NET MVC.
- The choice of return type depends on the specific requirements of your application and the kind of response you want to send to the client.

• What are filters in MVC?

- In ASP.NET MVC (Model-View-Controller), filters are a mechanism for injecting code into the request processing pipeline to perform tasks such as authentication, authorization, logging, caching, exception handling, and more.

- Filters allow you to add cross-cutting concerns to your application without cluttering your controller actions with repetitive code.

- Filters are applied globally, at the controller level, or at the action level, depending on your requirements.

- There are several types of filters in ASP.NET MVC:

1. Authorization Filters:
    - ✓ Used to control access to controller actions or entire controllers.
    - ✓ Examples include `AuthorizeAttribute` for checking user authentication and `AllowAnonymousAttribute` to allow anonymous access.

2. Action Filters:
    - ✓ Execute code before and after the execution of an action method.
    - ✓ Examples include `ActionFilterAttribute`, `OnActionExecuting`, and `OnActionExecuted`.

3. Result Filters:
    - ✓ Execute code before and after the execution of the action result.
    - ✓ Examples include `ResultFilterAttribute`, `OnResultExecuting`, and `OnResultExecuted`.

4. Exception Filters:
    - ✓ Handle exceptions thrown during the execution of an action.
    - ✓ Examples include `HandleErrorAttribute`, `OnException`.

5. Resource Filters:
    - ✓ Execute code during the beginning and end of request processing, independent of the action method execution.
    - ✓ Examples include `IResourceFilter`, `IAsyncResourceFilter`.

6. Custom Filters:
    - ✓ You can create your own custom filters by implementing specific interfaces like `IActionFilter`, `IResultFilter`, `IExceptionFilter`, or `IResourceFilter`.

    - o To use filters in ASP.NET MVC, you can decorate your controllers or actions with the appropriate filter attributes, or you can register filters globally in the `GlobalFilters` class in the `FilterConfig.cs` file.

Here's an example of using an authorization filter to restrict access to a controller action:

csharp
```
[Authorize] // AuthorizeAttribute is an example of an authorization filter
```

```
public ActionResult SecureAction()
{
   // Code for the secure action
}
```

- In this example, the `[Authorize]` attribute is an authorization filter that ensures that only authenticated users can access the `SecureAction` method.

- Filters provide a powerful way to add reusable and centralized behavior to your MVC application while maintaining separation of concerns between different aspects of your application logic.

- **What are HTML helpers in MVC?**

  - In ASP.NET MVC (Model-View-Controller), HTML helpers are utility methods provided by the framework to assist in generating HTML markup within views.

  - These helpers simplify the process of rendering HTML elements and can help maintain consistency and correctness in the generated HTML.

  - HTML helpers are typically used within Razor views to create HTML elements that correspond to model properties or other data.

  - Here are some common examples of HTML helpers in ASP.NET MVC:

1. TextBoxFor and TextAreaFor:
   - ✓ `Html.TextBoxFor` and `Html.TextAreaFor` helpers are used to generate `<input>` and `<textarea>` elements, respectively, for editing model properties.
   - ✓ Example:

```
csharp
@Html.TextBoxFor(model => model.FirstName)
@Html.TextAreaFor(model => model.Description)
```

2. DropDownListFor:

- ✓ `Html.DropDownListFor` helper is used to generate a `<select>` element for selecting a value from a list.
- ✓ Example:

csharp
@Html.DropDownListFor(model => model.CategoryId, SelectListItems)

3. CheckBoxFor:
   - ✓ `Html.CheckBoxFor` helper is used to generate a `<input type="checkbox">` element for Boolean model properties.
   - ✓ Example:

csharp
@Html.CheckBoxFor(model => model.IsSubscribed)

4. RadioButtonFor:
   - ✓ `Html.RadioButtonFor` helper is used to generate `<input type="radio">` elements for selecting a single option from a group.
   - ✓ Example:

csharp
@Html.RadioButtonFor(model => model.Gender, "Male") Male
@Html.RadioButtonFor(model => model.Gender, "Female") Female

5. ActionLink and RouteLink:
   - ✓ `Html.ActionLink` and `Html.RouteLink` helpers are used to generate hyperlinks to controller actions or custom routes.
   - ✓ Example:

csharp
@Html.ActionLink("Home", "Index", "Home")
@Html.RouteLink("Custom Route", "CustomRouteName")

6. DisplayFor and DisplayTextFor:
   - ✓ `Html.DisplayFor` and `Html.DisplayTextFor` helpers are used to render non-editable values from model properties.
   - ✓ Example:

csharp
@Html.DisplayFor(model => model.LastName)
@Html.DisplayTextFor(model => model.Age)

7. ValidationMessageFor:

- ✓ `Html.ValidationMessageFor` helper is used to display validation error messages associated with model properties.
- ✓ Example:

csharp
@Html.TextBoxFor(model => model.Email)
@Html.ValidationMessageFor(model => model.Email)

8. Partial:
   - ✓ `Html.Partial` helper is used to render partial views within other views, allowing you to reuse UI components.
   - ✓ Example:

csharp
@Html.Partial("_PartialViewName")

These are just a few examples of HTML helpers in ASP.NET MVC. HTML helpers simplify the process of generating HTML markup while allowing you to bind the markup to model properties and maintain consistency throughout your views.

- **Differences between Razor and ASPX View Engine in MVC?**

  - Razor and ASPX are two different view engines in ASP.NET MVC, and they have several differences in terms of syntax, performance, and ease of use.

  - Here are some key differences between Razor and ASPX view engines:

1. Syntax:
   - ✓ Razor:
     - Razor uses a concise and more readable syntax that resembles HTML with C# code interspersed using `@` symbols.
     - It's known for its clean and intuitive syntax.
   - ✓ ASPX:
     - ASPX views use a more verbose syntax with a mix of HTML and inline server-side code blocks enclosed in `<% %>` tags.

2. Readability:

- ✓ Razor:
  - Razor is often considered more readable and developer-friendly due to its simple and clean syntax.
  - It's easier to understand and maintain.
- ✓ ASPX:
  - ASPX views can become cluttered with a lot of server-side code, making them less readable, especially for complex views.

3. IntelliSense Support:
   - ✓ Razor:
     - Razor benefits from excellent IntelliSense support in modern code editors, making it easier to write and debug views.
   - ✓ ASPX:
     - ASPX views may not have as robust IntelliSense support as Razor.

4. Performance:
   - ✓ Razor:
     - Razor is generally considered more efficient in terms of performance compared to ASPX views.
     - Razor views are compiled into efficient code, leading to faster execution.
   - ✓ ASPX:
     - ASPX views may have a slightly higher overhead because they rely on the Web Forms view engine, which was designed for Web Forms applications.

5. Layouts:
   - ✓ Razor:
     - Razor uses `_Layout.cshtml` as the default layout file, making it easier to define and manage layout templates.
   - ✓ ASPX:
     - ASPX views can use master pages for layout, which have a different syntax and structure.

6. Extension:
    - ✓ Razor:
        - Razor views have the `.cshtml` extension for C# views and `.vbhtml` for VB.NET views.
    - ✓ ASPX:
        - ASPX views have the `.aspx` extension.

7. Model Binding:
    - ✓ Razor:
        - Razor has cleaner syntax for model binding and accessing model properties.
    - ✓ ASPX:
        - ASPX views use a different syntax for model binding, which can be less intuitive.

8. View Bag/View Data:
    - ✓ Razor:
        - Razor provides `@ViewBag` or `@ViewData` for passing data to views.
    - ✓ ASPX:
        - ASPX views use `<%= ViewData["Key"] %>` syntax for similar purposes.

9. Whitespace Control:
    - ✓ Razor:
        - Razor offers better control over whitespace rendering, allowing you to suppress whitespace when needed.
    - ✓ ASPX:
        - ASPX views may have less control over whitespace rendering.

- Overall, Razor is the preferred view engine for ASP.NET MVC due to its cleaner syntax, better performance, and improved developer experience.

- It has become the standard choice for creating views in modern MVC applications.

- However, existing ASPX views from older projects can still be used in newer MVC applications if necessary.

- **Pass one values from one view to another view using query string.**

    - To pass a value from one view to another view using a query string in ASP.NET MVC, you can use the `Url.Action` or `Url.RouteUrl` helper methods to generate URLs with query string parameters.

    - Here's a step-by-step guide:

1. In your controller action where you want to generate the link with the query string parameter, create the link using `Url.Action` or `Url.RouteUrl`.
    - ✓ For example:

```
csharp
public ActionResult FirstView()
{
    int valueToPass = 42; // The value you want to pass

    // Generate the URL with the query string parameter
    string url = Url.Action("SecondView", "Home", new { value = valueToPass });

    // Redirect to the URL
    return Redirect(url);
}
```

- ✓ In this example, `valueToPass` is the value you want to pass, and `Url.Action` generates a URL that includes this value as a query string parameter named "value."

2. In your target controller action (in this case, "SecondView" in the "Home" controller), you can retrieve the value from the query string using the `Request.QueryString` collection or by defining a parameter in the action method itself.

- ✓ Here's an example using a parameter:

```
csharp
public ActionResult SecondView(int value)
{
    // Use the 'value' parameter in your action method
    ViewBag.ValueFromQueryString = value;

    return View();
}
```

- ✓ In this example, the "value" parameter in the "SecondView" action method will automatically bind to the value passed in the query string.

3. In your "SecondView" Razor view, you can display or use the value as needed:

```
html
<p>Value from query string: @ViewBag.ValueFromQueryString</p>
```

- ✓ Now, when you navigate from the "FirstView" action to the "SecondView" action, the value will be passed in the query string, and you can access it in the "SecondView" action and display it in the view.