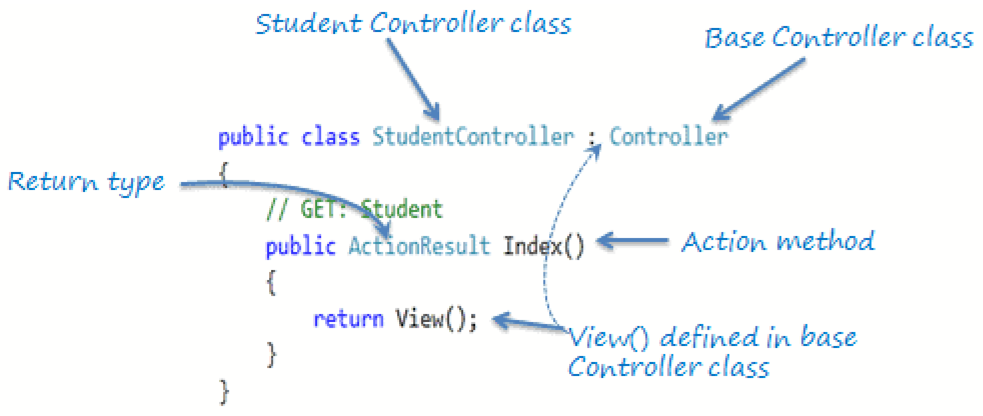
Chapter 4 sample question and answer

1. What is Action method in asp.net core? Types of action method available in asp.net core.

All the public methods of the `Controller` class are called `Action` methods. They are like any other normal methods with the following restrictions:

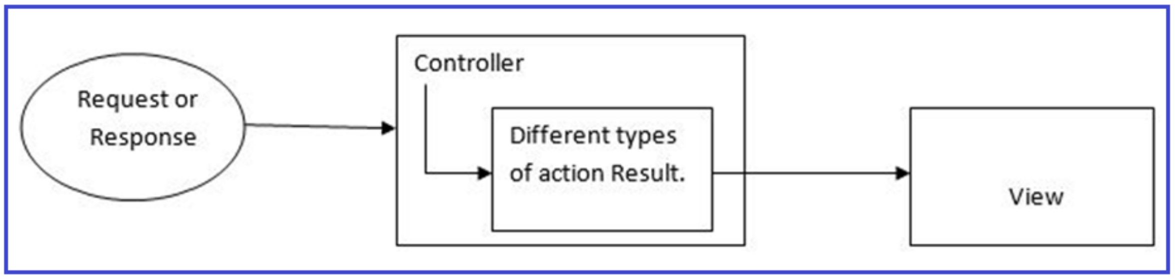
- Action method must be public. It cannot be private or protected
- Action method cannot be overloaded
- Action method cannot be a static method.

The following illustrates the `Index()` action method in the `StudentController` class.



```
public class StudentController : Controller
{
    // GET: Student
    public ActionResult Index()
    {
        return View();
    }
}
```

As you can see in the above figure, the **Index()** method is public, and it returns the **ActionResult** using the View() method. The **View()** method is defined in the Controller base class, which returns the appropriate **ActionResult**.



Action Method	Description
IActionResult	Defines a contract that represents the result of an action method.
ActionResult	A default implementation of IActionResult.
ContentResult	Represents a text result.
EmptyResult	Represents an ActionResult that when executed will do nothing.
JsonResult	An action result which formats the given object as JSON.
PartialViewResult	Represents an ActionResult that renders a partial view to the response.
ViewResult	Represents an ActionResult that renders a view to the response.

```
public IActionResult Index()
    {
        return View();
    }
    public ActionResult About()
    {
        return View();
    }

     public ContentResult ContentResult()
    {
        return Content("I am ContentResult");
    }

    public JsonResult GetValue()
    {
        var name = "Sunil Chaudhary";
        return Json(new { data = name });
    }
    public PartialViewResult PartialViewResult()
    {
        return PartialView("_PartialView");
    }

    public ViewResult ViewResult()
     {
         return View("About", "Employee");
     }
```

2.Write steps to render HTML with views

Step 1: The Home controller is represented by a Home folder inside the Views folder. The Home folder contains the views for the About, Contact, and Index (homepage) webpages. When a user requests one of these three webpages, controller actions in the Home controller determine which of the three views is used to build and return a webpage to the user.

Step:2 To create a view, add a new file and give it the same name as its associated controller action with the .cshtml file extension.

Step:3 Write razor code with following convention

- Razor code blocks are enclosed in @{ ... }

- Inline expressions (variables and functions) start with @
- Code statements end with semicolon
- Variables are declared with the var keyword
- Strings are enclosed with quotation marks
- C# code is case sensitive
- C# files have the extension .cshtml

3.What is Tab Helpers? Explain types of Tag Helpers present in asp.net core?

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files

Advantage of Tag Helper:

- Tag Helpers use server-side binding without any server-side code.
- This helper object is very much use full when HTML developers do the UI designing who does not have any idea or concept about Razor syntax.
- It provides us an experience of working on basic HTML environments.
- It Supports rich IntelliSense environment support to create a markup between HTML and Razor
- To enable tag helper import `@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers`

In header section of .cshtml page

Types of Tag Helpers in ASP.NET Core:

There are two types of Tag helpers in ASP.NET Core. They are as follows:

- **Built-In Tag Helpers**: They come in-built in the ASP.NET Core Framework and can perform common tasks like generating links, creating forms, loading assets, showing validation messages, etc.
- **Custom Tag Helper**: That can be created by us to perform our desired transformation on an HTML element.

Inbuilt Tag Helper

Form Tag Helper

- Form
- FormAction

- Input
- Label
- Option
- Select
- TextArea
- ValidationMessage
- ValidationSummary

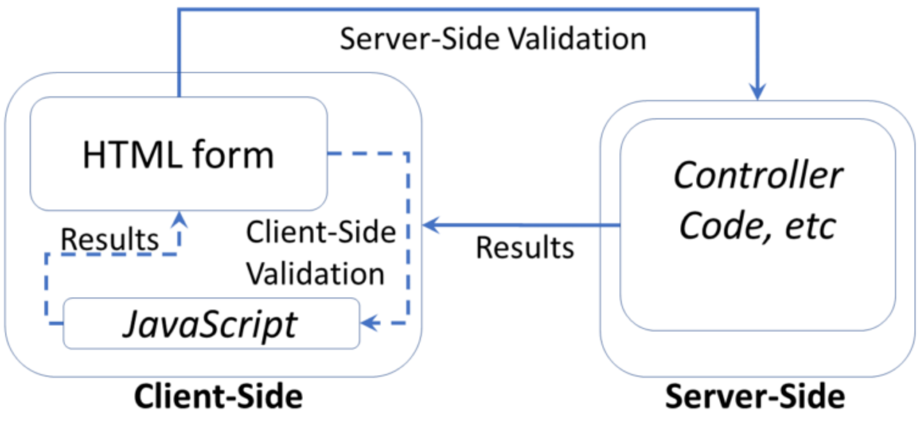
```
<form asp-controller="Home" asp-action="Index" method="post">
    @* Validation Summary *@
    <div asp-validation-summary="All"></div>

    @* Label, Input, Validation(span) *@
    <input asp-for="Id" />

    <label asp-for="Firstname"></label>
    <input asp-for="Firstname" />
    <span asp-validation-for="Firstname"></span> <br />

    <label asp-for="BirthDate"></label>
    <input asp-for="BirthDate" asp-format="{0:dd/MM/yyyy}" />
    <span asp-validation-for="BirthDate"></span> <br />

    @* Textarea *@
    <label asp-for="Notes"></label>
    <textarea asp-for="Notes"></textarea>
    <span asp-validation-for="Notes"></span> <br />

    @* Select (single) *@
    <label asp-for="Title"></label>
    <select asp-for="Title" asp-items="Model.GetTitles()">
        <option value="">--select--</option>
    </select><br />

    @* Select (multiple) *@
    <label asp-for="Interests"></label>
    <select asp-for="Interests" asp-items="Model.GetInterests()"></select><br />

    @* Select (enum) *@
    <label asp-for="Gender"></label>
    <select asp-for="Gender" asp-items="Html.GetEnumSelectList<Gender>()">
        <option value="" selected>--select--</option>
    </select><br />

    <br />
    <button type="submit">Save</button><br />
</form>
```

Anchor Tag

```
<a asp-controller="Home" asp-action="Index" asp-route-id=@Model.Id>Index</a>
```

4. What is Dataannotation in asp.net core? How client side and server side validation done in asp.net core?

Data Annotations are nothing but certain **validations that we put in our models to validate the input from the user**.
Two Types of validation:



1. **Client-Side Validation**

All the client-side validation in the world won't prevent a malicious user from sending a GET/POST request to your form's endpoint.

client-side validation helps to catch the problem before your server receives the request, while providing a better user experience.

```
<span asp-validation-for="Fullname" class="text-danger"></span>
```

2. **Server-Side Validation**

Validation occurs before an MVC controller action (or equivalent handler method for Razor Pages) takes over. As a result, you should check to see if the validation has passed before continuing next steps.

```
[HttpPost]
public IActionResult Create(StudentViewModel stu)
```

```
  {
     if (ModelState.IsValid)
       {
  ViewBag.Message = stu.Fullname + " : Record Submited Successfully";
            }
            return View();

         }
```

5. Types of Data Annotation in asp.net core?

- **Required**

This attribute specifies that the value is mandatory and cannot be skipped.

Syntax

[Required(ErrorMessage="Please enter name"),MaxLength(30)]

- **Range**

Using this attribute we can set a range between two numbers.

Syntax

*[Range(100,500,ErrorMessage="Please enter correct valu*e")]

- **StringLength**

Using this attribute we can specify maximum and minimum length of the property.

Syntax

[StringLength(30,ErrorMessage="Do not enter more than 30 characters")]

3. **DisplayName**

Using this attribute we can specify property name to be displayed on view.

Syntax

[Display(Name="Student Name")]

- **MaxLength**

Using this attribute we can specify maximum length of property.

Syntax

[MaxLength(3)]

- **DisplayFormat**

This attribute allows us to set date in the format specified as per the attribute.

Syntax

[DisplayFormat(DataFormatString = "{0:dd.MM.yyyy}")]

- **RegularExpression**

We can set a regex pattern for the property. For ex: Email ID.

Example:

```
public class Student

    public int StudentID { get; set; }
    [DataType(DataType.Text)]
    [Required(ErrorMessage = "Please enter name"), MaxLength(30)]
    [Display(Name = "Student Name")]

    public string Name { get; set; }
    [MaxLength(3),MinLength(1)]
    [Required(ErrorMessage = "Please enter marks")]

    publicint Marks { get; set; }
    [DataType(DataType.EmailAddress)]
    [Required(ErrorMessage = "Please enter Email ID")]
    [RegularExpression(@"^\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-
.]\w+)*$", ErrorMessage = "Email is not valid.")]

    public string Email { get; set; }
    [Required(ErrorMessage = "Please enter department")]

    public string Department { get; set; }
    [Required(ErrorMessage = "Please enter Mobile No")]
    [Display(Name = "Contact Number")]
    [DataType(DataType.PhoneNumber)]
    public int Mobile { get; set; }

}
```

6. What is URL Routing? How does it works

URL Routing

ASP.NET Core controllers use the Routing middleware to match the URLs of incoming requests and map them to actions. Route templates:

- Are defined in startup code or attributes.
- Describe how URL paths are matched to **actions**.
- Are used to generate URLs for links. The generated links are typically returned in responses.

Actions are either **conventionally-routed** or **attribute-routed**. Placing a route on the controller or **action** makes it attribute-routed.

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

7. What is ViewModel? Why it is important in asp.net core?

There are, however, a lot of situations where you may want to create a specific ViewModel for a specific View. This can be to extend or simplify an existing Model, or because you want to represent something in a View that's not already covered by one of your models.

 ViewModels are often placed in their own directory in your project, called "ViewModels".

 Some people also prefer to postfix the name of the class with the word ViewModel, e.g. "AddressViewModel" or "StudentViewModel".

The **purpose** of a **ViewModel** is for the view to have a single object to render, alleviating the need for UI logic code in the view that would otherwise be necessary. This means the only responsibility, or concern, of the view is to render that single **ViewModel** object, aiding in a cleaner separation of concerns (SoC)

```
public class UserLoginViewModel

{

[Required(ErrorMessage = "Please enter your username")]

[Display(Name = "User Name")]

[MaxLength(50)]
public string UserName { get; set; }

 [Required(ErrorMessage = "Please enter your password")]

 [Display(Name = "Password")]

 [MaxLength(50)]

 public string Password { get; set; }

}
```

8. what is Dependency Injection (DI) and IoC.

IoC is a design principle which recommends the inversion of different kinds of controls in object-oriented design to achieve loose coupling between application classes. In this case, control refers to any additional responsibilities a class has, other than its main responsibility, such as control over the flow of an application, or control over the dependent object creation and binding (Remember SRP - Single Responsibility Principle). If you want to do TDD (Test Driven Development), then you must use the IoC principle, without which TDD is not possible

Dependency Injection (DI) is a design pattern which implements the IoC principle to invert the creation of dependent objects. Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways.

The Dependency Injection pattern involves 3 types of classes.

- **Client Class:** The client class (dependent class) is a class which depends on the service class
- **Service Class:** The service class (dependency) is a class that provides service to the client class.
- **Injector Class:** The injector class injects the service class object into the client class.

The IoC container is a framework used to manage automatic dependency injection throughout the application, so that we as programmers do not need to put more time and effort into it.