Write a Asp.net core application for demonstrating the usage of models, binding, and validations in ASP.NET.

Explain various types of action results available in ASP.NET Core, ranging from basic text and JSON responses to more advanced scenarios like rendering views and serving files.

Describe routing used in asp.net core MVC.

**Routing in ASP.NET Core MVC**

Routing is the process through which the application matches an **incoming URL path** and executes the corresponding **action methods**. ASP.NET Core MVC uses a **routing** middleware to match the **URLs of incoming requests** and map them to specific action methods.

There are two types of routing for action methods:

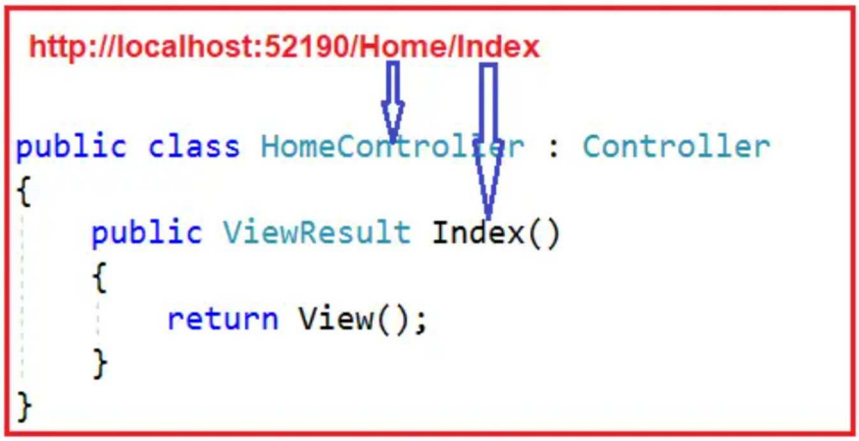
- Conventional Routing
- Attribute Routing
- Area Routing

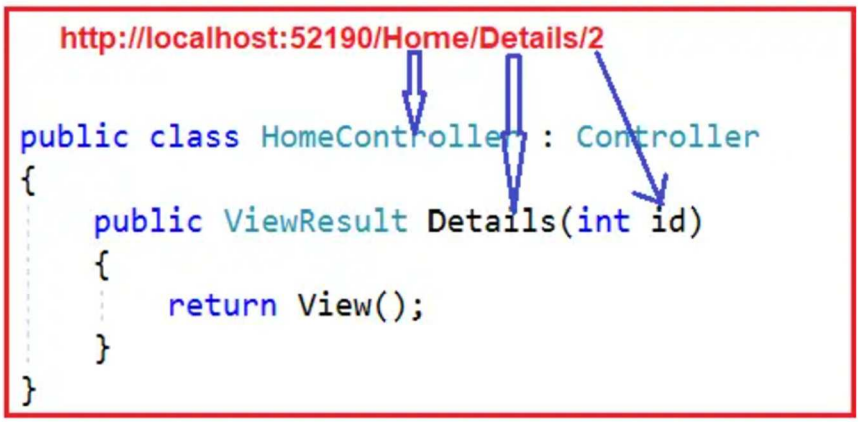
**Conventional Routing**

When we create a new ASP.NET Core MVC application using the **default** template, the application configures a default routing.

After creating a new project with the default ASP.NET Core MVC template, the **program.cs** class. We can see that the application has configured a default routing using

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



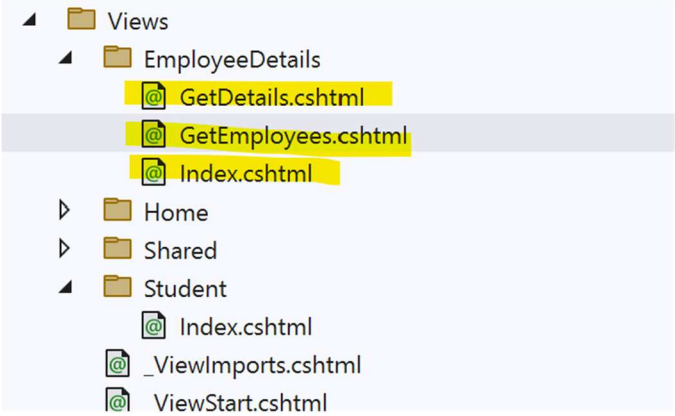



- **{controller}**: Represents the name of the controller class.
- **{action}**: Represents the action method's name within the controller.
- **{id?}**: Represents an optional route parameter called "id".

**Attribute routing**: The route is determined based on attributes that you set on your controllers and methods. These will define the mapping to the controller's actions.

In attribute-based routing, we make use of **[Route]** attributes at the controller and/or action level to specify route details for your application.

```
[Route("Employee")]
 public class EmployeeDetailsController : Controller
 {
     [HttpGet("Index")]
     public IActionResult Index()
     {
         return View();
     }
     [Route("Details/{id:int?}")]
     public IActionResult GetDetails(int Id)
     {
         return View();
     }
     [Route("~/EmployeeDetails/All")]
     public IActionResult GetEmployees()
     {
         return View();
     }
 }
```




`Url:`

Index Action: https://localhost:7251/Employee/Index

GetDetails Action: https://localhost:7251/Employee/Details/1

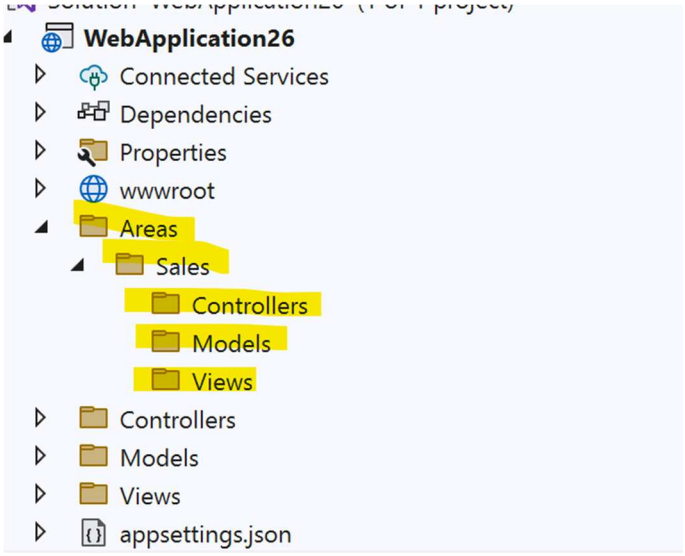
GetEmployees Action: https://localhost:7251/EmployeeDetails/All

**Areas Routing:**

In ASP.NET Core MVC, areas provide a way to organize controllers, views, and related files into separate functional sections within your application. Each area can have its own controllers, views, and other assets, allowing for better organization and separation of concerns. Routing within areas can be configured independently from the main application routes, providing additional flexibility. Here's how you can set up area routing in ASP.NET Core MVC:

**Create an Area**:
To create a new Area in an app, right click on the application name on the solution explorer and select Add ➤ New Folder and name this folder Areas. Create another folder inside this newly created Areas folder and name it as **Sales**. Now create folders for Controllers, Models & Views inside it. I have illustrated it on the below image:

**Add Controllers and Views**




**Configure Area Routing**:

Add a Route for the area in the Program.cs class of the app. I have shown this in the below code.

```
app.MapControllerRoute(
    name: "areas",
    pattern: "{area:exists}/{controller=Home}/{action=Index}");
```

To understand how the Sales Area will work, first create a new class called **Product.cs** inside the **Areas ➤Sales ➤Models** folder.

```
public class Product
{
    public string Name { get; set; }
    public int Quantity { get; set; }
    public int Price { get; set; }
}
```

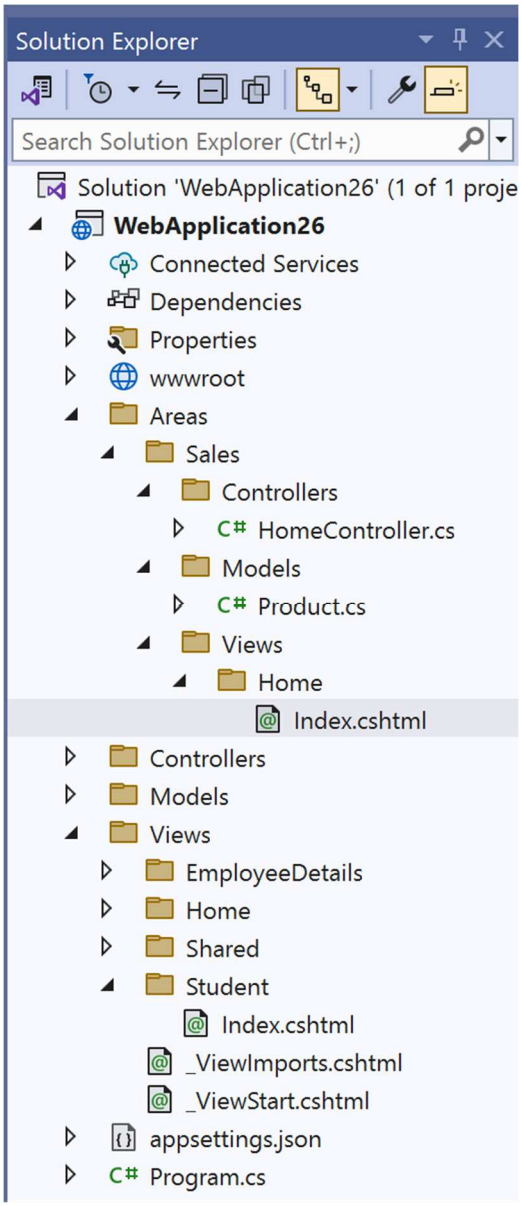
Next create a new controller inside Areas ➤ Sales ➤ Controllers folder, and name it HomeController.cs. Add the below code to it:

```
    [Area("Sales")]
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            List<Product> list = new List<Product>() {
                new Product { Name = "Pants", Quantity = 5, Price=100 },
                new Product { Name = "Shirts", Quantity = 10, Price=80 },
                new Product { Name = "Shoes", Quantity = 15, Price=50 }
            };
            return View(list);
        }
```

```
    }
```

Now Create Index page

```
@model IEnumerable<WebApplication26.Areas.Sales.Models.Product>

@{
    ViewData["Title"] = "Index";
}

<table>
    <tr>
        <th>Name</th>
        <th>Quantity</th>
        <th>Price</th>
    </tr>
    @foreach (var p in Model)
    {
        <tr>
            <td>@p.Name</td>
            <td>@p.Quantity</td>
            <td>@p.Price</td>
        </tr>
    }
</table>
```



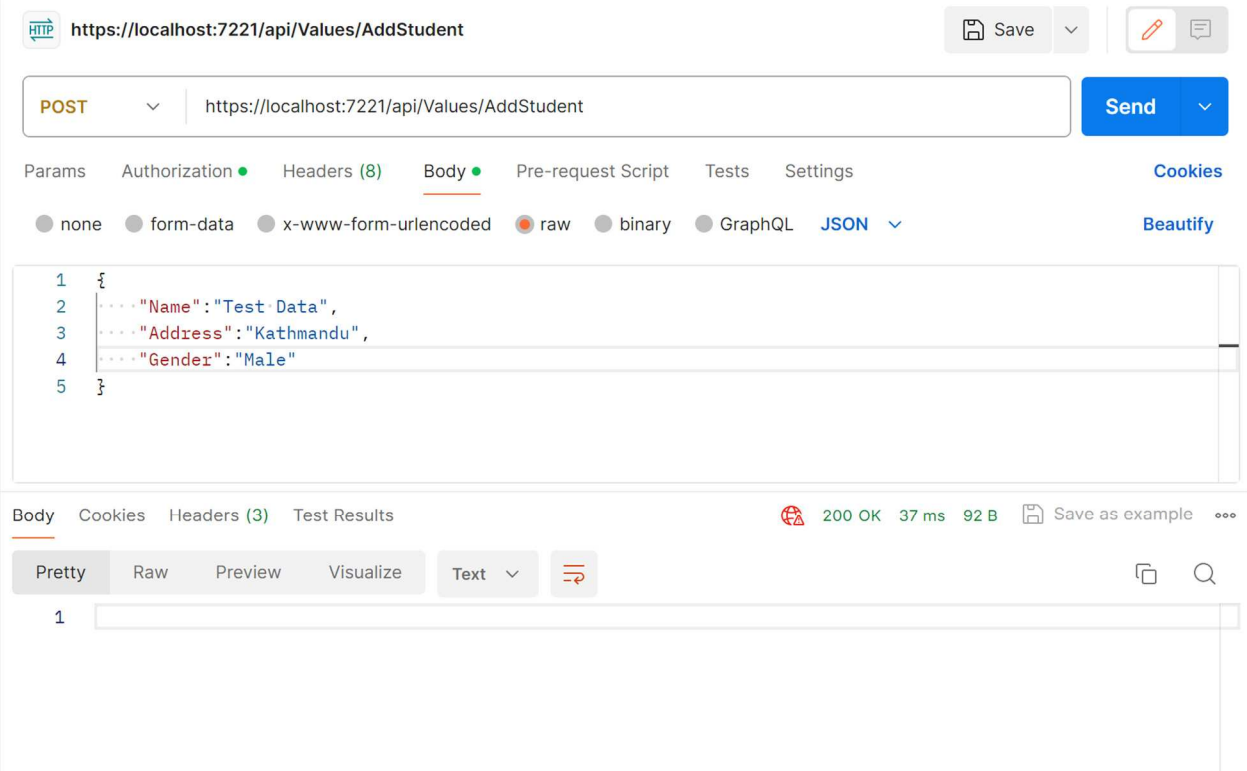

Accessing Area-Specific Routes

Use URL: https://localhost:7251/sales

**Web API Applications: API Controllers,**

Web API controller is a class which can be created under the Controllers folder or any other folder under your project's root folder. The name of a controller class must end with "**Controller**" and it must be derived from **System.Web.Http.ApiController** class. All the public methods of the controller are called action methods.

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    //https://localhost:7221/api/Values/GetStudents
    KathfordDbContext db =new KathfordDbContext();
    [HttpGet]
    [Route("GetStudents")]
    //https://localhost:7221/api/Values/GetStudents
    public List<TblStudent> Get()
    {
        return db.TblStudents.ToList();
    }

    //https://localhost:7221/api/Values/GetStudentByID/1
    [HttpGet()]
    [Route("GetStudentByID/{Id}")]

    public TblStudent Get(int id)
    {
        return db.TblStudents.Where(a=>a.Id==id).FirstOrDefault();
    }

    // POST api/<ValuesController>
    [HttpPost]
    [Route("AddStudent")]
    public void Post(TblStudent st)
    {
    }

    // PUT api/<ValuesController>/5
    [HttpPut()]
    [Route("UpdateStudent")]
    public void Put(TblStudent st)
    {
    }

    // DELETE api/<ValuesController>/5
    [HttpDelete]
    [Route("DeleteStudent")]
    public void Delete(int id)
    {
    }
}
```




# Dependency Injection and IOC containers

Dependency Injection (DI) is a **pattern** and IoC container is a **framework**.

Dependency Injection (often called just DI) is a software design pattern that helps us create **loosely coupled** applications. It is an **implementation of the Inversion of Control (IoC)** principle, and Dependency Inversion Principle (D in SOLID).

An IoC (Inversion of Control) container is a framework or a component responsible for managing the dependencies of various components within the application.

In ASP.NET Core MVC, the built-in IoC container is part of the framework and is quite powerful. You can register your services in the **Program.cs** class using methods like **AddTransient**, **AddScoped**, and **AddSingleton**. The container will then handle the injection of these services into your controllers, views, or other components as needed.

```
builder.Services.AddTransient<IMyServices, MyServices>
```

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.