ASP.NET Core MVC

XORIANT

TIME FOR CASE STUDY

User Story – Sprint 1

- As a **Customer** of Shopon, I should be allowed to view all products on my laptop, desktop or from mobile and from anywhere.

Configuring Middleware

Some of the middleware are

UseDeveloperExceptionPage()

UseStaticFiles()

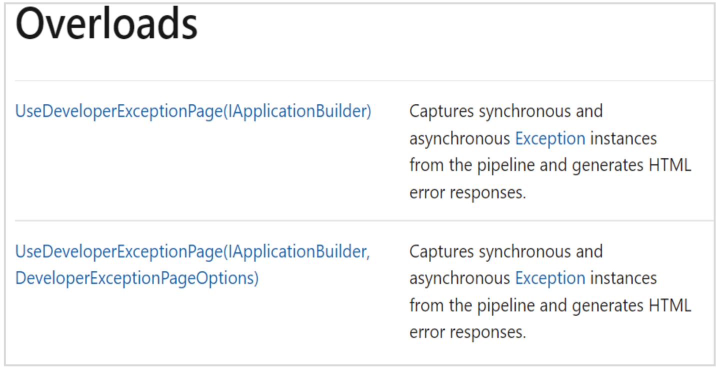
UseFileServer()

UseMvc()

UseEndpoints()

UseDeveloperExceptionPage()

- The major purpose of this method is to help the developers to inspect exception details during the development phase
- Purpose
  - To capture Synchronous and Asynchronous SystemException instance from the pipeline & generate HTML error response. It returns a reference to the application after the operation is completed
  - We use the **UseDeveloperException()** extension method to render the exception during the development mode
  - This method adds middleware into the request pipeline which displays developer-friendly exception detail page. This helps developers in tracing errors that occur during the development phase



Overloads

UseDeveloperExceptionPage(IApplicationBuilder)	Captures synchronous and asynchronous Exception instances from the pipeline and generates HTML error responses.
UseDeveloperExceptionPage(IApplicationBuilder, DeveloperExceptionPageOptions)	Captures synchronous and asynchronous Exception instances from the pipeline and generates HTML error responses.

Example

UseDeveloperExceptionPage - Example

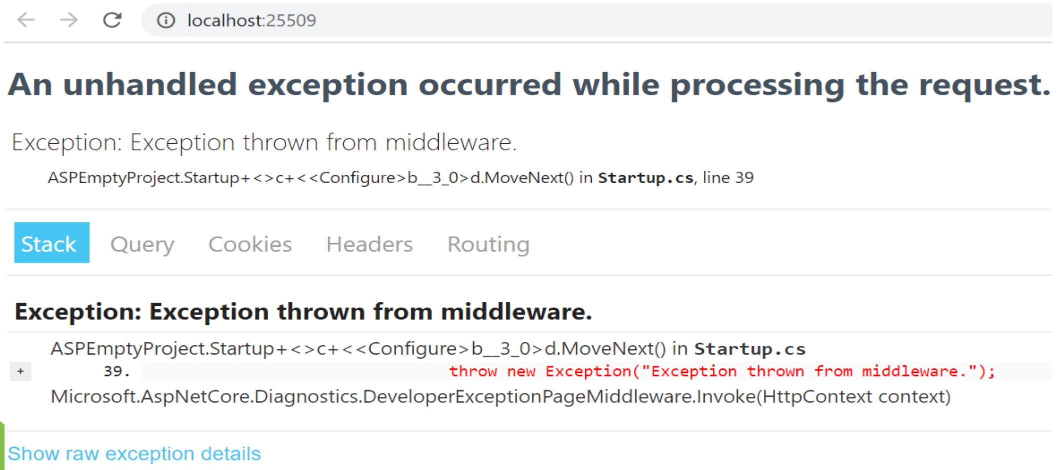
Startup.cs

```
public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.Run(async (context) =>
    {
        throw new Exception("Exception thrown from middleware.");
        await context.Response.WriteAsync(
            "Welcome to ASP.NET Core");
    });
}
```

NOTE: The number of lines displayed with exception line can be customized by **SourceCodeLineCount** property.

```
if (env.IsDevelopment())
{
    DeveloperExceptionPageOptions developerExceptionPage =
        new DeveloperExceptionPageOptions()
        {
            SourceCodeLineCount = 10
        };
    app.UseDeveloperExceptionPage(developerExceptionPage);
}
```

Browser Output



localhost:25509

An unhandled exception occurred while processing the request.

Exception: Exception thrown from middleware.

ASPEmptyProject.Startup+<>c+<<Configure>b__3_0>d.MoveNext() in **Startup.cs**, line 39

Stack Query Cookies Headers Routing

Exception: Exception thrown from middleware.

```
ASPEmptyProject.Startup+<>c+<<Configure>b__3_0>d.MoveNext() in Startup.cs
+    39.                    throw new Exception("Exception thrown from middleware.");
Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware.Invoke(HttpContext context)
```

Show raw exception details

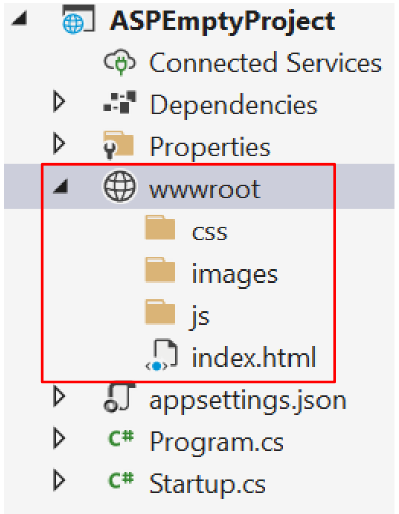
To get environment name

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync(env.EnvironmentName);
});
```

UseStaticFiles()

- ASP.NET Core by default will not support static file. To server it we must
  - Store all files in **wwwroot** folder(content root folder)
  - Register **UseStaticFiles** middleware
- All static files like .htm, html, .css, .js should be part of **wwwroot** folder

- Content folder should be directly placed in the root folder of the project



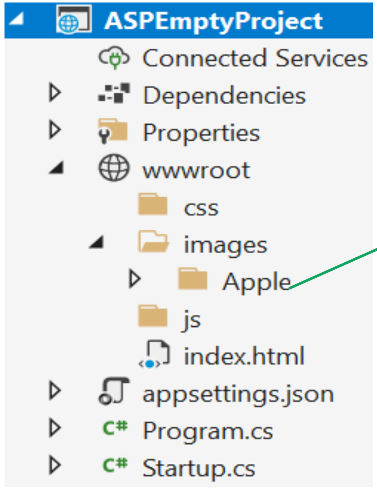
Example

Create Default Page

UseStaticFiles - Example

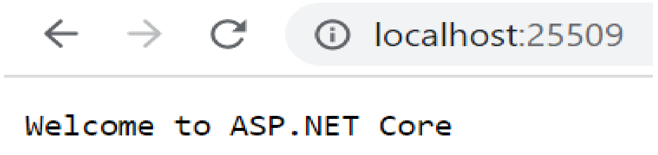
Startup.cs

```
public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync(
            "Welcome to ASP.NET Core");
    });
}
```



```
ASPEmptyProject
    Connected Services
    Dependencies
    Properties
    wwwroot
        css
        images
            Apple
        js
        index.html
    appsettings.json
    Program.cs
    Startup.cs
```

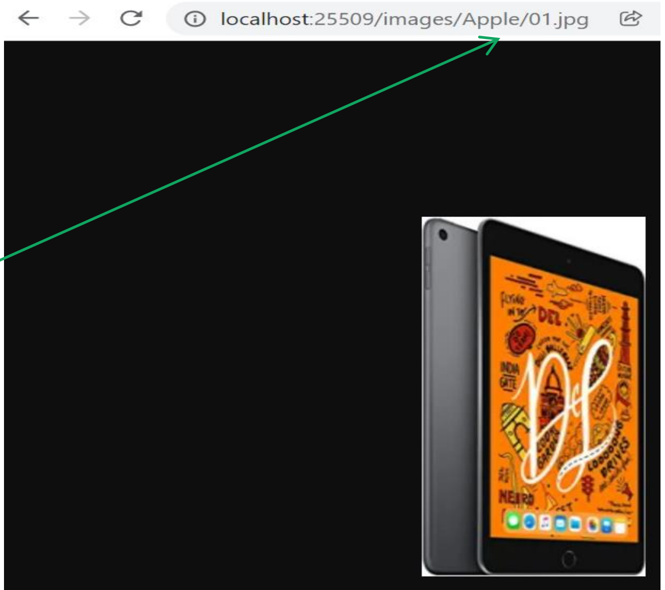
Browser Output



localhost:25509

```
Welcome to ASP.NET Core
```

Browser Output with static file request



localhost:25509/images/Apple/01.jpg

Creating Default Page

- ASP.NET Core supports creating static page as default page. To achieve this, the file name should be one of the following:
  - default.htm - index.htm
  - default.html - index.html

- **UseStaticFiles()** middleware cannot server default static page. We must chain **UseDefaultFiles()** middleware to serve the request

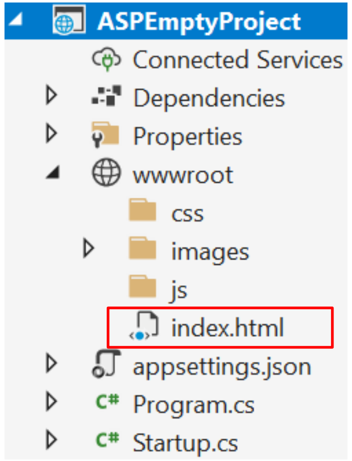
Example

Creating non default page as default

Creating Default Page - Example

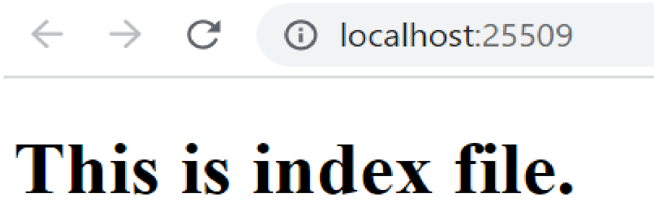
Startup.cs

```
public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseDefaultFiles();
    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync(
            "Welcome to ASP.NET Core");
    });
}
```



```
ASPEmptyProject
    Connected Services
    Dependencies
    Properties
    wwwroot
        css
        images
        js
        index.html
    appsettings.json
    Program.cs
    Startup.cs
```

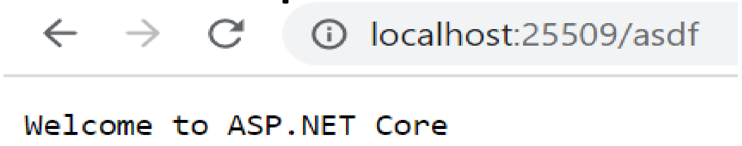
Browser Output



localhost:25509

This is index file.

Browser Output with different url



localhost:25509/asdf

```
Welcome to ASP.NET Core
```

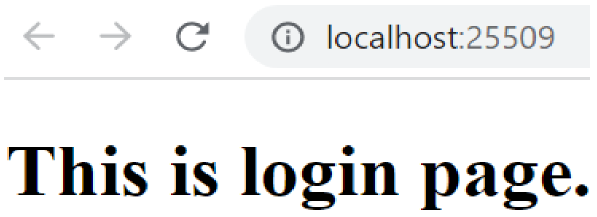
NOTE: The order of the middleware is important. Reversing them will not give the result as of this version of ASP.NET Core.
UseDefaultFiles middleware is used to point the default request path to the default file. It is not used to serve the static file.

UseFileServer - Example

Startup.cs

```
public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    FileServerOptions filesOptions = new FileServerOptions();
    filesOptions.DefaultFilesOptions.DefaultFileNames.Clear();
    filesOptions.DefaultFilesOptions.DefaultFileNames.Add("login.html");

    app.UseFileServer(filesOptions);

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync(
            "Welcome to ASP.NET Core");
    });
}
```

Browser Output


localhost:25509

This is login page.

NOTE: The app.UseDefaultFiles(filesOptions); and app.UseStaticFiles(); is been replaced with app.UseFileServer();

The options for this middle ware uses **Options** as keyword for example, **UseFileServer** takes **FileServerOptions** as parameter. In the same way, **UseDefaultFiles** takes **DefaultFilesOptions** as parameter.

Creating non default page as default

- **UseDefaultFiles** had 2 overrides. One among them takes **DefaultFilesOptions** as parameter
- DefaultFilesOptions has property **DefaultFileNames,** an ordered list of file names to select by default

```
DefaultFilesOptions filesOptions = new DefaultFilesOptions();
filesOptions.DefaultFileNames.Clear();
filesOptions.DefaultFileNames.Add("login.html");
app.UseDefaultFiles(filesOptions);
```

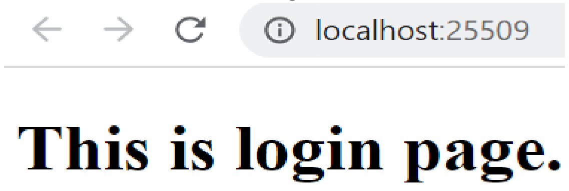
Example

Example

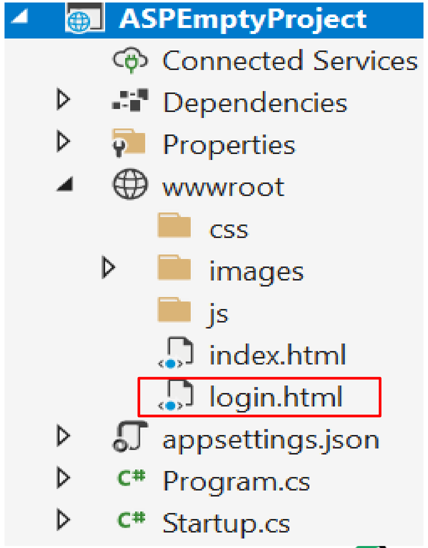
Startup.cs

```
public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    DefaultFilesOptions filesOptions = new DefaultFilesOptions();
    filesOptions.DefaultFileNames.Clear();
    filesOptions.DefaultFileNames.Add("login.html");
    app.UseDefaultFiles(filesOptions);
    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync(
            "Welcome to ASP.NET Core");
    });
}
```

Browser Output


localhost:25509

This is login page.



UseFileServer()

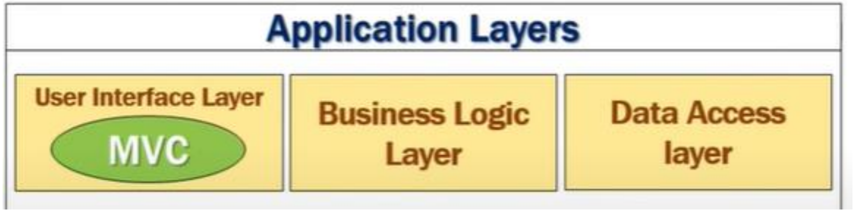
- **UseFileServer** middleware combines the functionality of **UseDefaultFiles**, **UseStaticFiles** and **UseDirectoryBrowser** middlewares
- **UseDirectoryBrowser** – this enables directory browsing and allows the user to see the list of files or folders in a specified directory
- UseFileServer take **FileServerOption** as parameter

Example

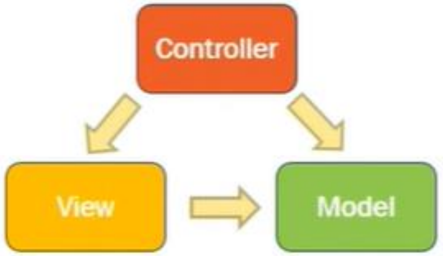
```
FileServerOptions filesOptions = new FileServerOptions();
filesOptions.DefaultFilesOptions.DefaultFileNames.Clear();
filesOptions.DefaultFilesOptions.DefaultFileNames.Add("login.html");

app.UseFileServer(filesOptions);
```

UseMvc()

- MVC is an architectural design pattern for implementing User Interface Layer of an application



- It consists of 3 parts
  - Model – Set of classes that represent data + the logic to manage that data
  - View – Contains the display logic to present the Model data provided to it by the Controller
  - Controller – Handles the http request, work with the model and selects a view to render that model

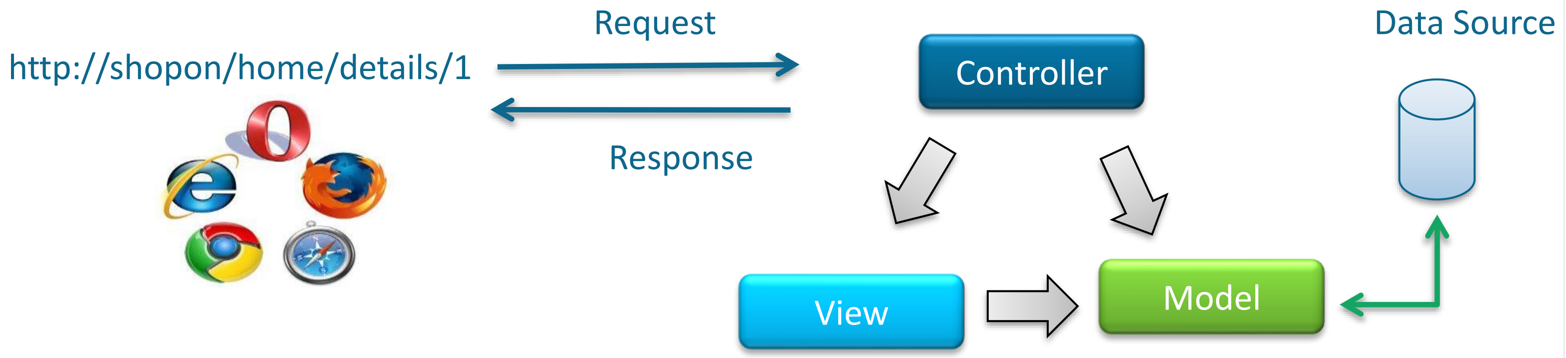


How it works

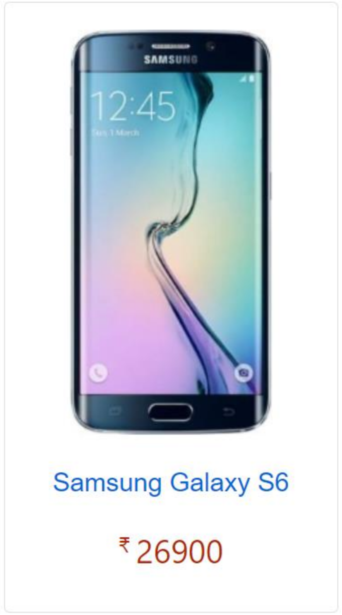
Configure MVC

Configure Routing

How it works



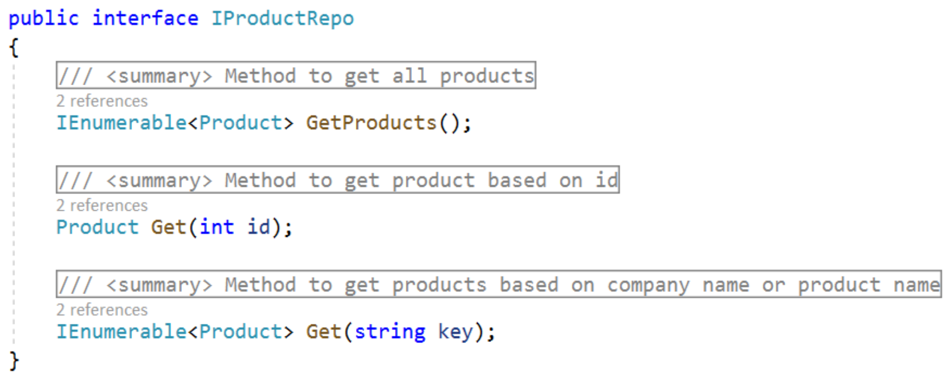
Request
http://shopon/home/details/1
Response
Controller
View
Model
Data Source



Samsung Galaxy S6
₹ 26900

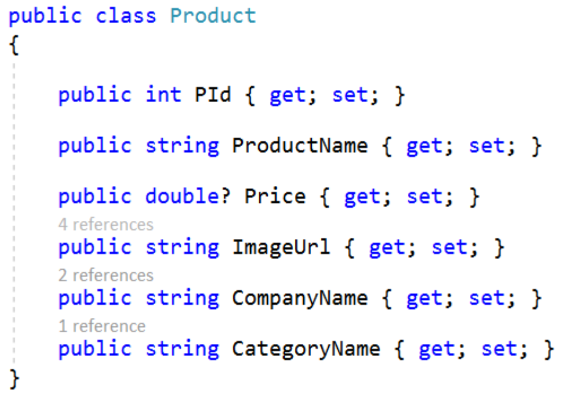
Model

IProductRepo.cs



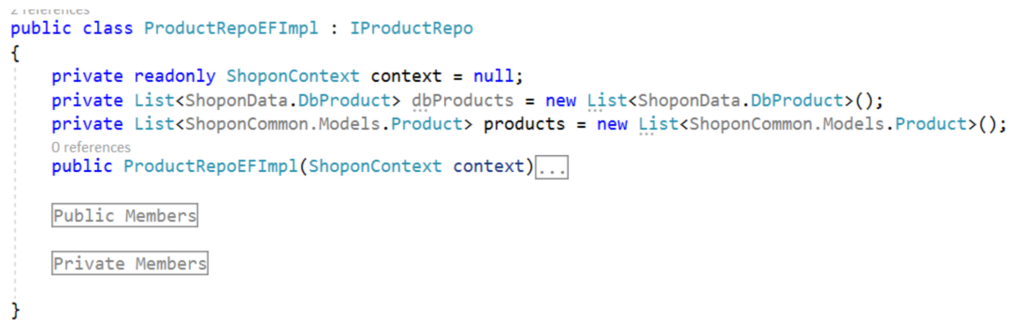
```
public interface IProductRepo
{
    /// <summary> Method to get all products
    2 references
    IEnumerable<Product> GetProducts();

    /// <summary> Method to get product based on id
    2 references
    Product Get(int id);

    /// <summary> Method to get products based on company name or product name
    2 references
    IEnumerable<Product> Get(string key);
}
```

Product.cs



```
public class Product
{

    public int PId { get; set; }

    public string ProductName { get; set; }

    public double? Price { get; set; }
    4 references
    public string ImageUrl { get; set; }
    2 references
    public string CompanyName { get; set; }
    1 reference
    public string CategoryName { get; set; }
}
```

ProductRepoImpl.cs



```
2 references
public class ProductRepoEFImpl : IProductRepo
{
    private readonly ShoponContext context = null;
    private List<ShoponData.DbProduct> dbProducts = new List<ShoponData.DbProduct>();
    private List<ShoponCommon.Models.Product> products = new List<ShoponCommon.Models.Product>();
    0 references
    public ProductRepoEFImpl(ShoponContext context)...

    Public Members

    Private Members

}
```

Model

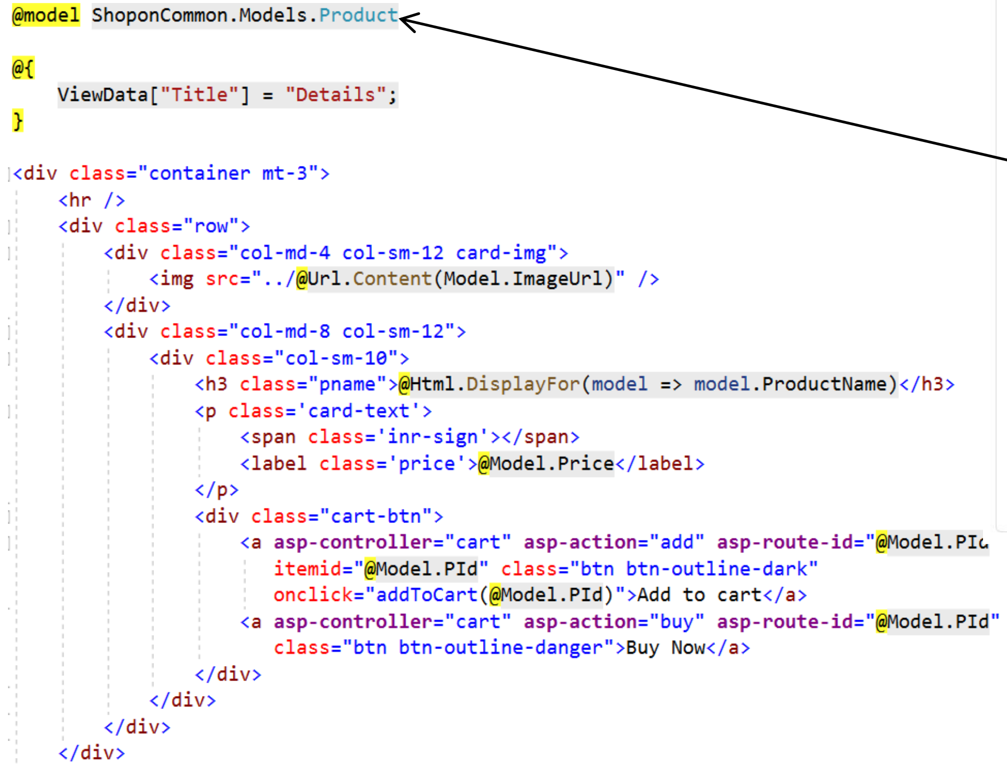
Product

+

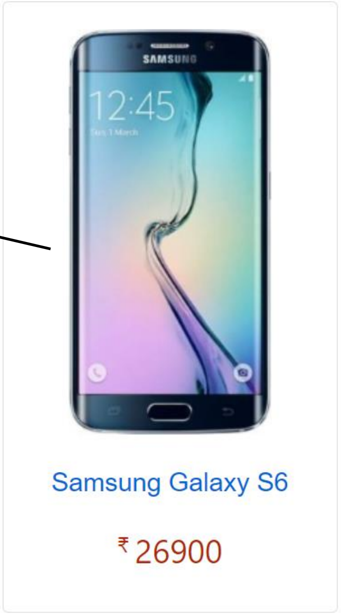
ProductRepository

View

Details.cshtml

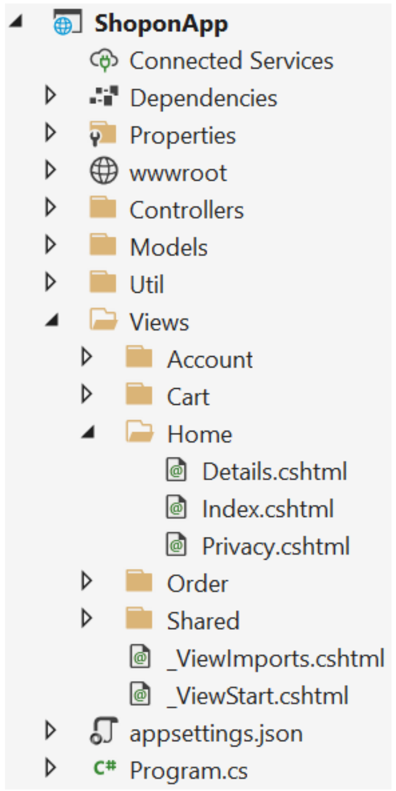


```
@model ShoponCommon.Models.Product

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

<div class="container mt-3">
    <hr />
    <div class="row">
        <div class="col-md-4 col-sm-12 card-img">
            <img src="../@Url.Content(Model.ImageUrl)" />
        </div>
        <div class="col-md-8 col-sm-12">
            <div class="col-sm-10">
                <h3 class="pname">@Html.DisplayFor(model => model.ProductName)</h3>
                <p class='card-text'>
                    <span class='inr-sign'></span>
                    <label class='price'>@Model.Price</label>
                </p>
                <div class="cart-btn">
                    <a asp-controller="cart" asp-action="add" asp-route-id="@Model.PI
                       itemid="@Model.PId" class="btn btn-outline-dark"
                       onclick="addToCart(@Model.PId)">Add to cart</a>
                    <a asp-controller="cart" asp-action="buy" asp-route-id="@Model.PId"
                       class="btn btn-outline-danger">Buy Now</a>
                </div>
            </div>
        </div>
    </div>
</div>
```



Samsung Galaxy S6

₹ 26900

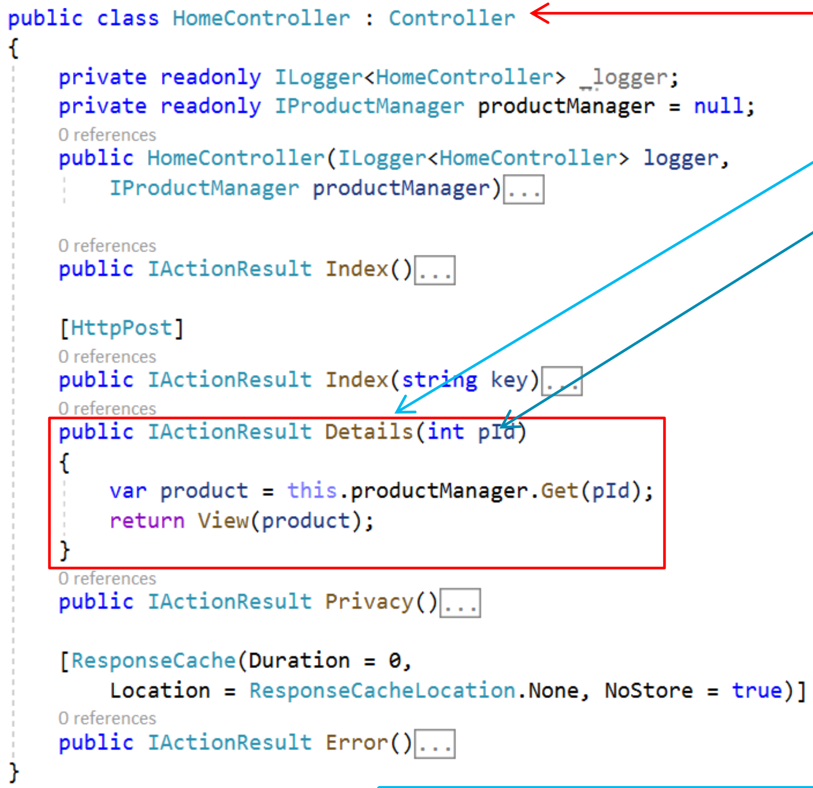


```
◢ ShoponApp
    Connected Services
  ▷ Dependencies
  ▷ Properties
  ▷ wwwroot
  ▷ Controllers
  ▷ Models
  ▷ Util
  ◢ Views
    ▷ Account
    ▷ Cart
    ◢ Home
        Details.cshtml
        Index.cshtml
        Privacy.cshtml
    ▷ Order
    ▷ Shared
      _ViewImports.cshtml
      _ViewStart.cshtml
  ▷ appsettings.json
  ▷ Program.cs
```

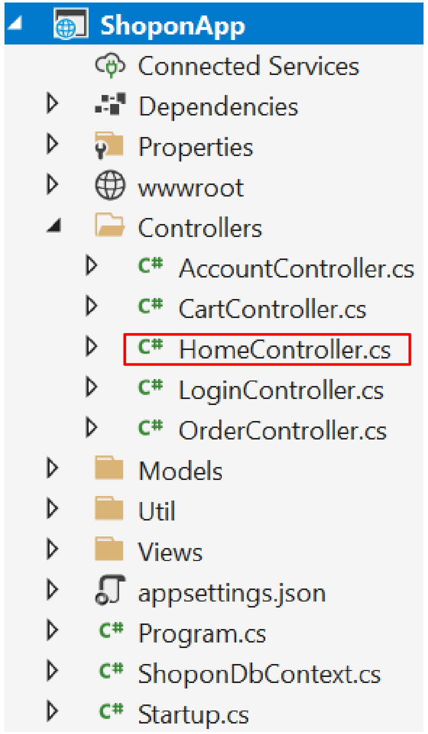
Controller

HomeController.cs

http://shopon/home/details?pid=1



```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly IProductManager productManager = null;
    0 references
    public HomeController(ILogger<HomeController> logger,
        IProductManager productManager)...

    0 references
    public IActionResult Index()...

    [HttpPost]
    0 references
    public IActionResult Index(string key)...
    0 references
    public IActionResult Details(int pId)
    {
        var product = this.productManager.Get(pId);
        return View(product);
    }
    0 references
    public IActionResult Privacy()...

    [ResponseCache(Duration = 0,
        Location = ResponseCacheLocation.None, NoStore = true)]
    0 references
    public IActionResult Error()...
}
```



```
ShoponApp
  Connected Services
  Dependencies
  Properties
  wwwroot
  Controllers
    AccountController.cs
    CartController.cs
    HomeController.cs
    LoginController.cs
    OrderController.cs
  Models
  Util
  Views
  appsettings.json
  Program.cs
  ShoponDbContext.cs
  Startup.cs
```

Routing Rules map URLs to Controller Action Method

Configure MVC

- Configuring MVC can be done using

Add MVC

Add MvcCore

- Once the MVC is configured we

Add Models

Create Views

Configure MVC

- To configure MVC, we must
  - Add MVC services to DI container(ConfigureServices())

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options => options.EnableEndpointRouting = false);
}
```

  - Add MVC middleware to requesting pipeline

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseStaticFiles();
    app.UseMvcWithDefaultRoute();

    app.Run(async (context) =>
        await context.Response.WriteAsync("Hello from ASP.NET Core")
    );
}
```

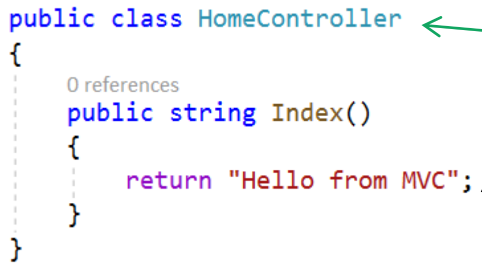
NOTE:
app.UseMvcWithDefaultRoute(); should be placed before UseStaticFiles middleware.

Add MVC Controller

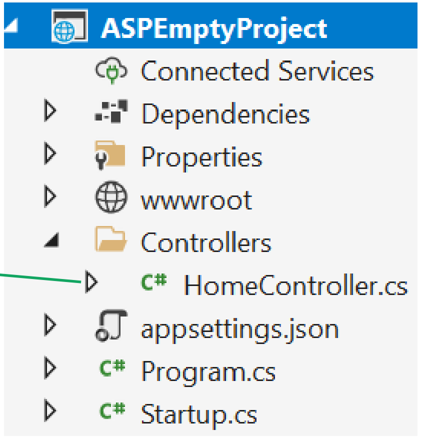
Add MVC Controller

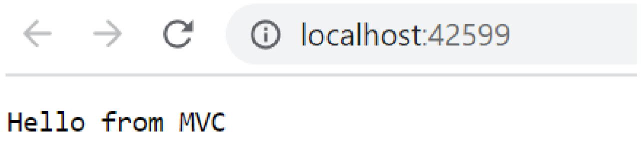
- Add new folder called ***Controllers*** in the project root folder
- Add new Controller with the name "***HomeController***" (HomeController.cs)
- Add new Index method in the HomeController

HomeController.cs



```
public class HomeController
{
    0 references
    public string Index()
    {
        return "Hello from MVC";
    }
}
```

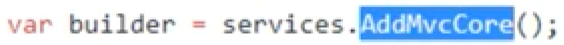




```
Hello from MVC
```

Configure MVC Core

- **addMvcCore()** - adds the minimum essential MVC services to the specified Microsoft.Extensions.DependencyInjection.IServiceCollection
- Additional services including MVC's support for authorization, formatters, and validation must be added separately using the Microsoft.Extensions.DependencyInjection.IMvcCoreBuilder returned from this method
- If we use addMvc() method, it adds all the required MVC services and MvcCore methods as well, as AddMvc internally calls AddMvcCore() method



```
var builder = services.AddMvcCore();
```

For more details, follow the link:
https://github.com/aspnet/Mvc/blob/release/2.2/src/Microsoft.AspNetCore.Mvc/MvcServiceCollectionExtensions.cs

UseEndpoints()

- Routing is responsible for matching incoming HTTP requests and dispatching those requests to the app's executable endpoints
- Endpoints are the app's units of executable request-handling code
- Endpoints are defined in the app and configured when the app starts
- The endpoint matching process can extract values from the request's URL and provide those values for request processing

Configure Endpoints

Configure Endpoints

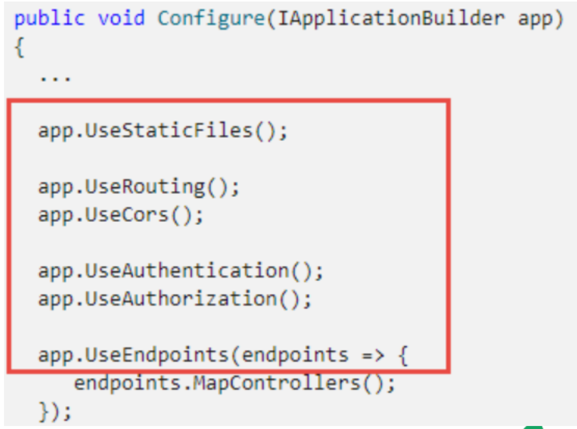
- To configure Endpoints we must
  - Register AddControllers or AddControllersWithViews

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    //services.AddControllersWithViews();
}
```

  - Add UseEndpoints middleware

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

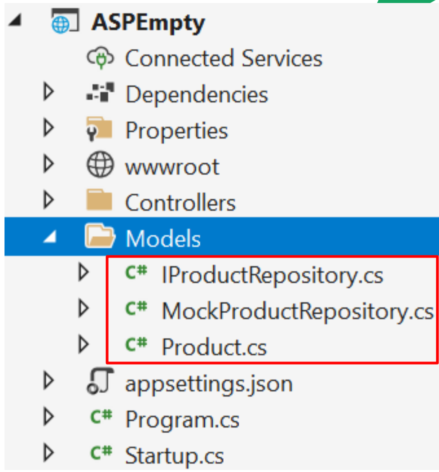
NOTE:
app.UseRouting(); should be used before app.UseEndpoints configuration as



```
public void Configure(IApplicationBuilder app)
{
  ...

  app.UseStaticFiles();

  app.UseRouting();
  app.UseCors();

  app.UseAuthentication();
  app.UseAuthorization();

  app.UseEndpoints(endpoints => {
      endpoints.MapControllers();
  });
```

Adding Model class to our project

- As we all know, model represents data and logic to fetch data from different data source
- All models may be placed in folder ***Models*** in our project
- Model will also have other class which help in getting the data
- We use interface and its implementation to get the data from different sources


```
◢ ASPEmpty
    Connected Services
  ▷ Dependencies
  ▷ Properties
  ▷ wwwroot
  ▷ Controllers
  ◢ Models
    ▷ C# IProductRepository.cs
    ▷ C# MockProductRepository.cs
    ▷ C# Product.cs
  ▷ appsettings.json
  ▷ C# Program.cs
  ▷ C# Startup.cs
```

Product.cs

```
public class Product
{
    6 references
    public int Id { get; set; }
    5 references
    public string ProductName { get; set; }
    5 references
    public string ImageUrl { get; set; }
    5 references
    public double Price { get; set; }
}
```

IProductRepository.cs

```
public interface IProductRepository
{
    1 reference
    Product GetProduct(int id);
}
```

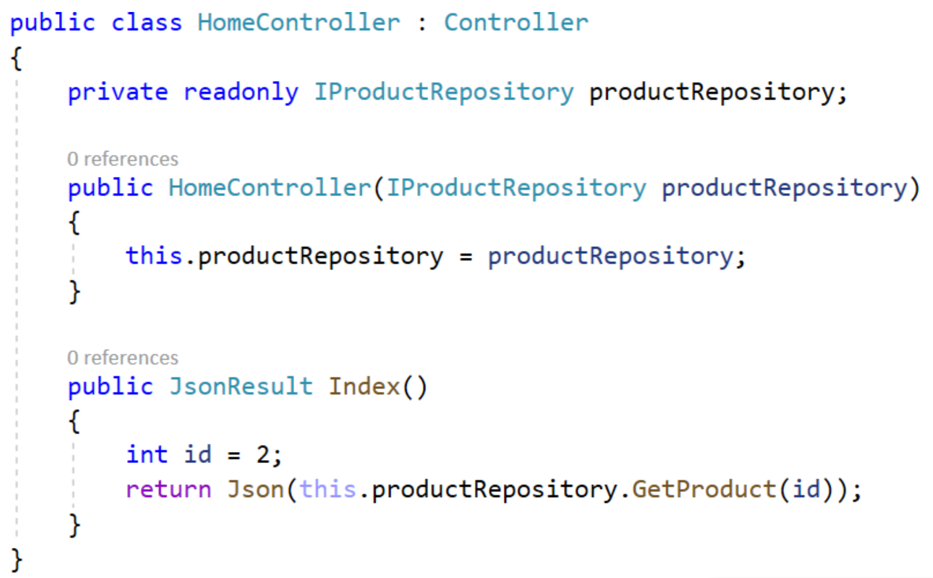
MockProductRepository.cs

```
public class MockProductRepository : IProductRepository
{
    private List<Product> products;

    0 references
    public MockProductRepository()
    {
        this.products = new List<Product>()...;
    }

    1 reference
    public Product GetProduct(int id)
    {
        return this.products.FirstOrDefault(x => x.Id == id);
    }
}
```

Model in Controller

Models in Controller

- To add model in controller, we use DI pattern
- We use Constructor based DI or Constructor Injection in MVC Core application often



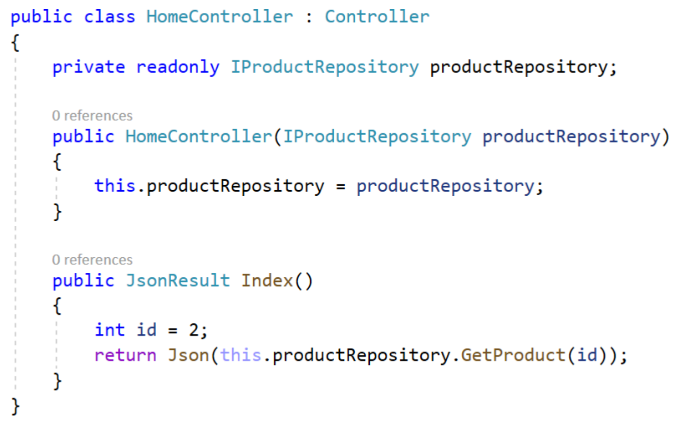
```
public class HomeController : Controller
{
    private readonly IProductRepository productRepository;

    0 references
    public HomeController(IProductRepository productRepository)
    {
        this.productRepository = productRepository;
    }

    0 references
    public JsonResult Index()
    {
        int id = 2;
        return Json(this.productRepository.GetProduct(id));
    }
}
```

Dependency Injection

Dependency Injection (DI)

- ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies
- Services are added as a **constructor parameter**, and the runtime resolves the service from the service container. Services are typically defined using interfaces

In HomeController, we need ProductRepository which is injected in the Constructor. This is called as **Constructor Injection**
We must register the implementation so that ASP.NET Core knows which is the implementation that should be injected



```
public class HomeController : Controller
{
    private readonly IProductRepository productRepository;

    0 references
    public HomeController(IProductRepository productRepository)
    {
        this.productRepository = productRepository;
    }

    0 references
    public JsonResult Index()
    {
        int id = 2;
        return Json(this.productRepository.GetProduct(id));
    }
}
```

Dependency Injection (DI)

- Registering DI – To register the dependencies, we will use **ConfigureServices** method
- ASP.NET Core allows us to register with Dependency Injection Container using
  - **AddSingleton()** – Singleton service is created when it is first requested. The same instance is used by all subsequent request
  - **AddScoped()** – A new instance of Scoped service is created once per request within the scope
  - **AddTransient()** – Creates a Transient service. A new instance of transient service is created each time it is requested

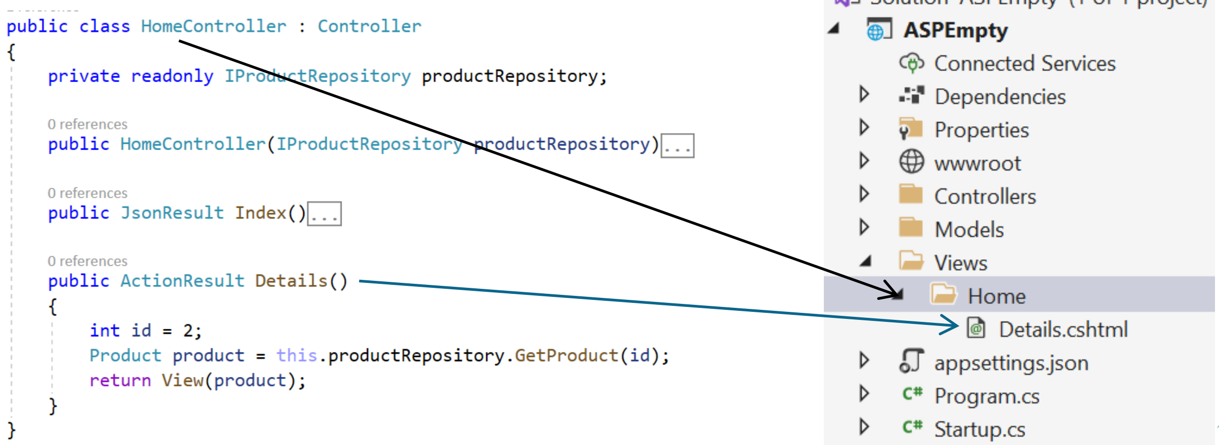
  More details - https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0

- Benefits of DI
  - Loose Coupling
  - Easy to Unit Test

Creating View

- In general, view represents presentation or display of the modal data in a specific format
- In MVC, view is a file with the extension **.cshtml** or **.vbhtml** based on the programming language used for coding
- As per the default convention of ASP.NET Core, all view files will be present in **Views** folder
- Based on the controller, respective folders will be present. Each Action method will have a View file created. Thus, one View folder associated to a controller will have all the View file(s) in it

To add new view, right click on the
Home folder →
Add New Item →
Razor View - Empty



```
public class HomeController : Controller
{
    private readonly IProductRepository productRepository;

    0 references
    public HomeController(IProductRepository productRepository)[...]

    0 references
    public JsonResult Index()[...]

    0 references
    public ActionResult Details()
    {
        int id = 2;
        Product product = this.productRepository.GetProduct(id);
        return View(product);
    }
}
```

Creating View

- A View file has .cshtml as extension
- A View is an html template with Razor markup
- Contains the logic to display the model data
- By default, ASP.NET Core uses conventional way to map the view. If we want to change it, we can the Customize View

Passing Data from Controller to View

There are 3 ways to do so

1. ViewData
2. ViewBag
3. Strongly Typed View

$\left.\begin{array}{l}\text{ViewData}\\ \text{ViewBag}\end{array}\right\}$ **Loosely typed view**

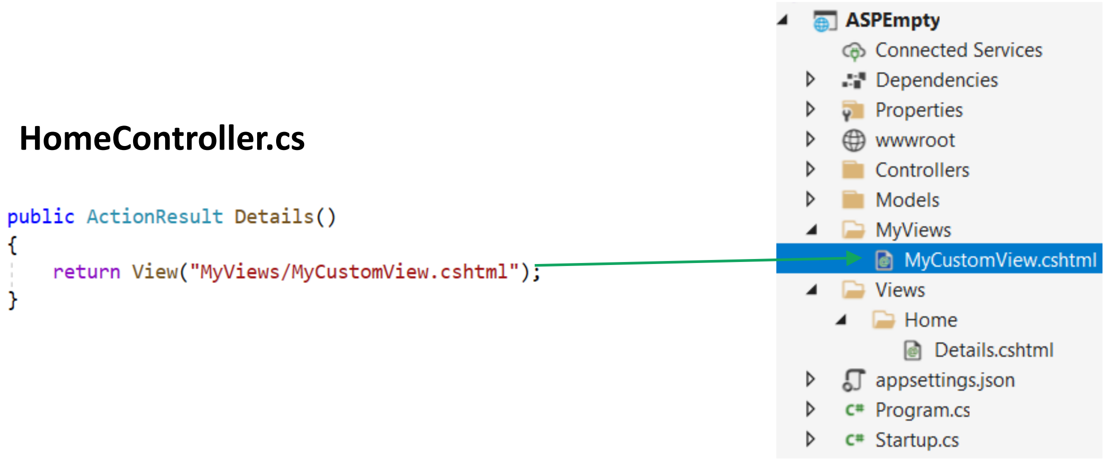
Customize View Discovery

- View() or View(object model) – Looks for a view file with the same name as the action method
- View(string viewName) –
  - Looks for a view file with our own custom name
  - We can specify a view name or a view file path
  - View file path can be absolute or relative
  - With absolute path .cshtml extension must be specified
  - With relative path, do not specify the file extension .cshtml
- View(string viewName, object model) – used to pass model object to the view

Customize View Discovery using Absolute path

- To customize view, we can use the view in different folder apart from the Views folder
- We can mention absolute path in following ways
  - MyViews/MyCustomView.cshtml
  - /MyViews/MyCustomView. cshtml
  - ~/MyViews/MyCustomView. cshtml

HomeController.cs



```
public ActionResult Details()
{
    return View("MyViews/MyCustomView.cshtml");
}
```

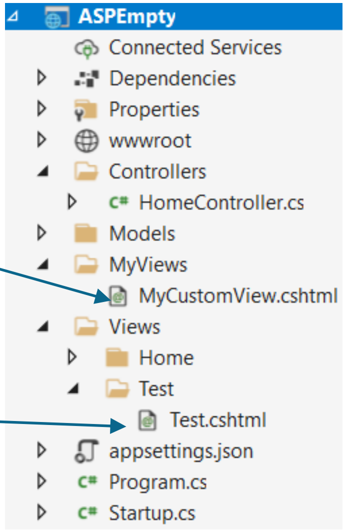
Customize View Discovery using Relative path

- ASP.NET MVC Core will always look for view file from Views folder
- We should always map the file from the Views Controllers folder to another folder
- We shall not specify the extensions

HomeController.cs

```
public ActionResult Details()
{
    return View("../../MyViews/MyCustomView");
}
```

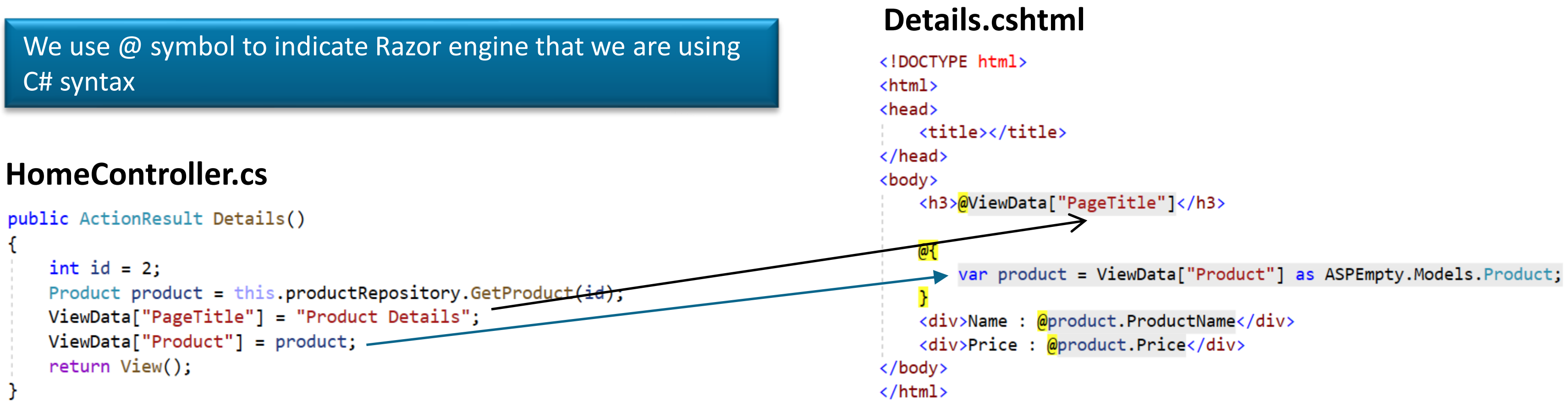
```
public ActionResult Details()
{
    return View("../Test/Test");
}
```



ViewData

- This is one of the way to pass data from controller to view
- ViewData is Dictionary of weekly typed object(s)
- Use string keys to store and retrieve the data
- Type casting is required if we are extracting any other data type apart from **string** type
- Dynamically resolved at runtime

We use @ symbol to indicate Razor engine that we are using C# syntax



HomeController.cs

```
public ActionResult Details()
{
    int id = 2;
    Product product = this.productRepository.GetProduct(id);
    ViewData["PageTitle"] = "Product Details";
    ViewData["Product"] = product;
    return View();
}
```

Details.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h3>@ViewData["PageTitle"]</h3>

    @{
        var product = ViewData["Product"] as ASPEmpty.Models.Product;
    }
    <div>Name : @product.ProductName</div>
    <div>Price : @product.Price</div>
</body>
</html>
```

ViewBag

- ViewBag is a wrapper around ViewData
- ViewBag is **Dynamic** type present in **`Microsoft.AspNetCore.Mvc.Controller`** class. Using this we can create dynamic properties
- With ViewBag, type casting is not required while fetching the data from it
- ViewBag returns **null** if the property does not exist

ViewData v/s ViewBag

HomeController.cs

```
public ActionResult Details()
{
    int id = 2;
    Product product = this.productRepository.GetProduct(id);
    ViewBag.PageTitle = "Product Details";
    ViewBag.Product = product;
    return View();
}
```

Details.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h3>@ViewBag.PageTitle</h3>

    <div>Name : @ViewBag.Product.ProductName</div>
    <div>Price : @ViewBag.Product.Price</div>
</body>
</html>
```

ViewData v/s ViewBag

Differences

- ViewBag is a wrapper around ViewData
- ViewData uses string keys to store and retrieve data. Where as ViewBag uses dynamic properties to store and retrieve data

Similarities

- No compile time type checking and intellisense
- Both creates a loosely typed view
- Both resolves dynamically at runtime

Preferred approach to pass data from a controller to a view is by using **Strongly Typed View**

Strongly Typed View

- Microsoft.AspNetCore.Mvc.View has overloaded method which takes model object as parameter
- We can pass view object using this overloaded method
- In the view page, we can access this model object using **@Model** property
- To make model strongly types, we use **@model directive**

HomeController.cs

```
public ActionResult Details()
{
    int id = 2;
    Product product = this.productRepository
                          .GetProduct(id);
    return View(product);
}
```

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h3>@ViewBag.PageTitle</h3>

    <div>Name : @Model.ProductName</div>
    <div>Price : @Model.Price</div>
</body>
</html>
```

This is **not** strongly typed view. We can still access the data. Properties are **dynamic** here.

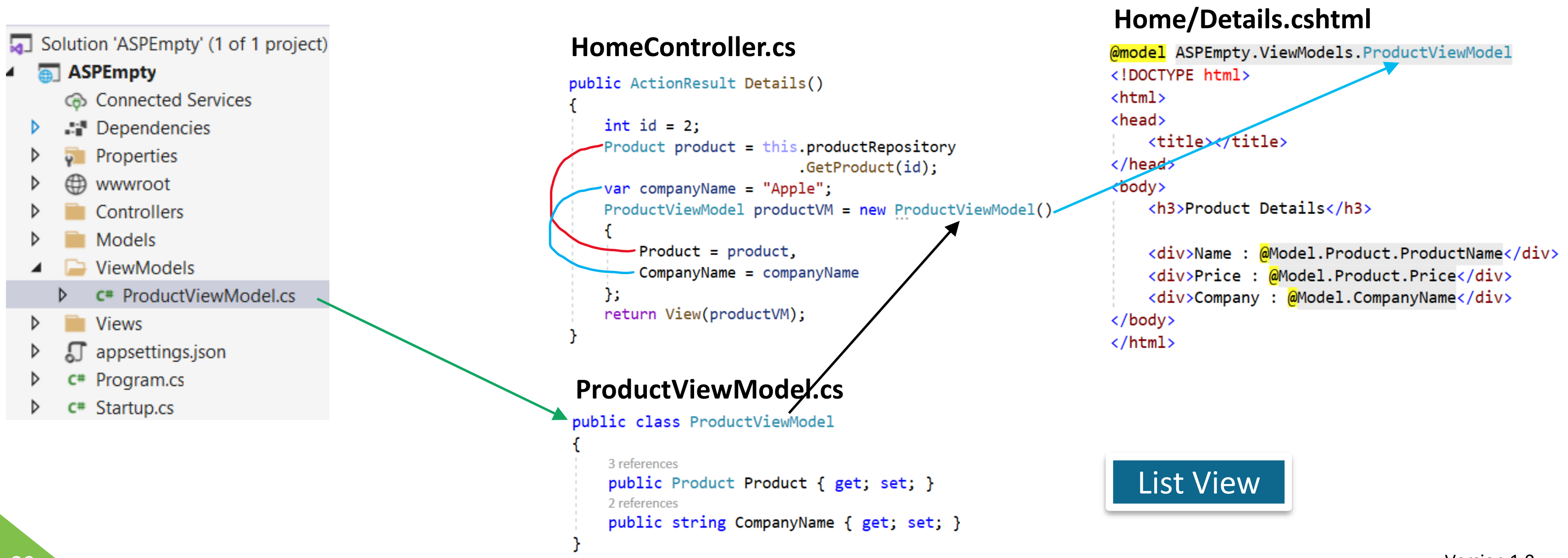
```
@model ASPEmpty.Models.Product
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h3>@ViewBag.PageTitle</h3>

    <div>Name : @Model.ProductName</div>
    <div>Price : @Model.Price</div>
</body>
</html>
```

This is strongly typed view. We can use intellisense to display the properties.

ViewModels

ViewModels

- There can be some use case where our model object may not have all the data that our view requires. That is when we create ViewModels
- ViewModels are also called as Data Transfer Objects(dto) as they are used to ***shuttle data*** between controllers and views



```
Solution 'ASPEmpty' (1 of 1 project)
◢ ASPEmpty
    Connected Services
  ▷ Dependencies
  ▷ Properties
  ▷ wwwroot
  ▷ Controllers
  ▷ Models
  ◢ ViewModels
    ▷ C# ProductViewModel.cs
  ▷ Views
  ▷ appsettings.json
  ▷ C# Program.cs
  ▷ C# Startup.cs
```

HomeController.cs

```
public ActionResult Details()
{
    int id = 2;
    Product product = this.productRepository
                          .GetProduct(id);
    var companyName = "Apple";
    ProductViewModel productVM = new ProductViewModel()
    {
        Product = product,
        CompanyName = companyName
    };
    return View(productVM);
}
```

ProductViewModel.cs

```
public class ProductViewModel
{
    3 references
    public Product Product { get; set; }
    2 references
    public string CompanyName { get; set; }
}
```

Home/Details.cshtml

```
@model ASPEmpty.ViewModels.ProductViewModel
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h3>Product Details</h3>

    <div>Name : @Model.Product.ProductName</div>
    <div>Price : @Model.Product.Price</div>
    <div>Company : @Model.CompanyName</div>
</body>
</html>
```

List View

List View

- In most of the applications, we may have to display a list of items, to implement this, we must pass collection of objects from controller to view
- In view we will loop through the collection and fetch the data

IProductRepository.cs

```
public interface IProductRepository
{
    2 references
    Product GetProduct(int id);
    2 references
    IEnumerable<Product> GetProducts();
}
```

MockProductRepository.cs

```
public class MockProductRepository : IProductRepository
{
    private List<Product> products;

    0 references
    public MockProductRepository()[...]

    2 references
    public Product GetProduct(int id)[...]

    2 references
    public IEnumerable<Product> GetProducts()
    {
        return this.products;
    }
}
```

HomeController.cs

```
public IActionResult Index()
{
    var model = this.productRepository
                    .GetProducts();
    return View(model);
}
```

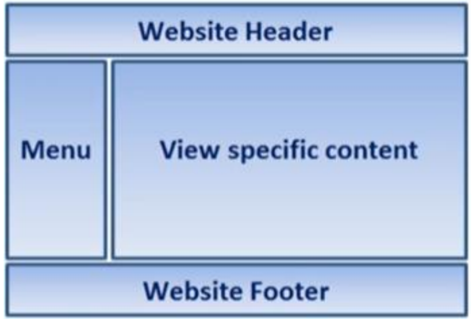
Layout View

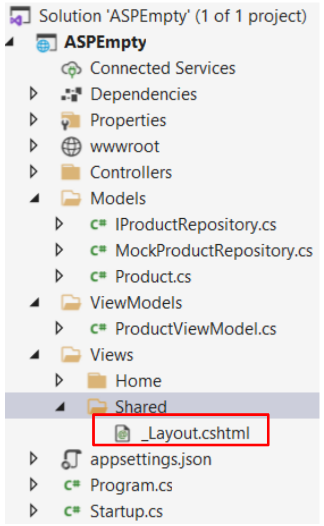
Home/Index.cshtml

```
@model IEnumerable<ASPEmpty.Models.Product>

<!DOCTYPE html>
<html>
<head>...</head>
<body>
    <table cellpadding="5" cellspacing="5" border="1">
        <thead>
            <tr>
                <th>ID</th>
                <th>Product Name</th>
                <th>Price</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var product in Model)
            {
                <tr>
                    <td>@product.Id</td>
                    <td>@product.ProductName</td>
                    <td>@product.Price</td>
                </tr>
            }
        </tbody>
    </table>
</body>
</html>
```

Layout View

- Most of the web applications will have the sections like Header, Menu, Footer in common
- It would be tedious to manage these repeated sections in all the web pages
- Creating layout view will help in managing these sections. These can be compared to a Masterpage in ASP.NET Web forms
- Layout view is like any other view page with extension .cshtml. By default, it will be named _Layout.cshtml
- They file will be placed in **Shared** sub folder inside **Views** folder. We can have more than 1 layout view





RenderBody | ViewStart

RenderSection | ViewImports

To add layout view, right click on the **Home folder →** **Add New Item →** **Razor Layout**

RenderBody

- RenderBody is used to render portion of a content page that is not within a named section

_Layout.cshtml

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ShoponApp - @ViewBag.Title</title>
</head>
<body>
    <div>
        <h2>Shopon Web App</h2>
        <hr />
    </div>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

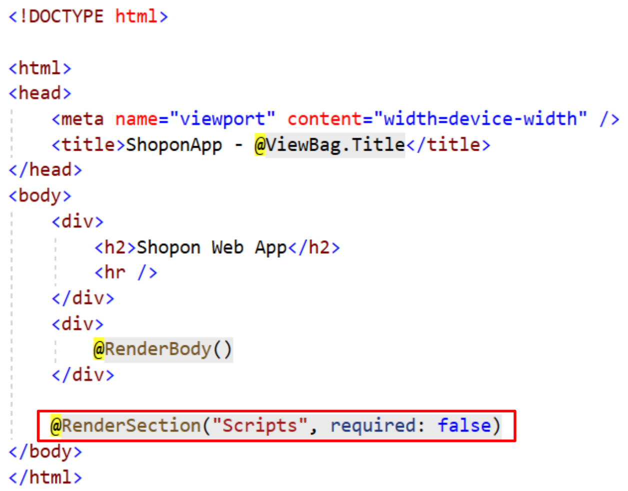
Index.cshtml

```
@model IEnumerable<ASPEmpty.Models.Product>

@{
    ViewBag.Title = "One stop shop for all your mobile accessories";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<table cellpadding="5" cellspacing="5" border="1">
    <thead>
        <tr>
            <th>ID</th>
            <th>Product Name</th>
            <th>Price</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var product in Model)
        {
            <tr>
                <td>@product.Id</td>
                <td>@product.ProductName</td>
                <td>@product.Price</td>
            </tr>
        }
    </tbody>
</table>
```

RenderSection

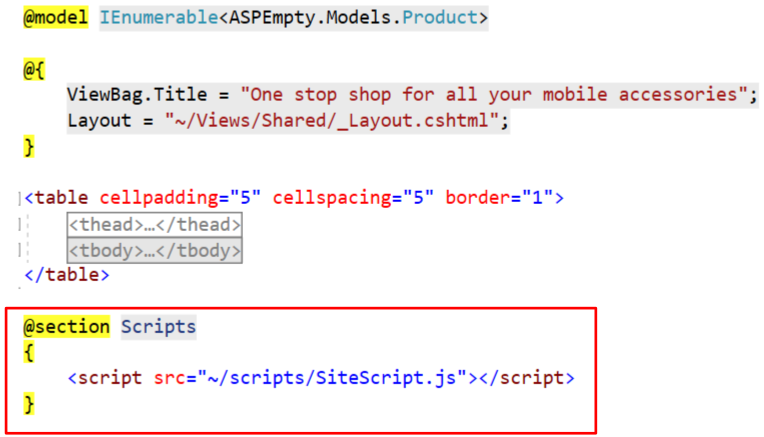
- A Section in a Layout View provides a way to organize where certain page elements should be placed
- A Section can be **optional** or **mandatory**
- A Section in the Layout View is rendered at the location where **RenderSection()** method is called

_Layout.cshtml



```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ShoponApp - @ViewBag.Title</title>
</head>
<body>
    <div>
        <h2>Shopon Web App</h2>
        <hr />
    </div>
    <div>
        @RenderBody()
    </div>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

Index.cshtml



```
@model IEnumerable<ASPEmpty.Models.Product>

@{
    ViewBag.Title = "One stop shop for all your mobile accessories";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<table cellpadding="5" cellspacing="5" border="1">
    <thead>…</thead>
    <tbody>…</tbody>
</table>

@section Scripts
{
    <script src="~/scripts/SiteScript.js"></script>
}
```

_ViewStart

- _ ViewStart.cshtml is a special file in ASP.NET Core MVC
- The code in this file gets executed before the code in individual view file is executed
- Instead of setting the property in each individual View, we can move that code into the _ViewStart file
- This file will be placed in the **Views** folder
- We can have multiple _ViewStart files in a project
- Dynamically we can load the view start file by checking conditions

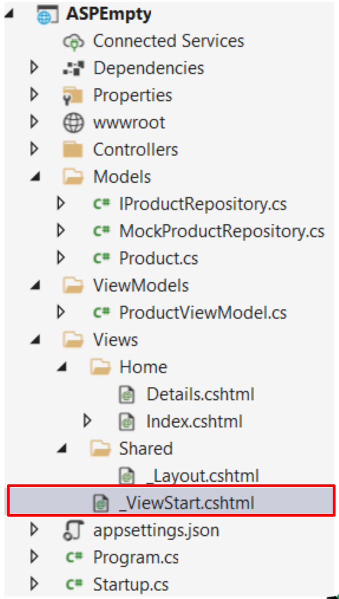
_Layout.cshtml

```
@{
    Layout = "_Layout";
}
```

_Layout.cshtml

```
@{
    if (User.IsInRole("Admin"))
    {
        Layout = "_AdminLayout";
    }
    Layout = "_Layout";
}
```

To add layout view, right click on the **Home folder →** **Add New Item →** **Razor View Start**

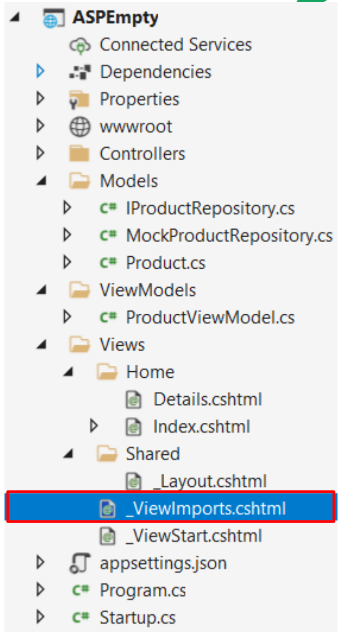


```
◢ ASPEmpty
    Connected Services
  ▷ Dependencies
  ▷ Properties
  ▷ wwwroot
  ▷ Controllers
  ◢ Models
    ▷ IProductRepository.cs
    ▷ MockProductRepository.cs
    ▷ Product.cs
  ◢ ViewModels
    ▷ ProductViewModel.cs
  ◢ Views
    ◢ Home
          Details.cshtml
      ▷ Index.cshtml
    ◢ Shared
          _Layout.cshtml
      _ViewStart.cshtml
  ▷ appsettings.json
  ▷ Program.cs
  ▷ Startup.cs
```

_ViewImports

- We use this files to **include common namespaces**, thus we don't have to include them in every view that needs those namespaces
- _ViewImports.cshtml is the file which should be included in **Views** folder

To add layout view, right click on the **Home folder →** **Add New Item →** **Razor View Imports**


```
ASPEmpty
  Connected Services
  Dependencies
  Properties
  wwwroot
  Controllers
  Models
    IProductRepository.cs
    MockProductRepository.cs
    Product.cs
  ViewModels
    ProductViewModel.cs
  Views
    Home
      Details.cshtml
      Index.cshtml
    Shared
      _Layout.cshtml
    _ViewImports.cshtml
    _ViewStart.cshtml
  appsettings.json
  Program.cs
  Startup.cs
```

_ViewImports.cshtml

```
@using ASPEmpty.Models;
@using ASPEmpty.ViewModels;
```

Details.cshtml.cshtml

```
@model ProductViewModel

@{
    ViewBag.Title = "Mobile Details";
}


<h3>Product Details</h3>

<div>Name : @Model.Product.ProductName</div>
<div>Price : @Model.Product.Price</div>
<div>Company : @Model.CompanyName</div>
```

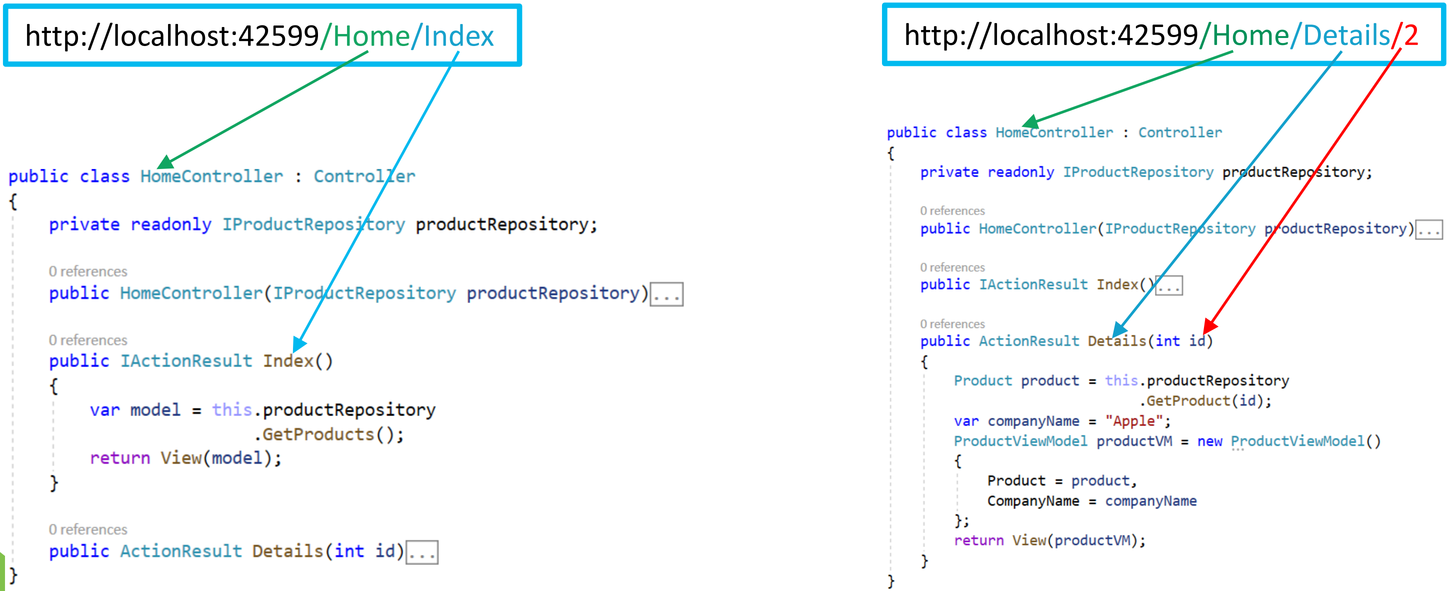
We are not specifying fully qualified name here as it is included in _ViewImports file

Configure Routing

- There are 2 routing techniques
  - Conventional Routing
  - Attribute Routing

Conventional Routing

- When the request arrives at our application, it is the controller in our application which will handle the http request and response to the user action
- The incoming request URL is mapped to a controller's action method. This mapping is done by the routing rules defined in our application



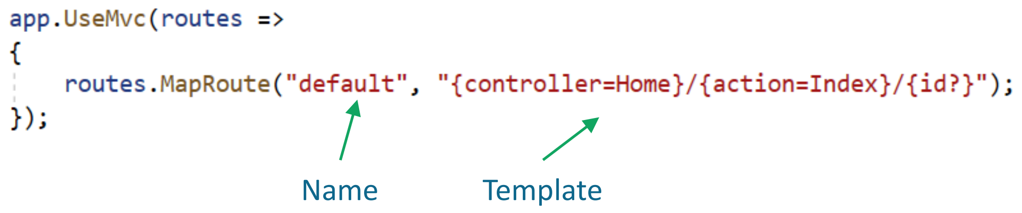
http://localhost:42599/Home/Index
http://localhost:42599/Home/Details/2

```
public class HomeController : Controller
{
    private readonly IProductRepository productRepository;

    0 references
    public HomeController(IProductRepository productRepository)...

    0 references
    public IActionResult Index()
    {
        var model = this.productRepository
                        .GetProducts();
        return View(model);
    }

    0 references
    public ActionResult Details(int id)...
}
```

```
public class HomeController : Controller
{
    private readonly IProductRepository productRepository;

    0 references
    public HomeController(IProductRepository productRepository)...

    0 references
    public IActionResult Index()...

    0 references
    public ActionResult Details(int id)
    {
        Product product = this.productRepository
                              .GetProduct(id);
        var companyName = "Apple";
        ProductViewModel productVM = new ProductViewModel()
        {
            Product = product,
            CompanyName = companyName
        };
        return View(productVM);
    }
}
```

Conventional Routing

- **app.UseMvcWithDefaultRoute()** - Adds MVC to the **Microsoft.AspNetCore.Builder.IApplicationBuilder** request execution pipeline with a default route named '**default**' and the following template: **'{controller=Home}/{action=Index}/{id?}'**
- **app.UseMvc()** - Adds MVC to the **Microsoft.AspNetCore.Builder.IApplicationBuilder** request execution pipeline. *This will not add any default route support to our application*. It takes IRouteBuilder as parameter, in return has **MapRoute()** method, using which we can customize our route template



```
app.UseMvc(routes =>
{
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

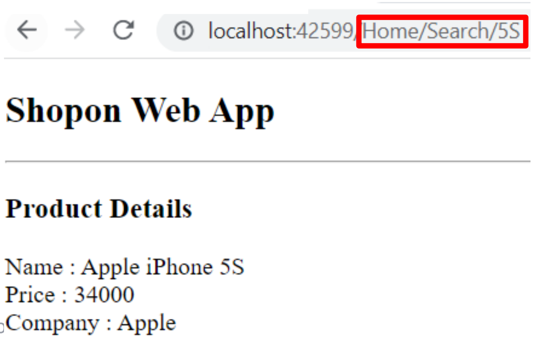
Attribute Routing

- Attribute routing is a customized way to route to a specific Action method within a controller
- To achieve attribute routing, we use **[Route]** attribute
- Route attribute takes *template* as parameter

```
public RouteAttribute(string template);
```

- If the route is common(like controller name), that can be attributed in Controller

```
[Route("Home")]
0 references
public class HomeController : Controller
{ }
```


localhost:42599/Home/Search/5S

Shopon Web App

Product Details

Name : Apple iPhone 5S
Price : 34000
Company : Apple

HomeController.cs

```
[Route("Home/Search/{key}")]
0 references
public ActionResult Details(string key)
{
    var products = this.productRepository
                        .GetProducts();
    var product = products.FirstOrDefault(x => x.ProductName.Contains(key));
    var companyName = "Apple";
    ProductViewModel productVM = new ProductViewModel()
    {
        Product = product,
        CompanyName = companyName
    };
    return View(productVM);
}
```

NOTE: If in controller Home is mentioned, then avoid specifying Home in the Action method.

Next Step

Recap

Useful links

Thank you

Recap

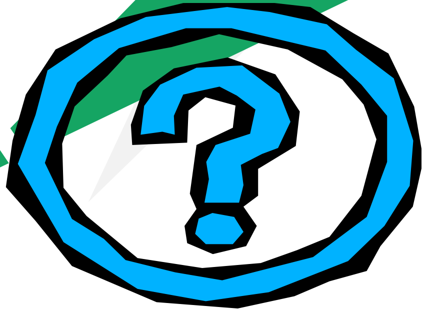
- Till now we have understood
  - Understanding of .NET Core
  - .NET Core features
  - Creating ASP.NET Core app
    - Using CLI
    - Using Visual Studio
  - .NET Core project structure
  - .NET Core file structure
  - .NET Core deployment

Useful Links

- https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0
- https://docs.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-6.0&tabs=windowshttps://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-6.0
- https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host?view=aspnetcore-6.0
- https://dotnet.microsoft.com/en-us/platform/community
- https://github.com/dotnet/aspnetcore
- https://dotnet.microsoft.com/en-us/download
- https://docs.microsoft.com/en-us/dotnet/core/tools/
- https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/aspnet-core-module?view=aspnetcore-6.0#:~:text=ASP.NET%20Core%20apps%20default,used%20instead%20of%20Kestrel%20server.
- https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/iis/in-process-hosting?view=aspnetcore-6.0

Thank YOU

Any Questions?



XORIANT

US – CORPORATE HEADQUARTERS

1248 Reamwood Avenue,
Sunnyvale, CA 94089
Phone: (408) 743 4400

343 Thornall St 720
Edison, NJ 08837
Phone: (732) 395 6900

UK

57 Rathbone Place,
4th Floor, Holden House,
London, W1T 1JU , UK

89 Worship Street Shoreditch,
London EC2A 2BF, UK
Phone: (44) 2079 938 955

INDIA

Mumbai
4th Floor, Nomura
Powai , Mumbai 400 076
Phone: +91 (22) 3051 1000

Pune
5th floor, Amar Paradigm Baner Road
Baner, Pune 411 045
Phone: +91 (20) 6604 6000

Bangalore
4th Floor, Kabra Excelsior,
80 Feet Main Road, Koramangala 1st Block,
Bengaluru (Bangalore) 560034
Phone: +91 (80) 4666 1666

www.xoriant.com