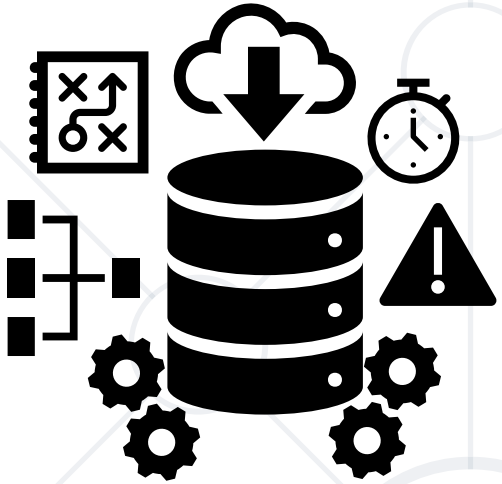
Routing and Binding, Views, DI and Services

Custom Model Binding and Validation, Files, Razor Syntax, Special Views, Routing and Dependency Injection



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

Have a Question?

Software University

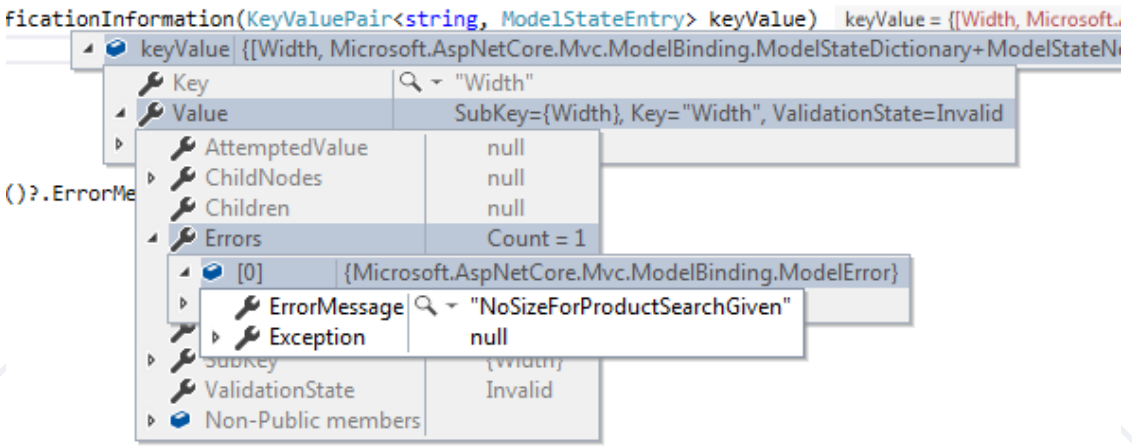
sli.do

#csharp-web

Custom Model Binding

Model Binding Overview

- Bridge between **HTTP request** and **action method parameters**
- Data from HTTP requests is used by controllers
  - Retrieved from **route data**, **form fields**, **query strings**, etc.
- Request data is bound to action parameters by **name**
  - If binding is **not** successful, an error is **not** thrown



- The model binding behavior can be **customized**

Attributes

- Built-in Model binding behavior can be directed to a different source
  - The framework provides several attributes for that

Attribute	Description
`[BindRequired]`	Adds a **model state error** if binding cannot occur.
`[BindNever]`	Tells the model binder to **never bind** this parameter.
`[From{source}]`	Used to specify the exact binding source. `[FromHeader]`, `[FromQuery]`, `[FromRoute]`, `[FromForm]`
`[FromServices]`	Uses **dependency injection** to bind parameters from services.
`[FromBody]`	Use configure formatters to bind data from request body. Formatter is selected based on **Content-Type** of Request.
`[ModelBinder]`	Used to override the default **model binder**, **binding source** and **name**.

Custom Model Binder (1)

- **Custom Model Binding** can be completely customized
  - You need to create a **BindingProvider** and a **Binder**

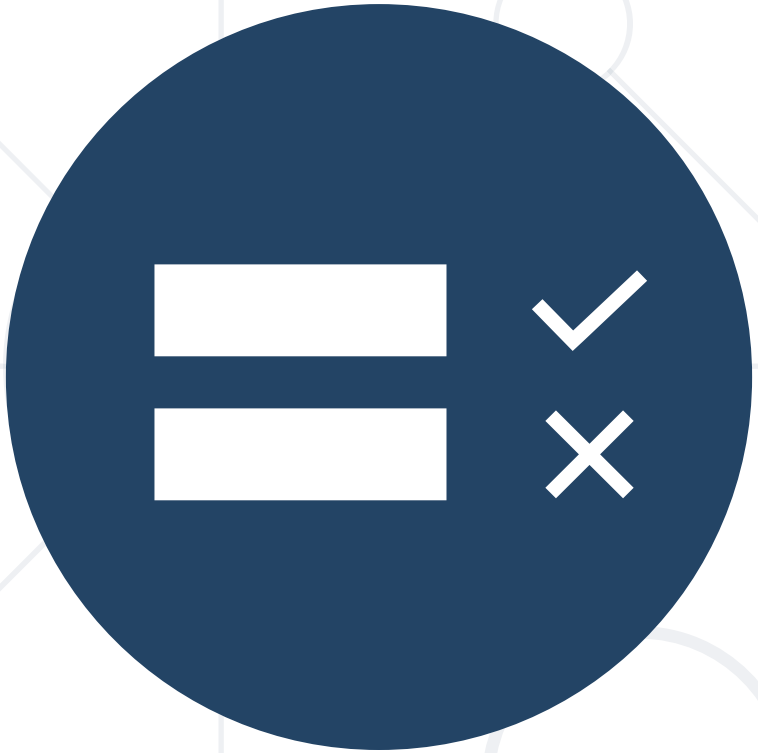
```
[ModelBinder(BinderType = typeof(StudentEntityBinder))]
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

```
public class StudentEntityBinder : IModelBinder
{
    public Task BindModelAsync
         (ModelBindingContext bindingContext)
    {
        // TODO: Do Magic ...
        bindingContext.Result
            = ModelBindingResult.Success(model);
        return Task.CompletedTask;
    }
}
```

Custom Model Binder (2)

```
public class StudentEntityBinderProvider : IModelBinderProvider
{
    public IModelBinder GetBinder(ModelBinderProviderContext context)
    {
        if(context == null)
        {
            throw new ArgumentNullException(nameof(context));
        }
        if(context.Metadata.ModelType == typeof(Student))
        {
            return new BinderTypeModelBinder(typeof(StudentEntityBinder));
        }
        return null;
    }
}
```

```
services.AddControllerWithViews(options =>
{
    options.ModelBinderProviders
           .Insert(0, new StudentEntityBinderProvider());
    // Add custom binder to beginning
});
```



Custom Model Validation

Model Validation Overview

- Model validation occurs **after** model binding
  - Reports **errors** that originate from model binding
- Two types of validation
    - Server-side
    - Client-side
- `ModelState.IsValid` property indicates if the model validation is successful
  - Iterates over the errors

Custom Model Validation (1)

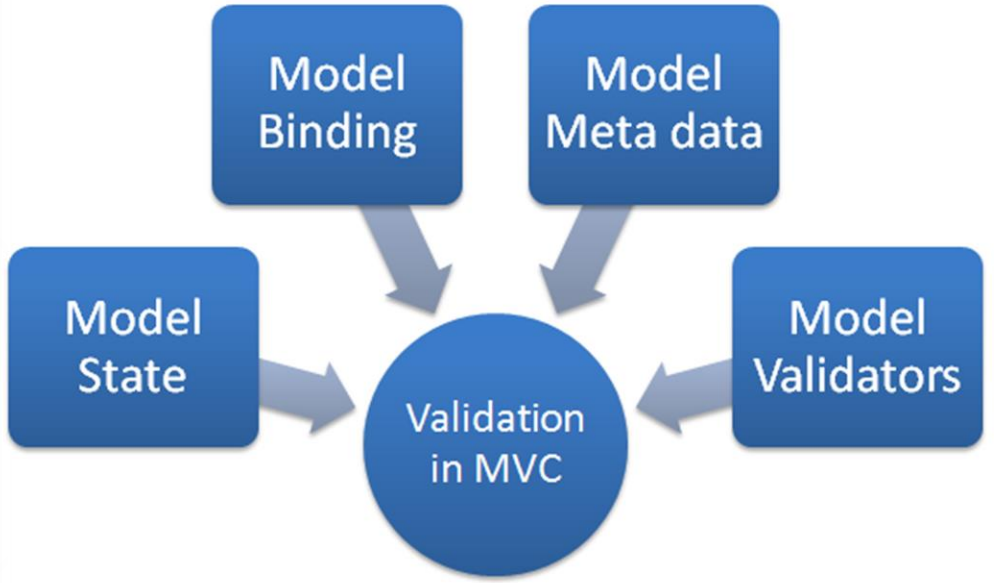
- **Validation attributes** work for most needs, but not for all
  - Sometimes you need to implement your own validation attributes

```
public class IsBefore : ValidationAttribute
{
    private const string DateTimeFormat = "dd/MM/yyyy";
    private readonly DateTime date;

    public IsBefore(string dateInput)
    {
        date = DateTime.ParseExact(dateInput, DateTimeFormat, CultureInfo.InvariantCulture);
    }

    protected override ValidationResult IsValid(object value, ValidationContext validationContext)
    {
        if ((DateTime)value >= date) return new ValidationResult(ErrorMessage);
        return ValidationResult.Success;
    }
}
```

Custom Model Validation (2)

- Then you can use it in your model

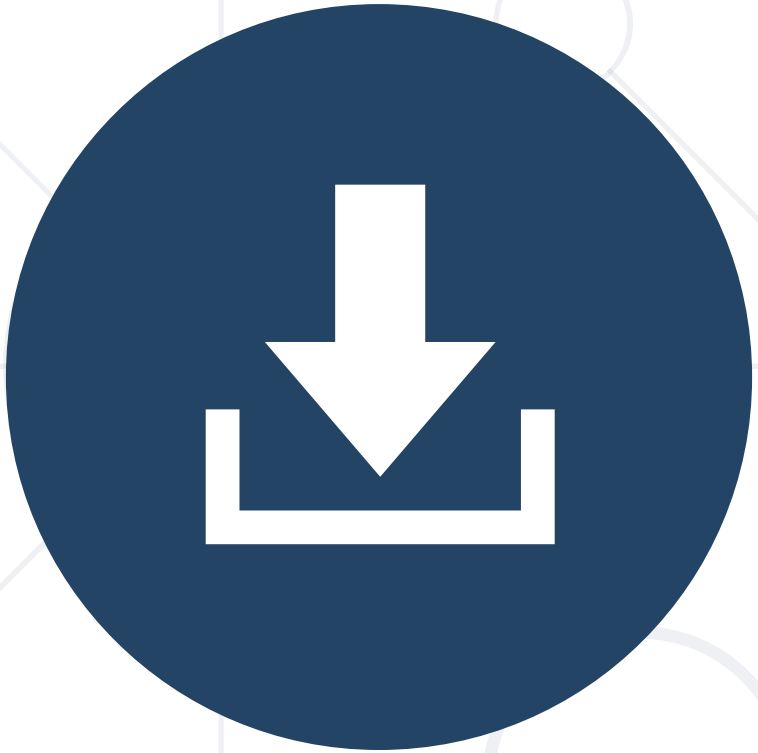
```
public class RegisterUserModel
{
    [Required]
    public string Username { get; set; }

    [Required]
    [StringLength(20)]
    public string Password { get; set; }

    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    [IsBefore("01/01/2000")]
    public DateTime BirthDate { get; set; }
}
```



Custom Model Validation (3)

- You can also use validation **directly** in the Binding Model
  - This is done by using the **IValidatableObject** interface

```
public class RegisterUserModel : IValidatableObject
{
    public string Username { get; set; }
    public string Password { get; set; }
    public string ConfirmPassword { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        if(string.IsNullOrEmpty(Username)) yield return new ValidationResult("Username cannot be empty");
        if(string.IsNullOrEmpty(Password)) yield return new ValidationResult("Password cannot be empty");
        if(ConfirmPassword != Password) yield return new ValidationResult("Passwords do not match");
    }
}
```



Uploading and Downloading Files

Files

Uploading Files (1)

- **ASP.NET Core MVC** supports **File Upload** using simple model binding
  - For larger files, **Streaming** is used

```
<form method="post" enctype="multipart/form-data"
                    asp-controller="Files" asp-action="Upload">
    <input type="file" name="file">
    <button type="submit" value="Upload" />
</form>
```

  - **Multiple-file** upload is also supported

```
<form method="post" enctype="multipart/form-data"
                    asp-controller="Files" asp-action="Upload">
    <input type="file" name="files" multiple >
    <button type="submit" value="Upload" />
</form>
```

Uploading Files (2)

- When uploading files using model binding, your action should accept
  - `IFormFile` (for single file) or `IEnumerable<IFormFile>` (or `List<IFormFile>`)

```
[HttpPost("Upload")]
public async Task<IActionResult> Upload(List<IFormFile> files)
{
    var filePath = Path.GetTempFileName(); // Full path to file in temp location

    foreach (var formFile in files.Where(f => f.Length > 0))
    {
        using (var stream = new FileStream(filePath, FileMode.Create))
        {
            await formFile.CopyToAsync(stream);
        }
    } // Copy files to FileSystem using Streams

    var bytes = files.Sum(f => f.Length);
    return Ok(new { count = files.Count, bytes, filePath});
}
```

Downloading Files (1)

- **ASP.NET Core** abstracts file system access through **File Providers**
  - File Providers are used throughout the ASP.NET Core framework
- Examples of where **ASP.NET Core** uses **File Providers** internally
  - `IHostingEnvironment` exposes the app's **content root** and **web root**
  - **Static File Middleware** uses **File Providers** to locate static files
  - **Razor** uses **File Providers** to locate pages and views

Downloading Files (2)

- To access physical files, you have to use **PhysicalFileProvider**
  - You'll have to initialize it with your server physical files folder path
  - Then you can extract information about the **File**

```
public IActionResult Download(string fileName)
{
    // Construct the path to the physical files folder
    string filePath = this.env.ContentRootPath + this.config["FileSystem:FilesFolderPath"];

    IFileProvider provider = new PhysicalFileProvider(filePath); // Initialize the Provider
    IFileInfo fileInfo = provider.GetFileInfo(fileName); // Extract the FileInfo

    var readStream = fileInfo.CreateReadStream(); // Extact the Stream
    var mimeType = "application/octet-stream"; // Set a mimeType

    return File(readStream, mimeType, fileName); // Return FileResult
} // NOTE: There is no check if the File exists. This action may result in an error
```

Razor Syntax

What is Razor?

Software University

- Simple-syntax **view engine**
- **Code-focused** templating approach
- Easy transition between HTML and code
- Combining **HTML** and **C#**

```
<div class="article">
    <div>@article.Title</div>
    <div>@article.Content</div>
</div>
```

```
<ul id="products">
    @foreach (var p in products)
    {
        <li>@p.Name ($@p.Price)</li>
    }
</ul>
```

Razor Syntax (1)

- @ – For values (HTML encoded)

```
<p>
    Current time is: @DateTime.Now
    Not HTML encoded value: @Html.Raw(someVar)
</p>
```

- @{…} – For code blocks (keep the view simple)

```
@{
    var productName = "Energy drink";
    if (Model != null) { productName = Model.ProductName; }
    else if (ViewBag.ProductName != null) { productName = ViewBag.ProductName; }
}
<p>Product "@productName" has been added in your shopping cart</p>
```

Razor Syntax (2)

- `If`, `else`, `for`, `foreach`, etc. C# statements
  - HTML markup lines can be included at any part
  - `@:` – For plain text line to be rendered

```
<div class="products-list">
@if (Model.Products.Count() == 0) { <p>Sorry, no products found!</p> }
else
{
      @:List of the products found:
      foreach(var product in Model.Products)
      {
            <b>@product.Name, </b>
      }
}
</div>
```

Razor Syntax (3)

- Comments

```
@*
    A Razor Comment
*@
@{
    // A C# comment
    /* A Multi
       line C# comment
    */
}
<!-- HTML Comment -->
```

- Escaping @

```
<p>
    This is the sign that separates email names from domains: @@<br />
    And this is how smart Razor is: spam_me@gmail.com
</p>
```

Razor Syntax (4)

- `@(…)` – Explicit code expression

```
<p>
    Current rating(0-10): @Model.Rating / 10.0  @* 6 / 10.0 *@
    Current rating(0-1): @(Model.Rating / 10.0) @* 0.6 *@
    spam_me@Model.Rating                        @* spam_me@Model.Rating *@
    spam_me@(Model.Rating)                      @* spam_me6 *@
</p>
```

- `@using` – for including namespace into view
- `@model` – for defining the model for the view

```
@using MyWebApp.Models;
@model UserModel
<p>@Model.Username</p>
```

Views – Dependency Injection

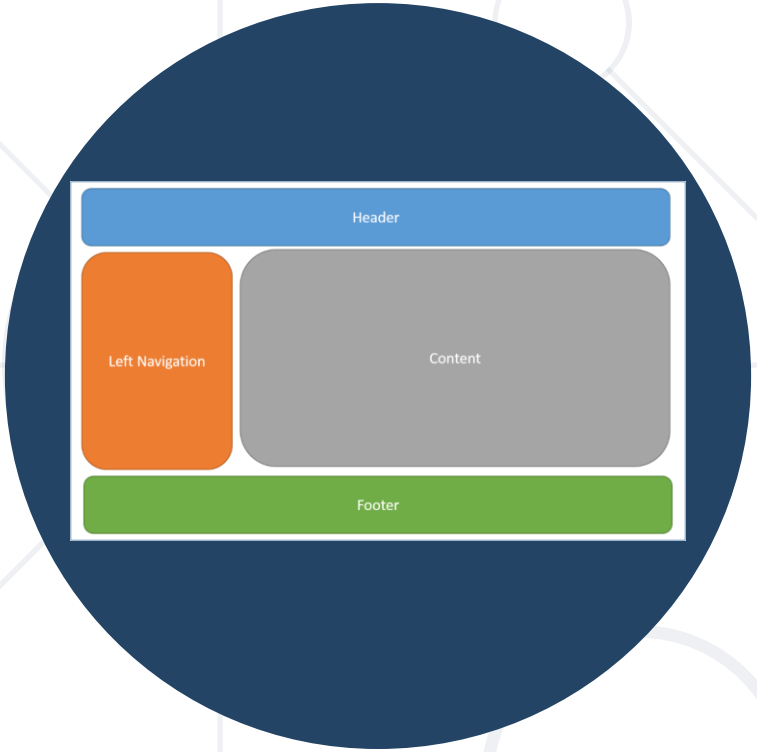
- **ASP.NET Core** supports **dependency injection** into views
  - You can inject a **Service** into a **View** by using **`@inject`**

```
public class DataService
{
    1 reference
    public IEnumerable<string> GetData()
    {
        return new[] { "David", "John", "Max", "George" };
    }
}
```

```
builder.Services.AddScoped<DataService, DataService>();
```

```
@using Demo.Services
@inject DataService DataService

<div class="list">
    @foreach(var item in DataService.GetData)
    {
        <h1>@item</h1>
    }
</div>
```

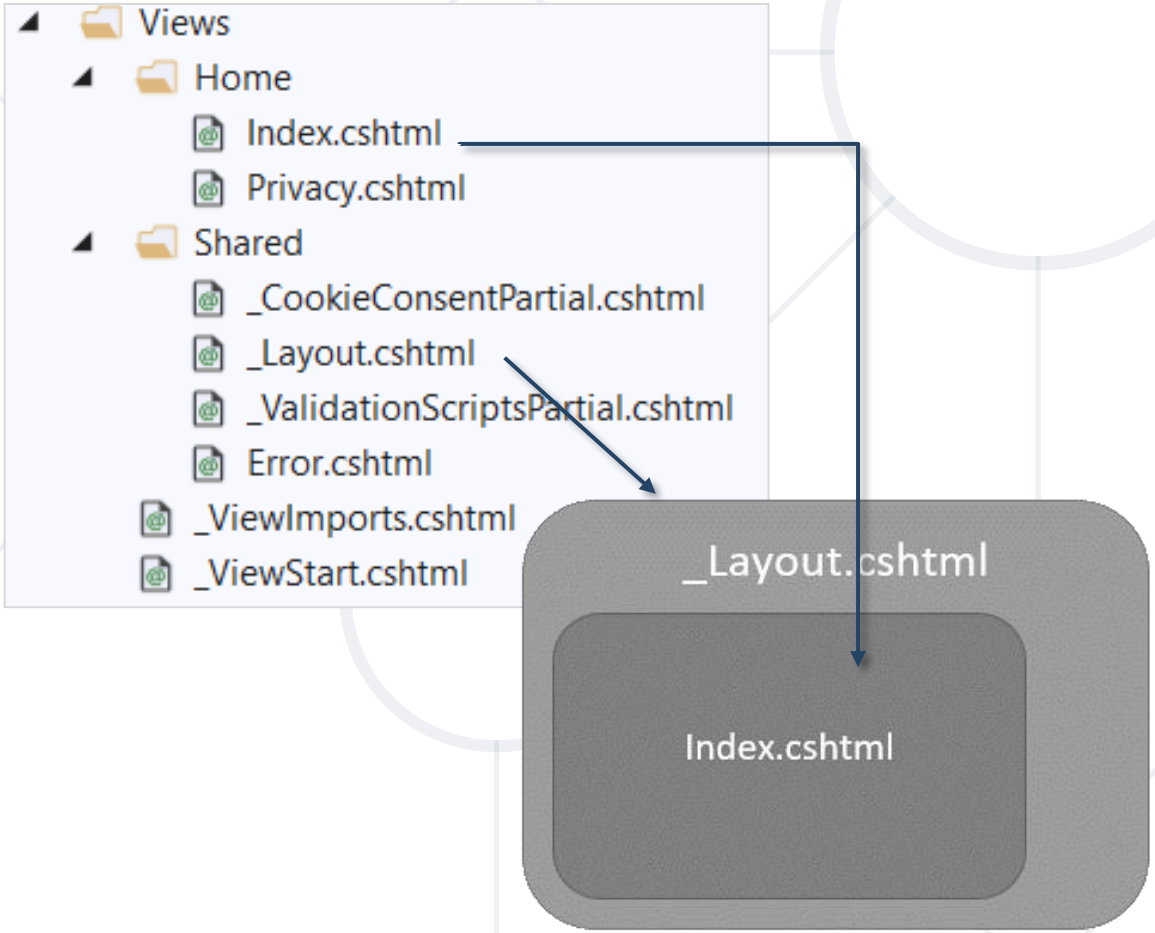
David
John
Max
George

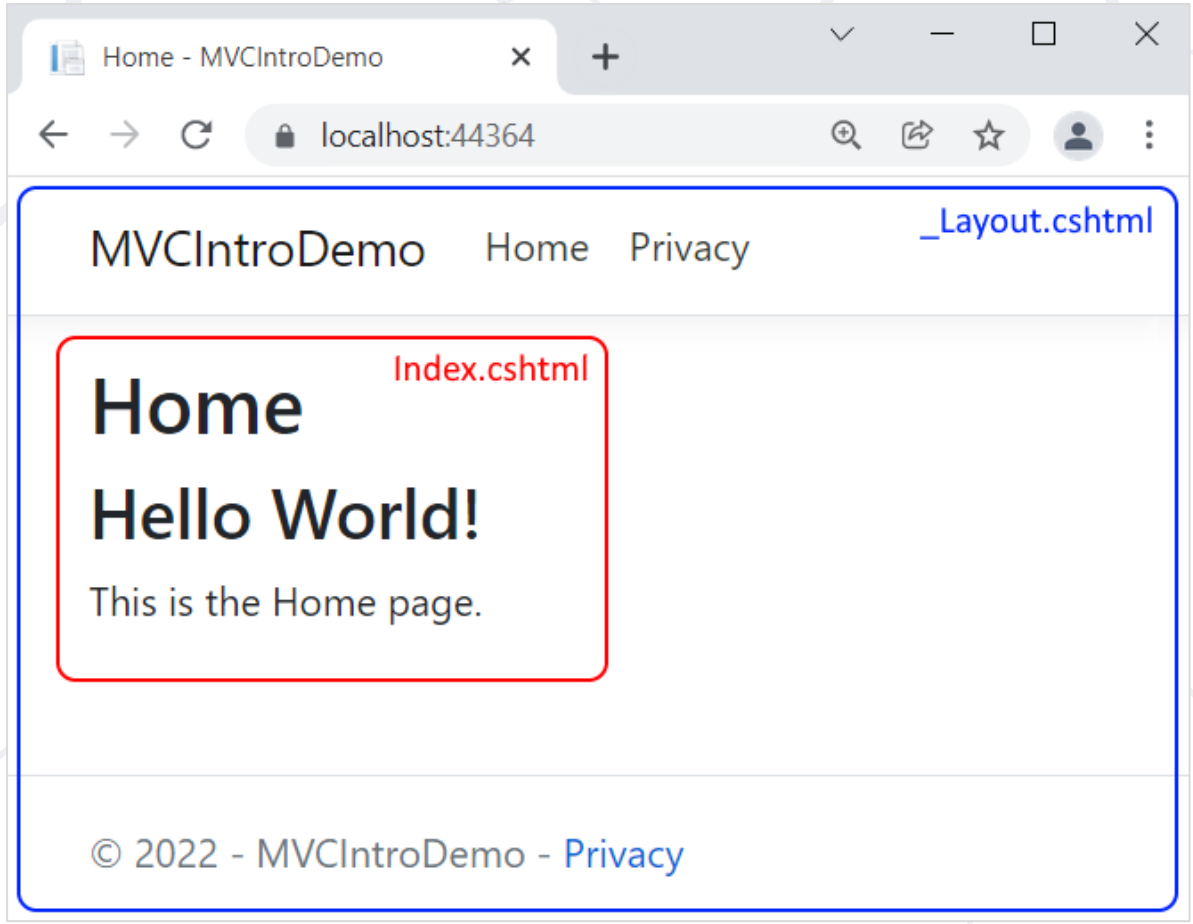


Layout and Special View Files

_Layout.cshtml (1)

- Defines a **common site template** (~/Views/Shared/_Layout.cshtml)





_Layout.cshtml (2)

- Razor View engine renders content **inside-out**
  - First the **View** is rendered, and after that – the **Layout**
- **@RenderBody()** – indicate where we want the views based on this layout to "**fill in**" their core content at that location in the HTML

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <nav>@* Menu *@</nav>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Sections

- You can have one or more "**sections**" (optional), defined in views

Index.cshtml

```
@section SideBar {
    <aside> Some side info</aside>
}
```

- Can be rendered anywhere in the layout page using the method **RenderSection()**
  - **@RenderSection(string name, bool required)**
  - If the section is required and not defined, an exception will be thrown (**IsSectionDefined()**)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/site.css")
        rel="stylesheet" type="text/css" />
</head>
<body>
    <div id="header">
        <h1>My Site Header</h1>
    </div>

    <div id="sidebar">
        @RenderSection("SideBar", required: false);
    </div>

    <div id="content">
        @RenderBody();
    </div>

    <div id="footer">
        <h1>Site Footer - &copy; </h1>
    </div>
</body>
</html>
```

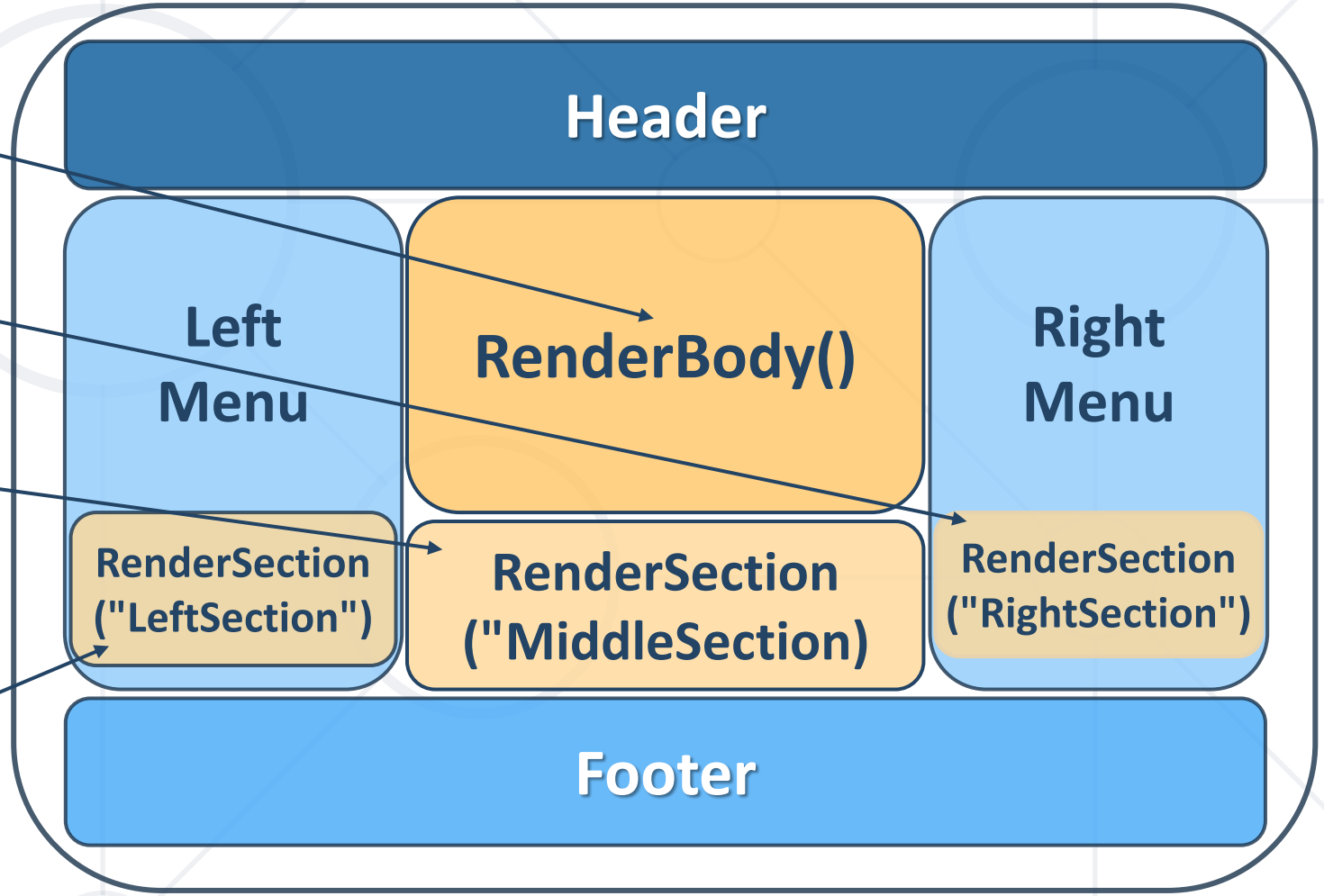
Scheme

Software University

Index.cshtml

```
<div>
    This is the main content
</div>

@section RightSection {
    <text>
        This is the right section
    </text>
}

@section MiddleSection {
    <text>
        This is the middle section
    </text>
}

@section LeftSection {
    <text>
        This is the middle section
    </text>
}
```

_Layout.cshtml



_ViewStart.cshtml

Software University

- Views don't need to specify layout since their default layout is set in their `_ViewStart` file
  - `~/Views/_ViewStart.cshtml` (code for all views)
- Each view can specify custom layout pages

```
@{
    Layout = "~/Views/Shared/_UncommonLayout.cshtml";
}
```

- Views without layout

```
@{
    Layout = null;
}
```

_ViewImports.cshtml

Software University

- If a directive or a dependency is shared between many Views, it can be specified globally in the **ViewImports**
  - **~/Views/_ViewImports.cshtml** (code for all views)

```
@using MyWebApp
@using MyWebApp.Models
@using MyWebApp.Models.AccountViewModels
@using MyWebApp.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

- This file does not support other Razor features

_ValidationScriptsPartial.cshtml

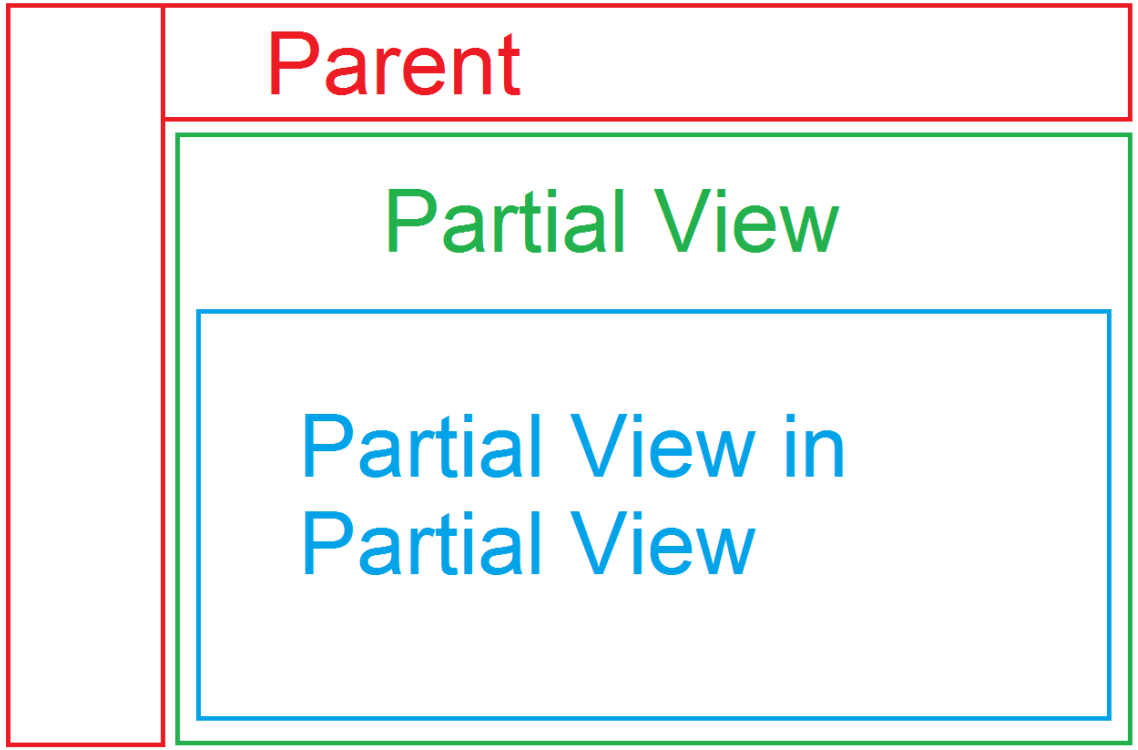
Software University

- This file contains **validation scripts** in the form of a partial view
  - **~/Views/Shared/_ValidationScriptsPartial.cshtml**

```
<script
   src="~/lib/jquery-validation/dist/jquery.validate.min.js">
</script>
<script
   src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js">
</script>
```

- To use them, render the **partial view** inside a **view** in a **section**

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



Partial Views and View Components

Partial Views

- **Partial Views** render portions of a page
  - Break up large markup files into smaller components
  - Reduce the duplication of common view code
- Razor partial views are normal views (`.cshtml` files)
  - Usually placed in `/Shared/` or in the same directory where used

Use of Partial Views

- HTML Helper for Partial Views

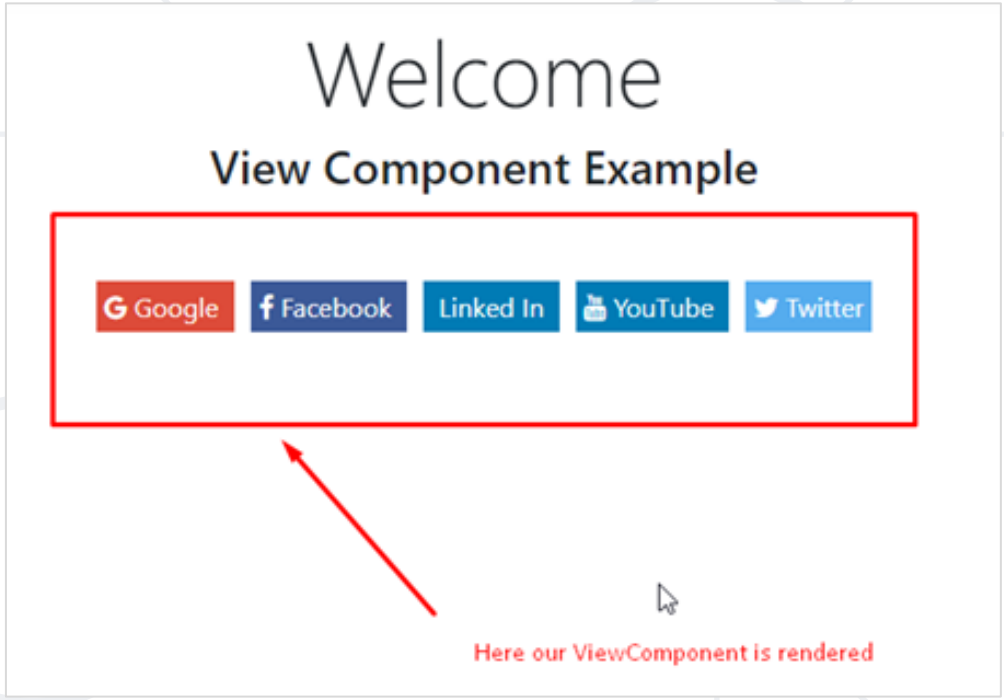
```
@using WebApplication.Models;
@model ProductsListViewModel

@foreach (var product in Model.Products)
{
    @await Html.PartialAsync("_ProductPartial", product);
}
```

- Tag Helper for Partial Views

```
@foreach (var product in Model.Products)
{
    <partial name="_ProductPartial" model="product" />
}
```

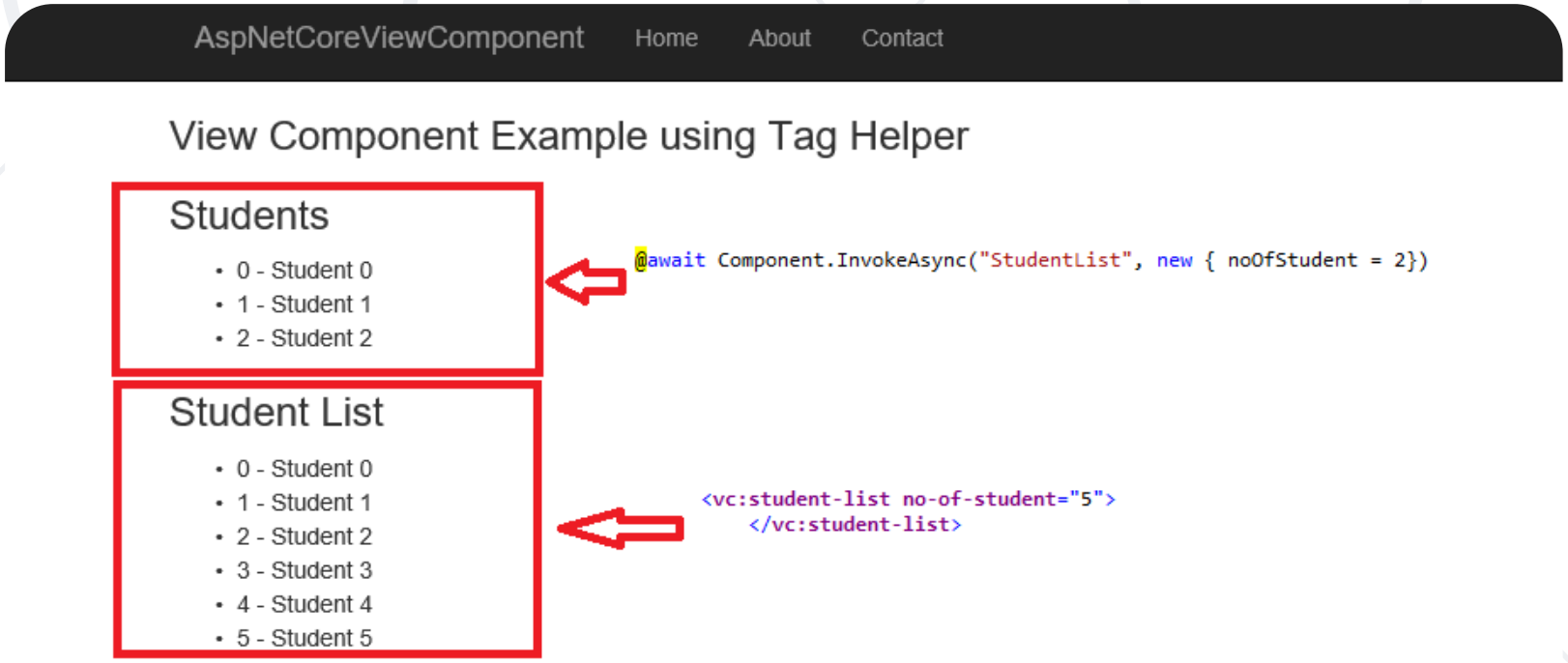
View Components (1)

- **View Components** are similar to **Partial Views**, but much more powerful
  - No model binding
  - Depend only on the data provided to it
- **View Components**
  - Render a **chunk** rather than a whole response
  - Can have parameters and business logic
  - Typically invoked from a **Layout** page
  - Include the same **separation of concerns** and testability benefits between controller / view



View Components (2)

- **View components** are intended anywhere you have reusable rendering logic that's too complex for a partial view
  - Dynamic navigation menus
  - Login panels
  - Shopping carts
  - Sidebar content
  - Recently published articles
  - Tag cloud



View Components (3)

- **View Components** consist of 2 parts
  - A **class** – typically derived from **`ViewComponent`**
  - A **result** – typically a **`View`**
- **View Components**
  - Define their logic in a method called **`InvokeAsync()`**
  - Never directly handle a **`Request`**
  - Typically initialize a **`Model`** which is passed to the **`View`**

Defining Your Own ViewComponent (1)

Software University

\ViewComponents\HelloWorldViewComponent.cs

```
public class HelloWorldViewComponent : ViewComponent
{
    private readonly DataService _dataService;
    public HelloWorldViewComponent(DataService dataService)
       => _dataService = dataService;

    public async Task<IViewComponentResult> InvokeAsync(string name)
    {
        string helloMessage =
            await _dataService.GetHelloAsync();

        ViewData["Message"] = helloMessage;
        ViewData["Name"] = name;

        return View();
    }
}
```

Inherit the ViewComponent class

Components don't handle requests directly

Async method with logic

They often initialize a model which is passed to the view

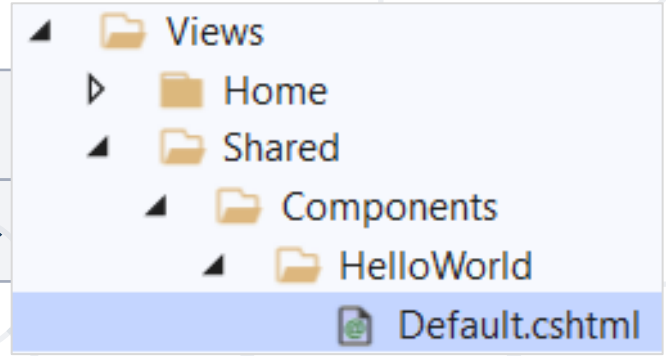
Typically return a view

Defining Your Own ViewComponent (2)

Software University

\Views\Shared\Components\HelloWorld\Default.cshtml

```
<h1>@ViewData["Message"]!!! I am @ViewData["Name"]</h1>
```



\Views\Home\Index.cshtml

```
...
<div class="view-component-content">
    @await Component.InvokeAsync("HelloWorld", new { name = "David" });
    <vc:HelloWorld name="John"></vc:HelloWorld>
</div>
```

To use a Tag Helper, register the assembly of the view component using the @addTagHelper directive

HTML Helpers and Tag Helpers

HTML Helpers

- Each view inherits **RazorPage**
  - **RazorPage** has a property named **Html**
- The **Html** Property has methods that return string can be used to
  - Create inputs
  - Create links
  - Create forms
- **Avoid** using HTML Helpers
  - Use Tag Helpers instead

```
@using (Html.BeginForm("Search", "Users",
                       FormMethod.Post))
{
    @Html.TextBox("username")
    <input type="submit" />
}
@Html.Raw(htmlContent)
```

HTML Helpers	
@Html.ActionLink	@Html.TextBox
@Html.BeginForm	@Html.TextArea
@Html.CheckBox	@Html.Password
@Html.Display	@Html.Hidden
@Html.Editor	@Html.Label
@Html.DropDownList	@Html.Action

Tag Helpers

- **Tag Helpers** enable the participation of Server-side code in the HTML element creation and rendering, in **Razor views**
  - There are built-in **Tag Helpers** for many common tasks
    - Forms, Links, Assets, etc.
  - There are **custom** Tag Helpers in **GitHub** repos and **NuGet**

Often start with asp-

```
<div class="form-group">
    <label asp-for="Password" class="col-md-2">Password</label>
    <div class="col-md-10">
        <input asp-for="Password" class="form-control" />
        <span asp-validation-for="Password" class="text-danger"></span>
    </div>
</div>
```

Tag Helpers vs HTML Helpers

- **Tag Helpers** attach to HTML elements in Razor Views
- **Tag Helpers** reduce the explicit transitions between **HTML** & **C#**
- **Tag Helpers** make the Razor markup quite **clean** and the views – quite **simple**

```
<label asp-for="firstName">First Name: </label>
```

- **HTML Helpers** are invoked as methods which generate content
- **HTML Helpers** tend to include a lot of C# code in the markup
- **HTML Helpers** use complex and very **C#-specific** Razor syntax in some cases

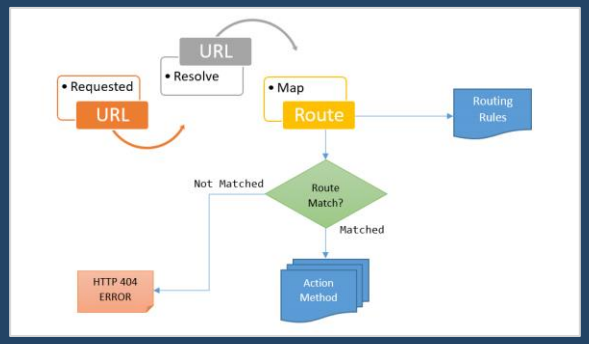
```
@Html.Label("firstName", "FirstName: ");
```

Creating Your Own Tag Helper

Software University

```
[HtmlTargetElement("h1")]
public class HelloTagHelper : TagHelper
{
    private const string MessageFormat = "Hello, {0}";
    public string TargetName { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        string formattedMessage = string.Format(MessageFormat, this.TargetName);
        output.Content.SetContent(formattedMessage);
    }
}
```

```
@using WebApplication;
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelper
@addTagHelper WebApplication.TagHelpersHelloTagHelper, WebApplication

<div class="tag-helper-content">
    <h1 target-name="John"></h1>
</div>
```


URL
• Resolve
• Requested
URL
• Map
Route
Routing Rules
Not Matched
Route Match?
Matched
HTTP 404 ERROR
Action Method

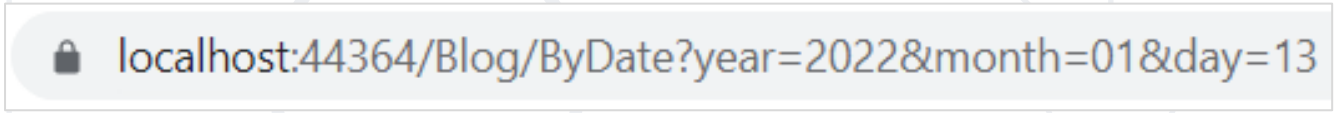
Routing

Route Constraints

- **Route Constraints** are rules on the URL segments

```
endpoints.MapControllerRoute(
  name: "blog",
  pattern: "{year}/{month}/{day}",
  defaults: new { controller = "Blog", action = "ByDate" },
  constraints: new { year = @"\d{4}", month = @"\d{1,2}", day = @"\d{1,2}" }
);
```

- All the constraints are **regular expression compatible** with the **Regex** class




```
class BlogController : Controller {
  public IActionResult ByDate(
    string year, string month, string day)
    { … }
}
```

Attribute Routing (1)

- It uses a set of attributes to map **actions** directly to **route template**
- It can also directly define the **request method**
- **Http{Action}** attributes are quite often used in **REST APIs**

```
public class HomeController : Controller
{
    [Route("/")]
    public IActionResult Index() => View();
}
```

```
public class HomeController : Controller
{
    [HttpGet("/")]
    public IActionResult Index() => View();
}
```

```
public class UsersController : Controller
{
    [HttpPost("Login")]
    public IActionResult Login() => View();
}
```

Attribute Routing (2)

- **Attribute routing** allows you to create multiple routes for a single action
- It also allows you to **combine** a route for a **controller** and an **action route**

```
public class HomeController :
Controller
{
    // …

    [Route("/")]
    [Route("Index")]
    public IActionResult Index()
    {
        return View();
    }
}
```

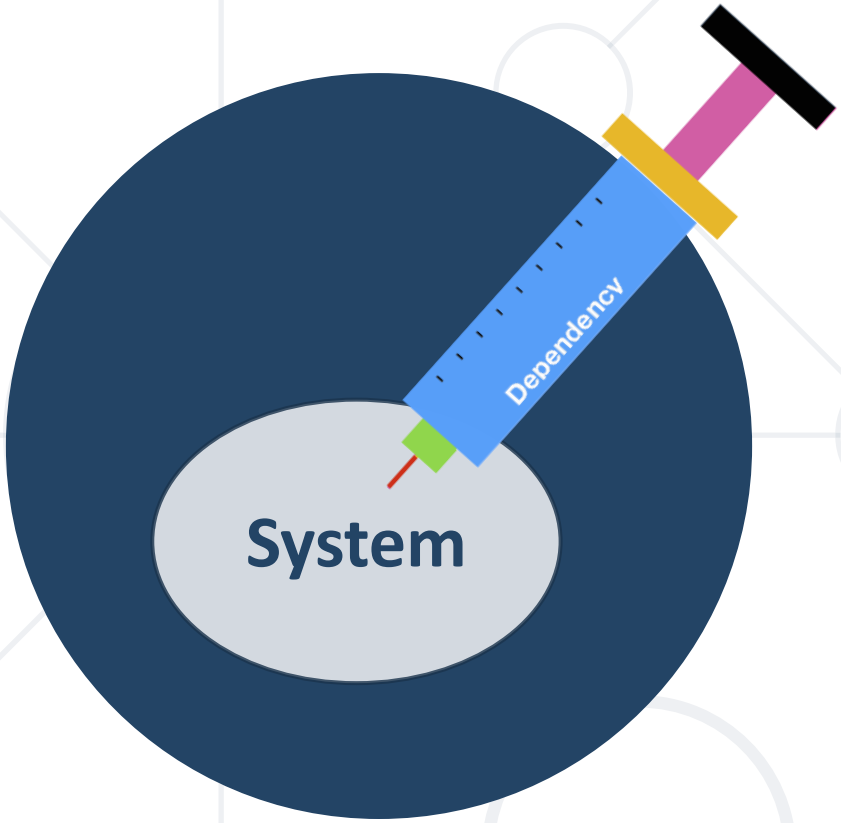
```
[Route("Home")]
public class HomeController : Controller
{
    // …

    [Route("/")] // Does not combine, Route – /
    [Route("Index")] // Route - /Home/Index
    [Route("")] // Route - /Home
    public IActionResult Index()
    {
        return View();
    }
}
```

Static Files Routing

- Can be modified to serve other folders

```
app.UseStaticFiles(
new StaticFileOptions()
{
      FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "OtherFiles")),
      RequestPath = new PathString("/files")
});
```

This will serve "style.css" file upon request "http://{app}/files/style.css" from "OtherFiles" instead of "wwwroot"



Dependency Injection

Design Pattern for IoC Implementation

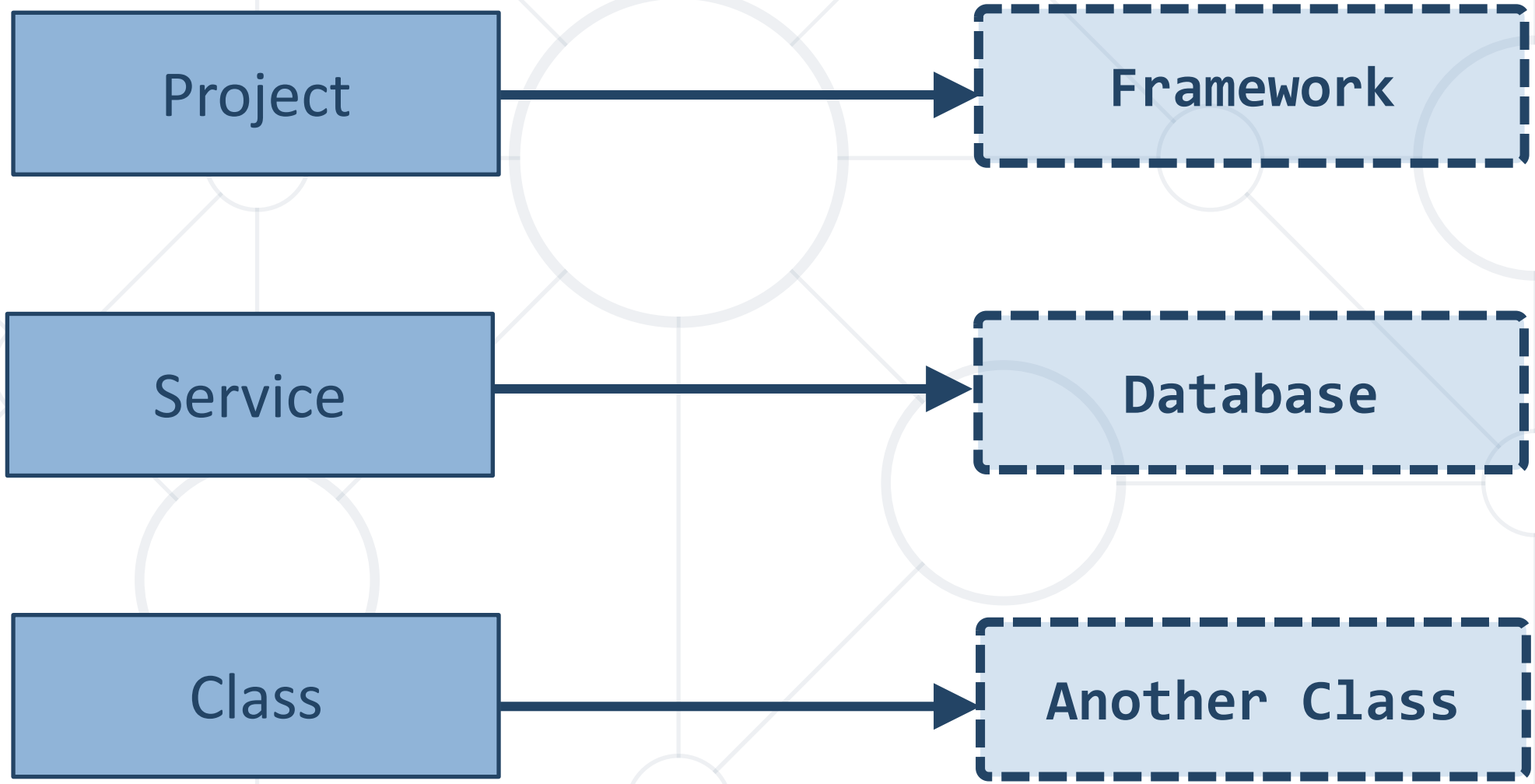
What is a Dependency?

- Another **object** that your class **needs**
  - Other examples (**Framework**, **Database**, **File System**, **Providers**)
- Classes **dependent** on each other are called **coupled**
- Dependencies are **bad** because they **decrease** reuse

```
public class Customer
{
    var customerService =
            new CustomerService('Service');
}
```

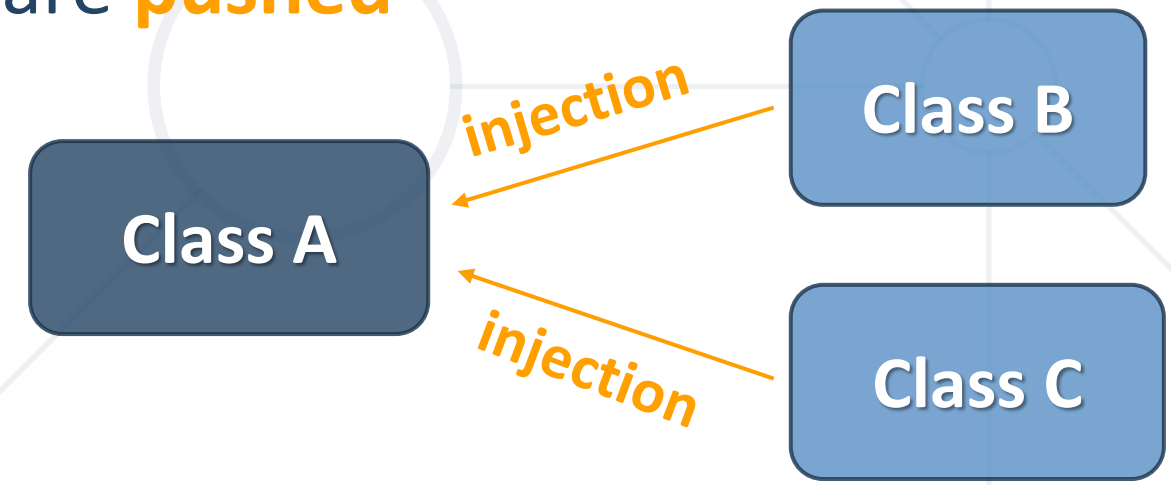
Customer class is **dependent** on **concrete** service

Dependency Examples

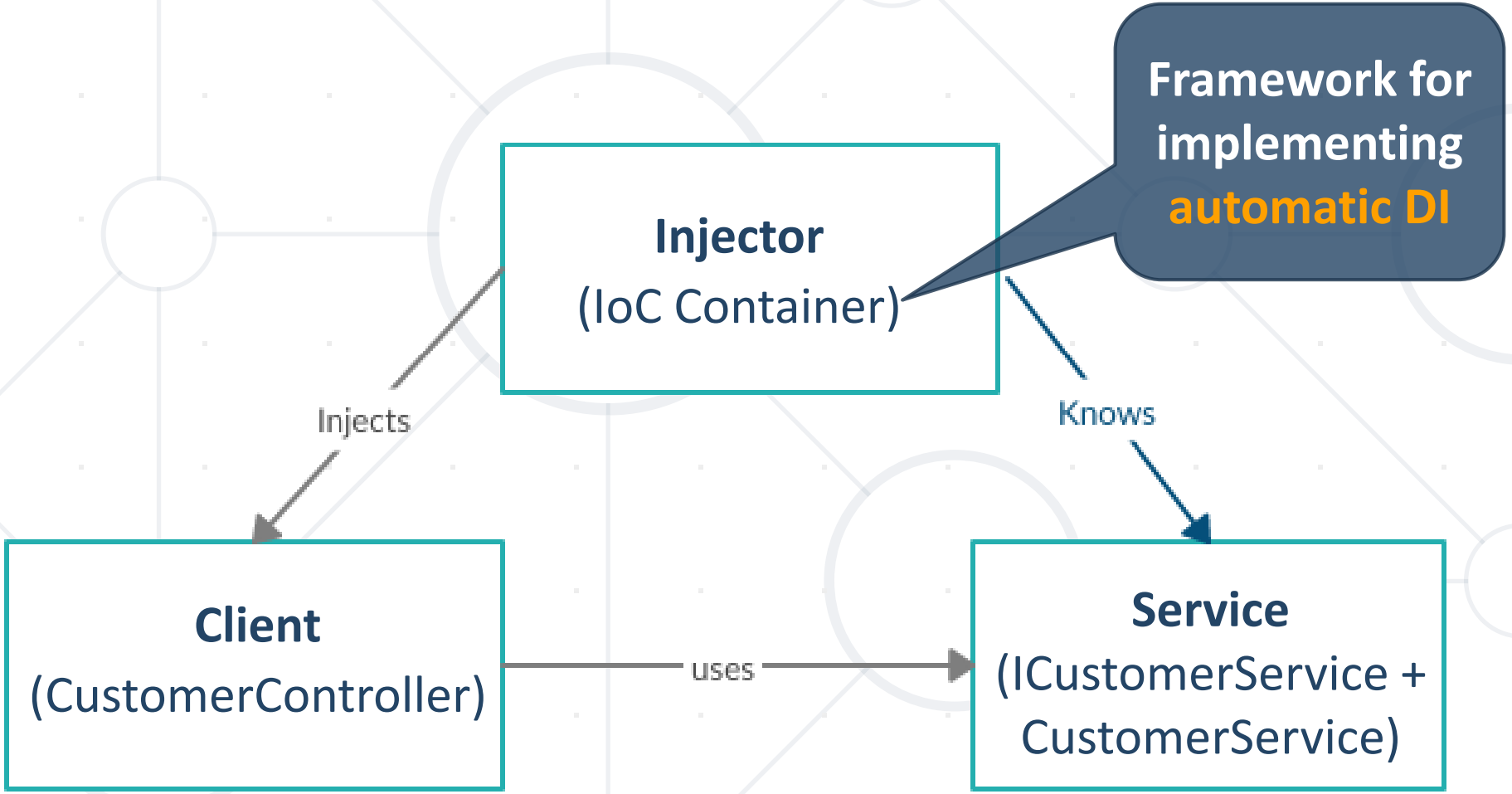


What is Dependency Injection?

- **Dependency Injection** (DI) is a popular **design pattern**
- It is a technique for achieving **Inversion of Control** (IoC)
  - Classes should declare what they need
  - Constructors should inject dependencies (**constructor injection**)
  - Dependencies (abstractions) are **pushed** in the class from the **outside**
  - Classes do **not** instantiate their dependencies



Dependency Injection Scheme

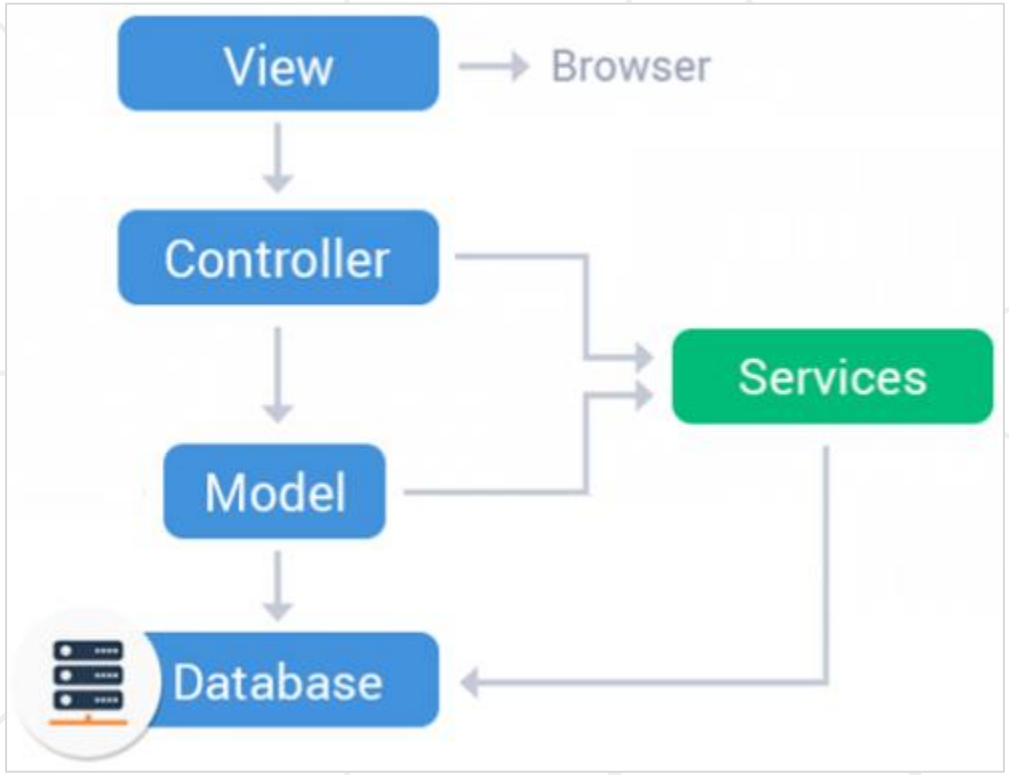


Constructor Injection

- Decouples dependencies
- **Pros**
  - Classes **self document** requirements
  - Works well **without** container
  - Always **valid** state
- **Cons**
  - Many **parameters**
  - Some methods may **not** need **everything**

```
public class Customer
{
    private ICustomerService _customerService;

    public Customer(ICustomerService service)
    {
        _customerService = service;
    }
}
```

The service **comes** from **outside**

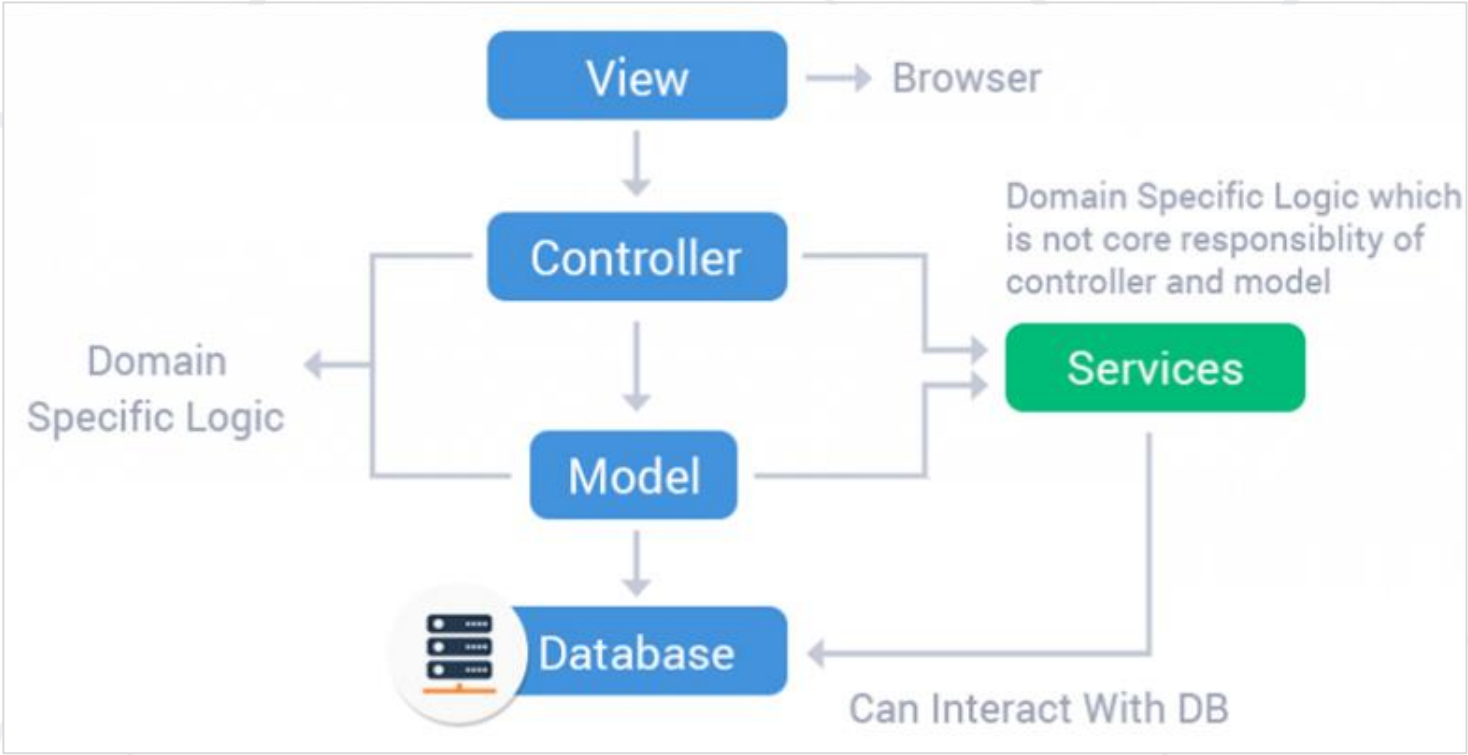


Services

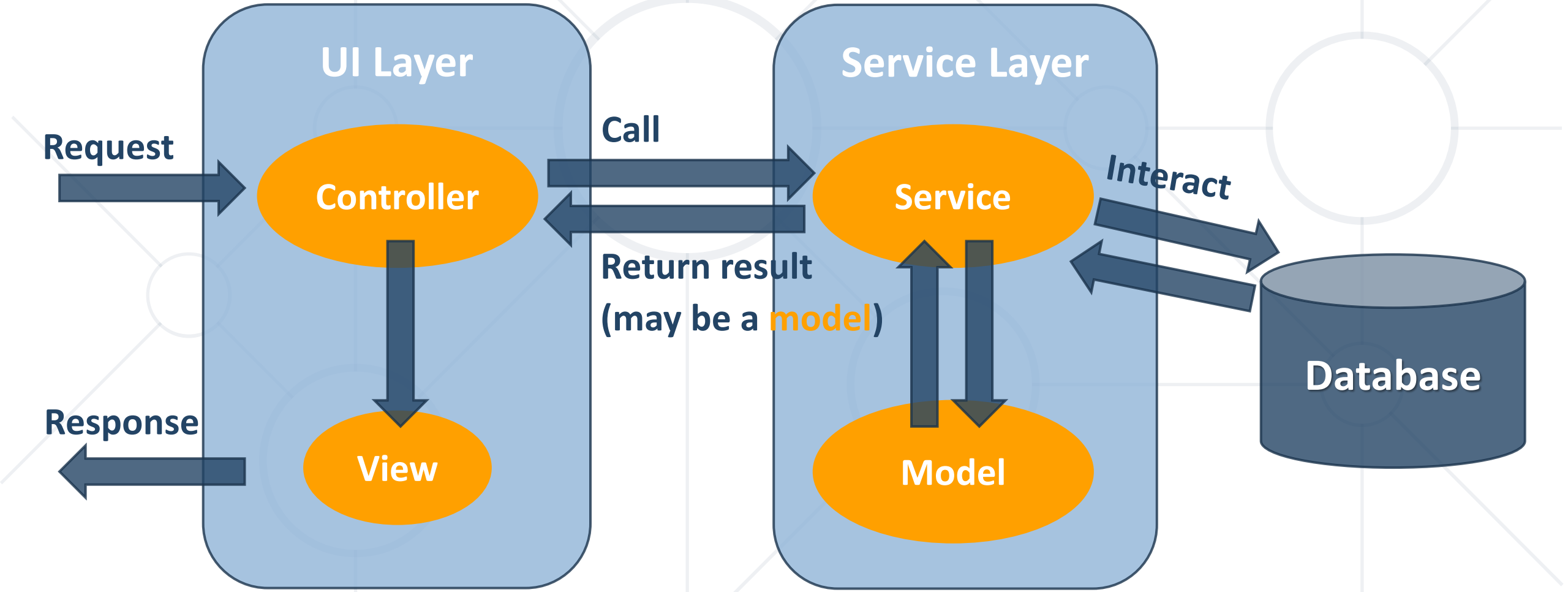
Service Layer in MVC

Service Layer

- **Service layer** is an additional layer in an ASP.NET MVC app between controllers and database layer
- Resolves the problem with duplicating code in controller actions
- It contains **business logic**
  - Controller actions should **not contain database logic**
  - Controllers may get a **model** from the **service layer** and pass it to a **view**



MVC with Services



Application Services Configuration

- **Configuration options**, by convention, are set in **Program.cs**
- Services can be configured for **Dependency Injection** differently

```
builder.Services.AddTransient<DataService>();

builder.Services.AddScoped(typeof(DataService));

builder.Services.AddSingleton<DataService>();
```

Transient objects are always different. A new instance is provided to every controller and service

Singleton objects are the same for every object and request

Scoped objects are the same within a request. They are different across different requests

Service Interface + Configuration

- Services are typically defined using **interfaces**
  - Interfaces **define** service methods

```
public interface IProductService
{
    List<ProductServiceModel> All();

    void CreateProduct(string name, string description);
}
```

- Configure the **service** in the **Program.cs** class

```
builder
       .Services
       .AddTransient<IProductService, ProductService>();
```

Allows you to **inject services** into **controller classes constructors** via **DI**

Service

- Should contain the **business logic**
- May interact with the **database context**

```
public class ProductService : IProductService
{
    private readonly ApplicationDbContext _data;

    public ProductService(ApplicationDbContext data)
        => _data = data;

    public void CreateProduct(string name, string description)
    {
        var product = new Product()
              { Name = name, Description = description};

        _data.Products.Add(product);
        _data.SaveChanges();
    }
}
```

Accept the db context through the constructor

Method contains business logic for creating a product

Controller

- Controllers should be responsible only for the request and response

```
public class ProductsController : Controller
{
    private IProductService _productService;

    public ProductsController(IProductService service)
        => _productService = service;

    public IActionResult Create() => View();

    [HttpPost]
    public IActionResult Create(ProductFormModel model)
    {
        if (!ModelState.IsValid)
        {
            return View(model);
        }
        _productService.CreateProduct(model.Name, model.Description);

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

Inject the service through the constructor

Invoke service methods for the business logic

Service with Service Model

Special model for the service

```
public class ProductServiceModel
{
    0 references
    public int Id { get; set; }

    0 references
    public string Name { get; set; }
}
```

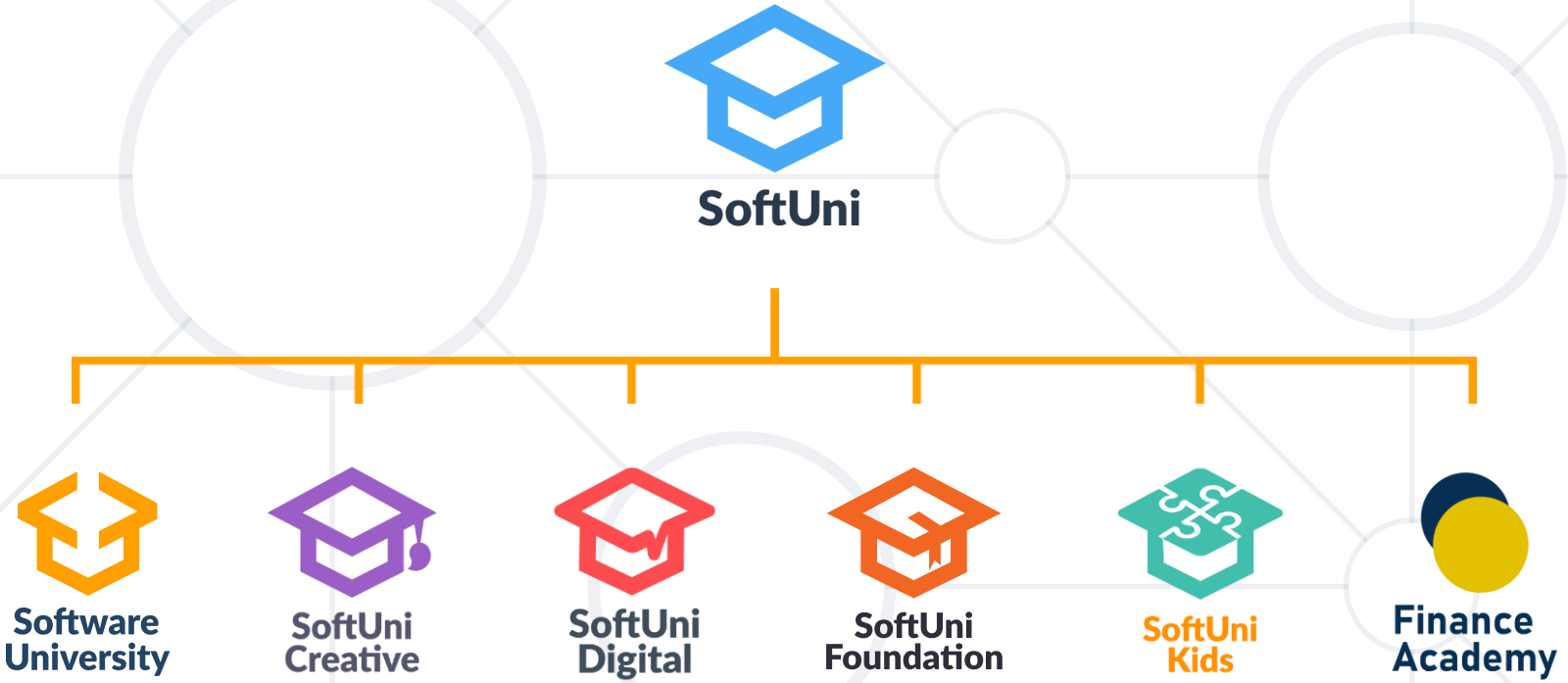
```
public class ProductController : Controller
{
    private IProductService _productService;

    0 references
    public ProductController(IProductService service)
     => _productService = service;

    0 references
    public IActionResult All()
    {
        var model = _productService.All();
        return View(model);
    }
}
```

```
public class ProductService : IProductService
{
    private readonly ApplicationDbContext _data;

    0 references
    public ProductService(ApplicationDbContext data)
     => _data = data;

    0 references
    public List<ProductServiceModel> All()
    {
        var products = _data.Products
            .Select (p => new ProductServiceModel
            {
                Id = p.Id,
                Name = p.Name,
            })
            .ToList();
        return products;
    }
}
```

Summary

- Model **Binding**
- Model **Validation**
- Working with **Files**
- **Razor** syntax
- **Layout** and **Special** View Files
- **Partial** Views and View Components
- **HTML Helpers** and **Tag Helpers**
- **Routing**
- **Dependency Injection** and **Services**

Questions?



SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

createX

Postbank
Решения за твоето утре

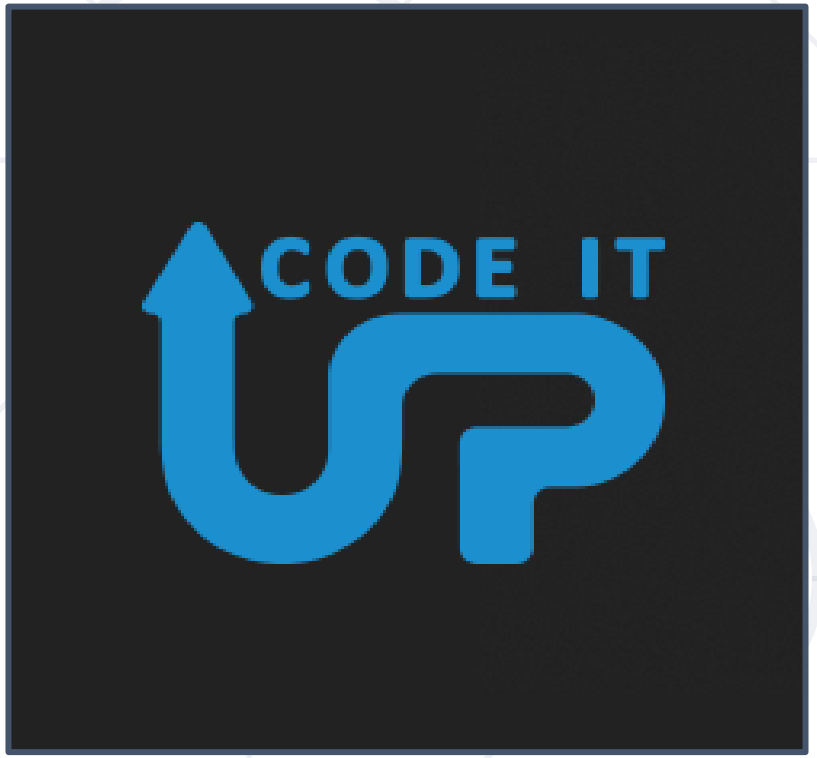
BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

Software University



Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg