

# ASP.NET Core Introduction

ASP.NET Core, MVC, Controllers & Actions,  
Creating an ASP.NET Core MVC app



SoftUni Team  
Technical Trainers



**SoftUni**

Software University

<https://softuni.bg>

sli.do

**#csharp-web**

# Table of Content

1. ASP.NET Core
2. MVC
3. ASP.NET Core MVC
4. Creating an ASP.NET Core MVC App
5. Controllers and Actions
6. MVC Views and Razor
7. Routing
8. Static Files
9. Dependency Injection
10. Model Binding
11. Model Validation

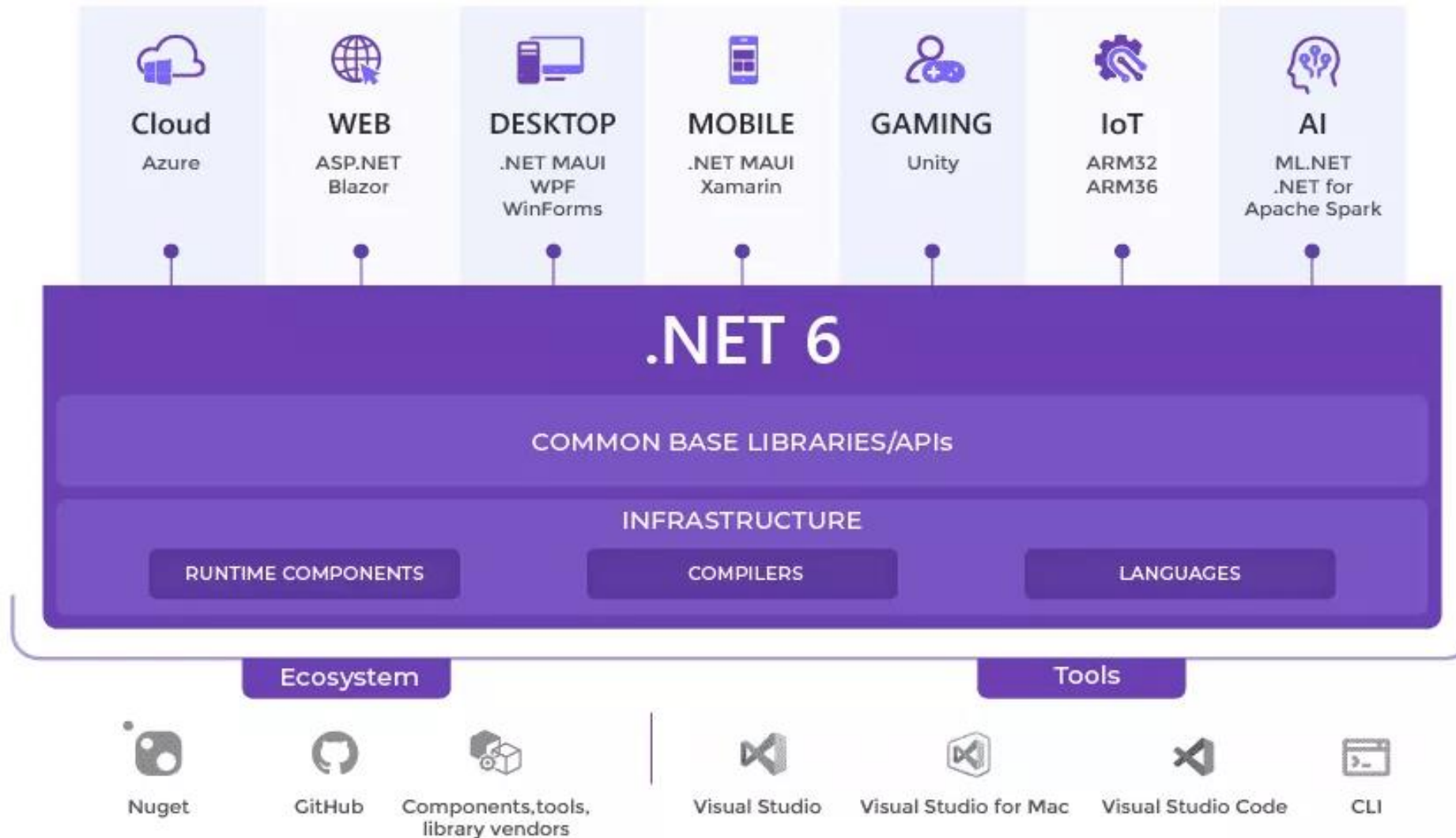




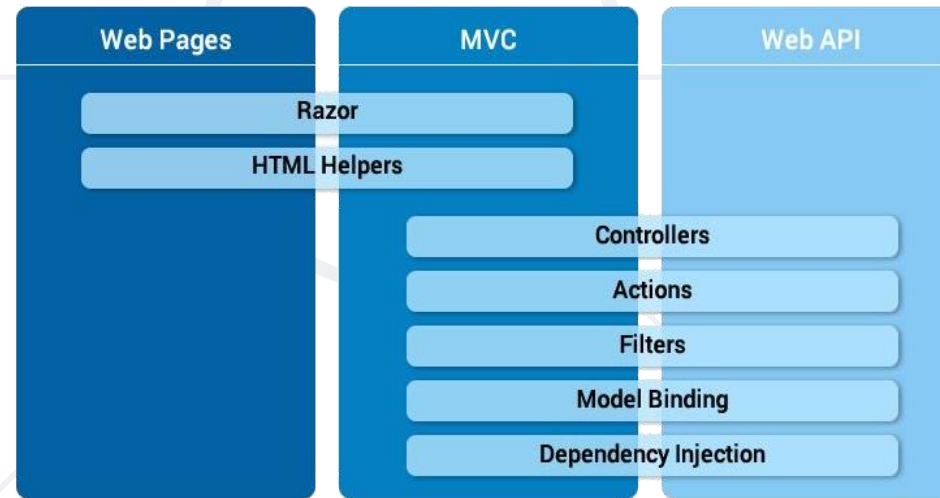
# ASP.NET Core

## Overview

# .NET Core: Bird's Eye View



- **ASP.NET Core** is a cross-platform open-source back-end development framework for C#



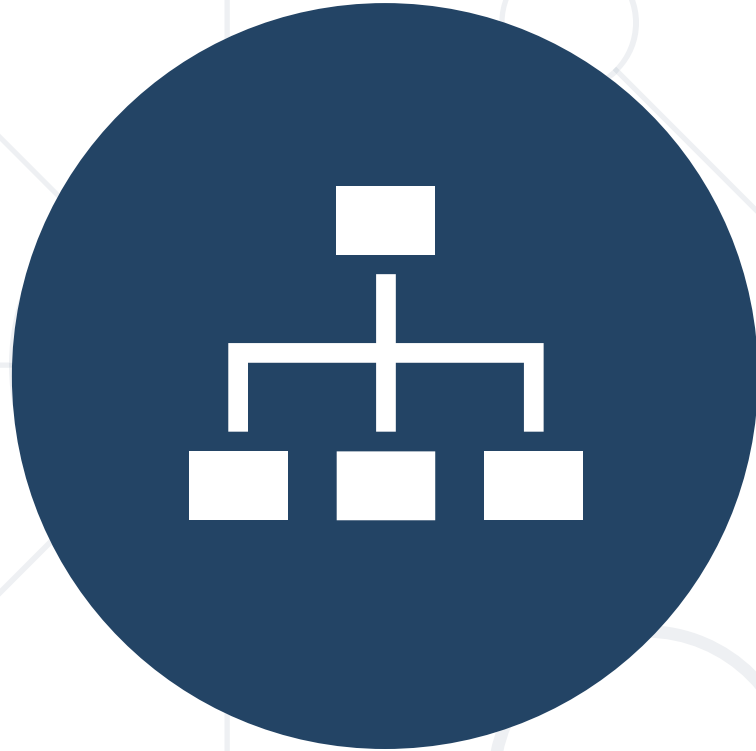
- ASP.NET Core **Web Pages**: build simple Web apps
- ASP.NET Core **MVC**: build server-side Web apps
- ASP.NET Core **Web API**: build Web services and REST APIs

- Great documentation: <https://docs.microsoft.com/en-us/aspnet>
- **ASP.NET Core** provides
  - Integration of modern client-side frameworks (Angular, React, Blazor, etc.)
  - Development workflows (MVC, WebAPI, Razor Pages, SignalR)
- **ASP.NET Core** applications run both on **.NET Core** and **.NET Framework**

# ASP.NET Core Main Features

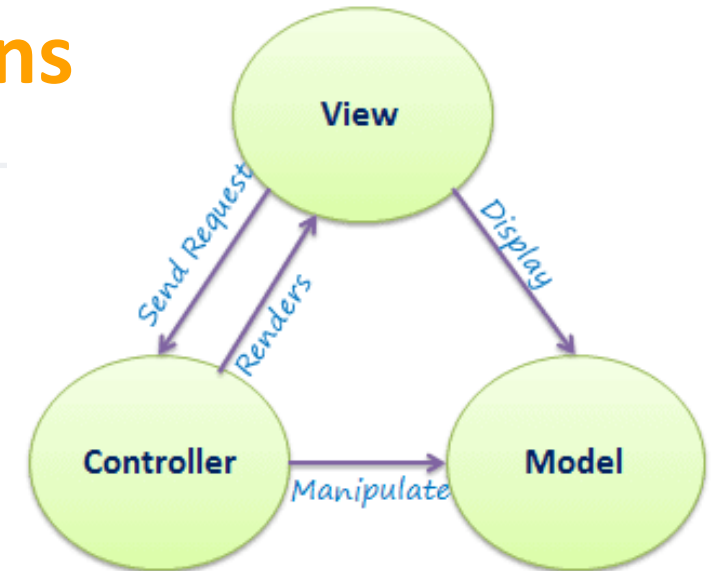
- A **unified framework** for building web UI and web APIs, architected for testability
- Ability to develop and run on **Windows, macOS and Linux**
  - Ability to host on IIS, Nginx, Apache, Docker or self-host in your own process
- Built-in **dependency injection**
- A lightweight, high-performance and modular HTTP request pipeline (**middlewares**)
- **Razor Pages** is a page-based programming model that makes building web UI easier
- **Blazor** lets you use C# in the browser and share server-side and client-side app logic
- **Razor markup** provides syntax for Razor Pages, **MVC** views and Tag Helpers
- Cloud-ready, environment-based configuration system
- Side-by-side app versioning
- Tooling that simplifies modern web development (Visual Studio, VS Code, CLI)





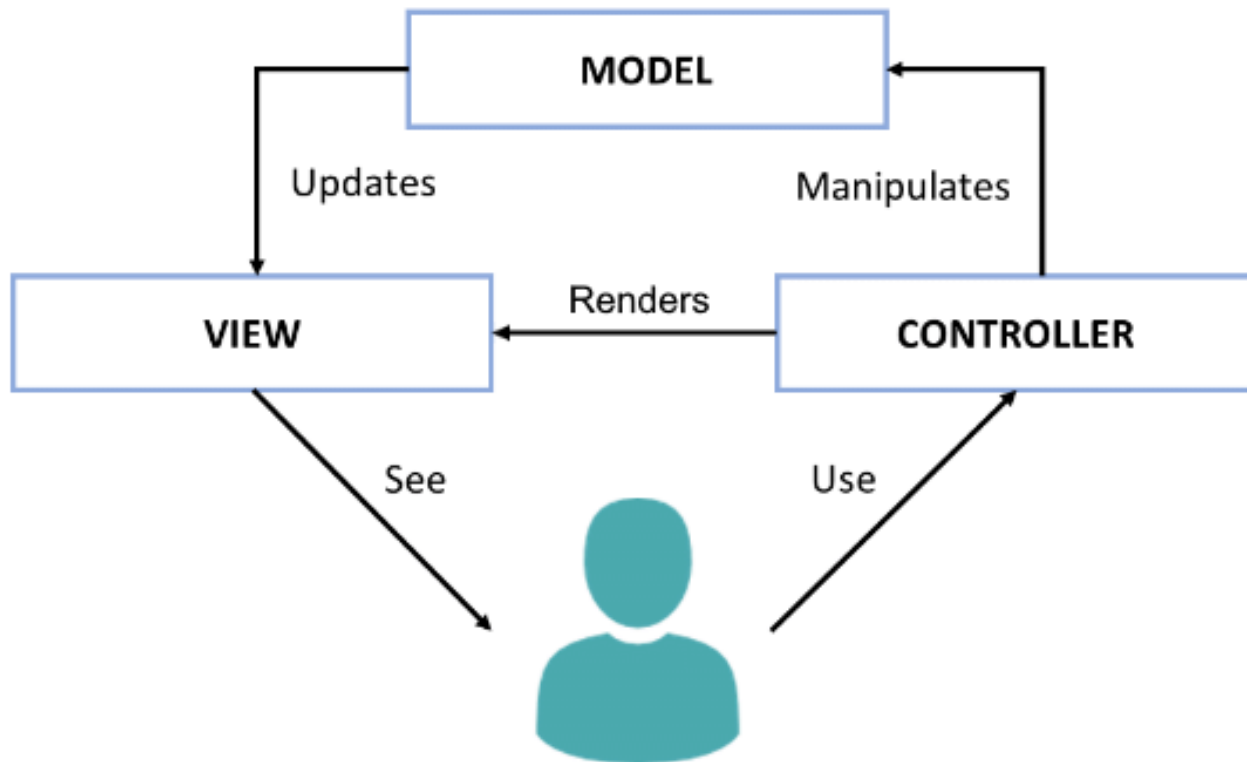
# The MVC Pattern

- **Model–View–Controller** (MVC) is a **software architectural pattern**
- Originally formulated in the late 1970s by Trygve Reenskaug as part of the **Smalltalk** (object-oriented programming language)
- **Code reusability** and **separation of concerns**
- Originally developed for **desktop**, then adapted for **internet applications**



# The Model-View-Controller (MVC) Pattern

- The **Model-View-Controller (MVC)** pattern



- **Controller**

- Handles user actions
- Updates the model
- Renders the view (UI)

- **Model**

- Holds app data

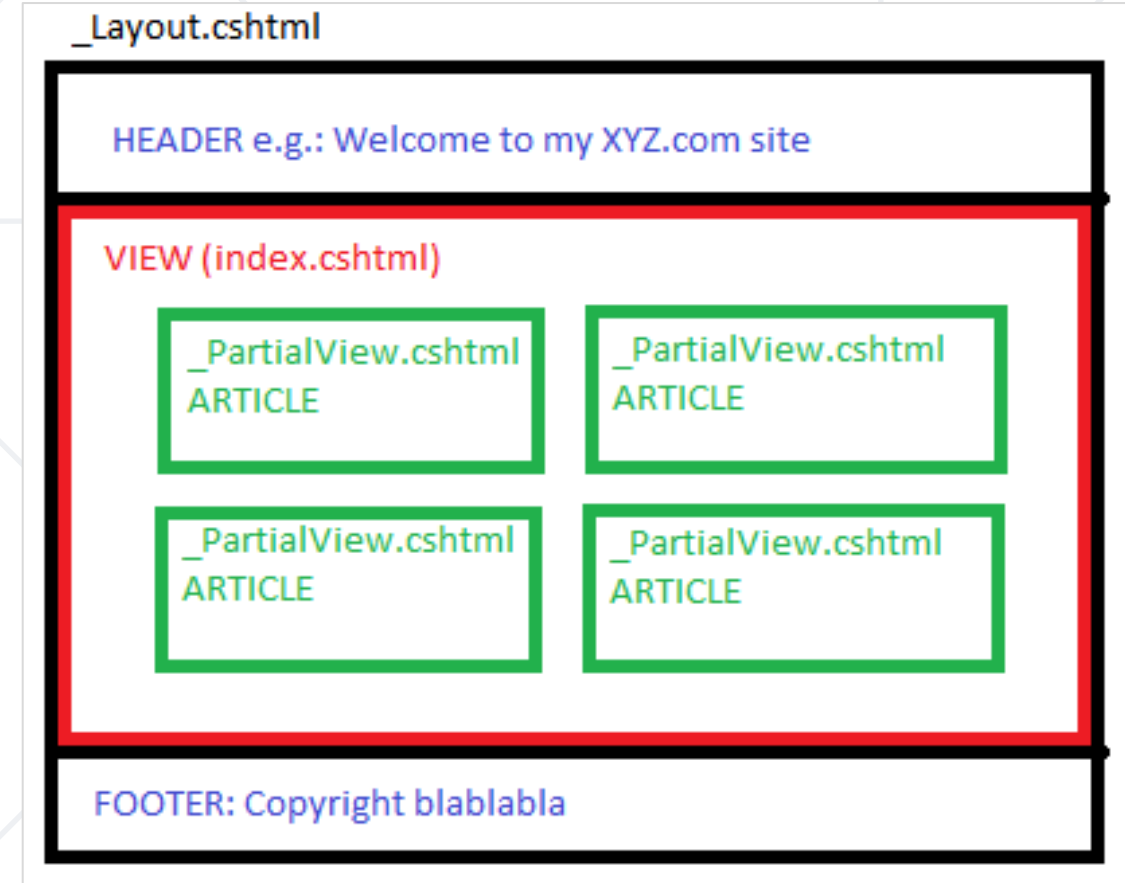
- **View**

- Displays the UI, based on the model data

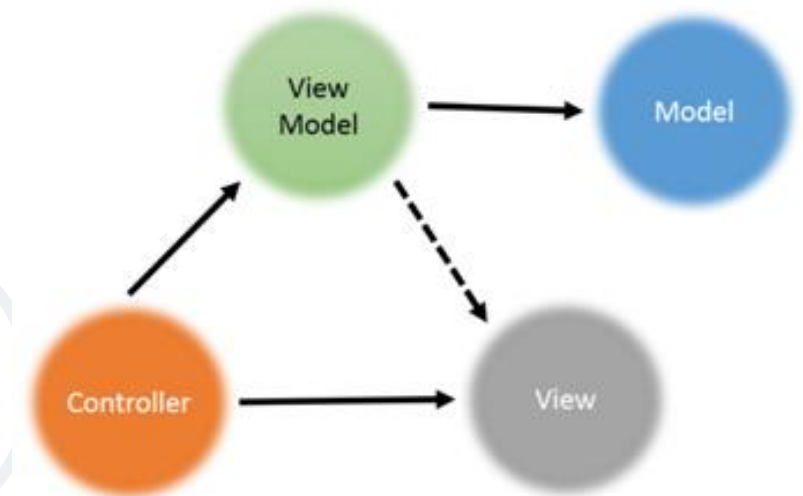
- The **Controller** in **MVC** represents
  - **Processes user's actions** and produces a response
  - Process the requests with the help of **Views** and **Models**
  - A set of classes that handles
    - Communication from the user
    - Overall application flow
    - Application-specific logic
  - Every **Controller** has one or more "**Actions**"

Controller	Action
AccountController	Login
AccountController	Login
AccountController	LogOff
AccountController	MixPanelApiToken

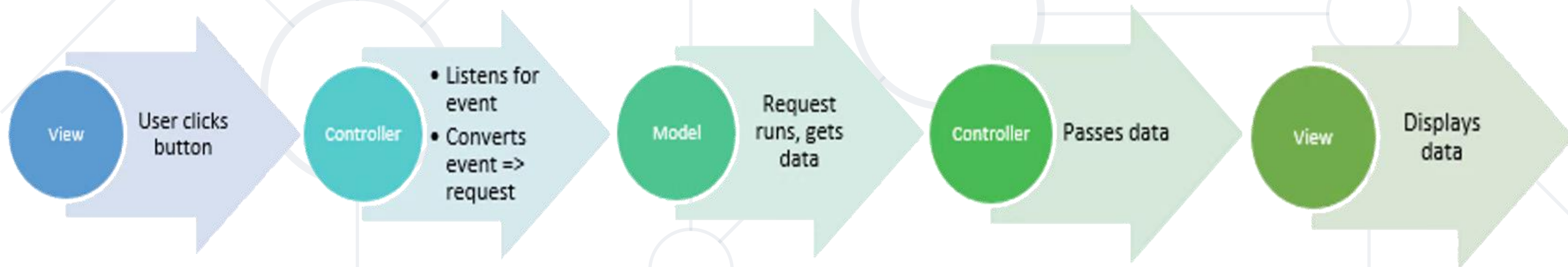
- The **View** in **MVC** represents
  - Defines how the application's user interface (**UI**) will be displayed
  - May support **Master Views (layouts)** and **Sub-Views (partial views or controls)**
  - In Web apps: template to dynamically generate HTML



- The **Model** in **MVC** represents
  - A **set of classes** that describes the **data** we display in the UI
  - May contain **data validation rules**
- Two types of models
  - **View model / binding model**
    - Maps the UI of the Web page to C# class
    - Part of the **MVC** architecture
  - **Database model / domain model**
    - Maps database table to C# class (using ORM)



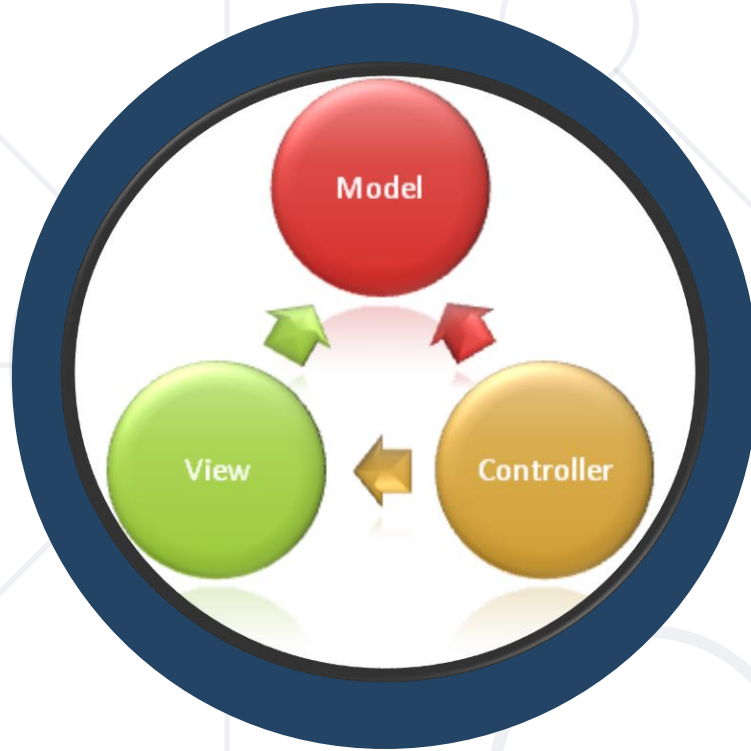
- Incoming **Request** routed to **Controller**
- **Controller** processes **Request** and creates a **Model** (view model)
  - Controller also selects **appropriate result** (for example: **View**)
- **Model** is passed to the **View**
- **The View** transforms **Model** into appropriate output format (HTML)
- **Response** is rendered (**HTTP Response**)



- **Web MVC frameworks** are used to build Web applications
  - It provides the MVC **structure** and **engine** to build Web apps
  - **Controllers** handle HTTP GET / POST requests and render a view
  - **Views** display HTML + CSS, based on the models
  - **Models** hold app data for views, prepared by controllers
- Examples of Web MVC frameworks
  - **ASP.NET Core MVC** (C#), **Spring MVC** (Java), **Express** (JS), **Django** (Python), **Laravel** (PHP), **Ruby on Rails** (Ruby), **Revel** (Go), ...







# ASP.NET Core MVC

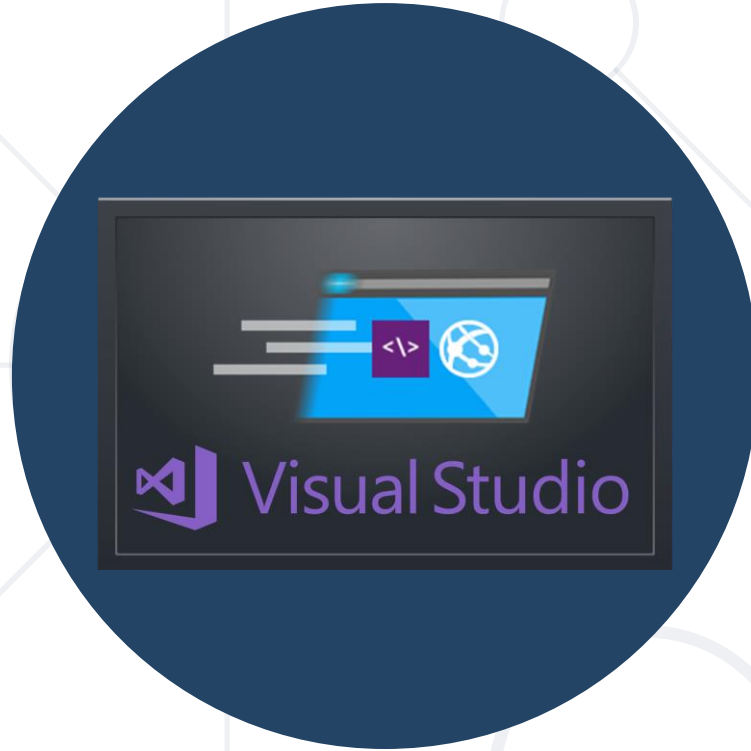
## Overview

- **ASP.NET Core MVC** provides features for building web APIs and web apps
  - Uses the **Model-View-Controller (MVC)** design pattern
  - Lightweight, open source, testable, good tooling
  - **Razor** markup for Razor Pages and MVC views
  - RESTful services with **ASP.NET Core Web API**
    - Built-in support for multiple data formats, content negotiation and CORS
  - Achieve high-quality architecture design, optimizing developer work
    - **Convention over Configuration**
  - **Model binding** automatically maps data from HTTP requests
  - **Model validation** with client-side and server-side validation
  - Often combined with **Entity Framework** for **ORM**



- **Routing** for mapping requests
- **Dependency injection** for injecting components at runtime
- Strongly-typed views with the **Razor view engine**
- **Model binding** automatically maps data from HTTP requests
- **Model validation** with client-side and server-side validation
- **Tag helpers** enable server-side code in HTML elements
- Filters, Areas, Middlewares
- Built-in security features
- **Identity** with users and roles
- And many more...





# Creating an ASP.NET Core MVC App

Project Setup in Visual Studio. What's Inside?

# Create ASP.NET Core MVC App Project



## ASP.NET and web development

Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker supp...



Install this in  
**Visual Studio!**

## Create a new project

ASP.NET Core MVC



Clear all

C#



All platforms



All project types



### ASP.NET Core Web App (Model-View-Controller)

A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

C#

Linux

macOS

Windows

Cloud

Service

Web

Back

Next

# Create ASP.NET Core MVC App: Choose Template

Choose the **.NET version**

Add a **user account functionality**

Additional information

ASP.NET Core Web App (Model-View-Controller) C# Linux macOS Windows Cloud Service Web

Framework ⓘ  
.NET 6.0 (Long Term Support) ▼

Authentication type ⓘ  
Individual Accounts ▼

☒ Configure for HTTPS ⓘ

☐ Enable Docker ⓘ

Docker OS ⓘ  
Linux ▼

☐ Do not use top-level statements ⓘ

Back Create

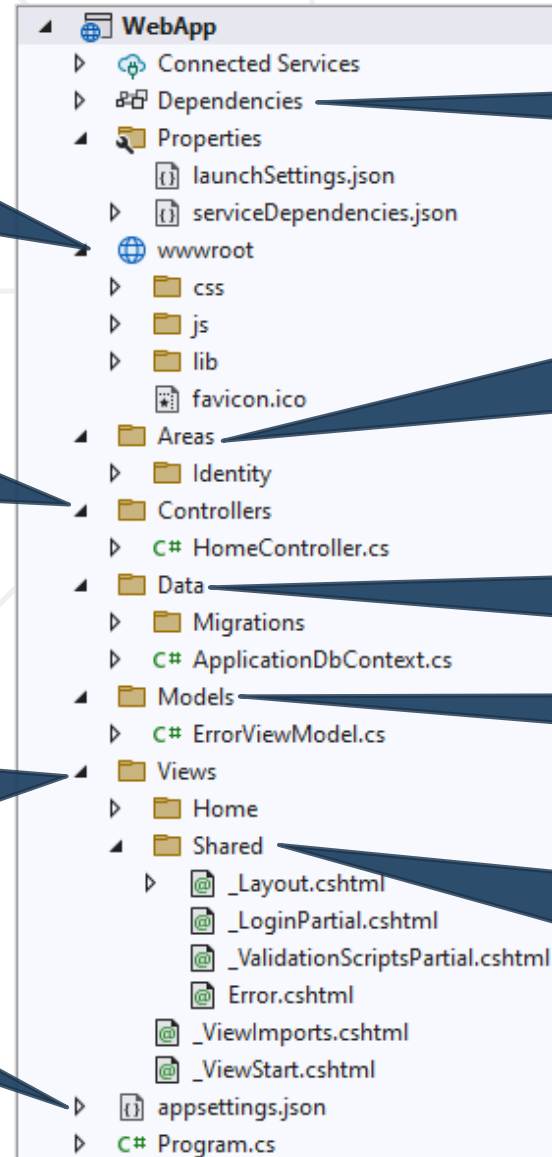
# MVC App: What's Inside?

**Static files:**  
CSS styles, images,  
fonts, ...

**Controller** classes  
holding actions

**Views:**  
HTML templates  
for the pages

**App start** files



**NuGet packages +  
Projects References**

**Areas:** physically  
partition a web app  
in separate units

**Data:** EF models + DB  
context + migrations

**Models:** view models

**Shared views:**  
layout for all pages  
+ partial views

- MVC controllers hold logic to process user actions
- The URL **/Home/About** invokes **HomeController** → **About()**

`\Controllers\HomeController.cs`

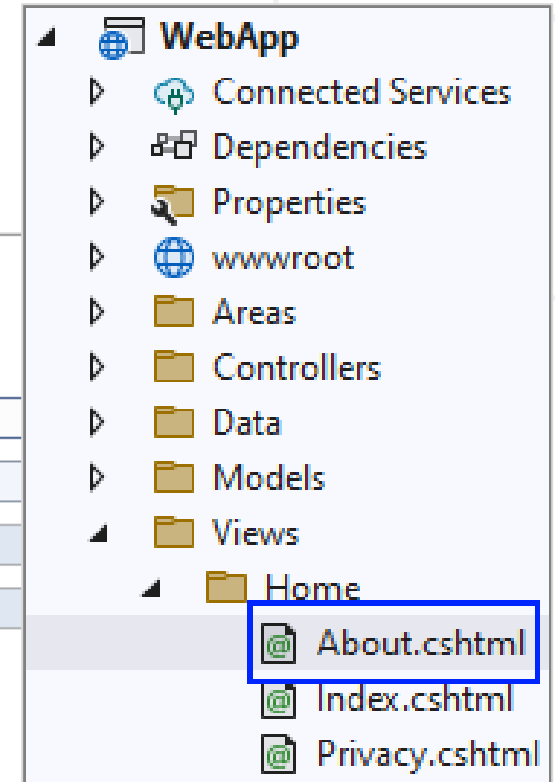
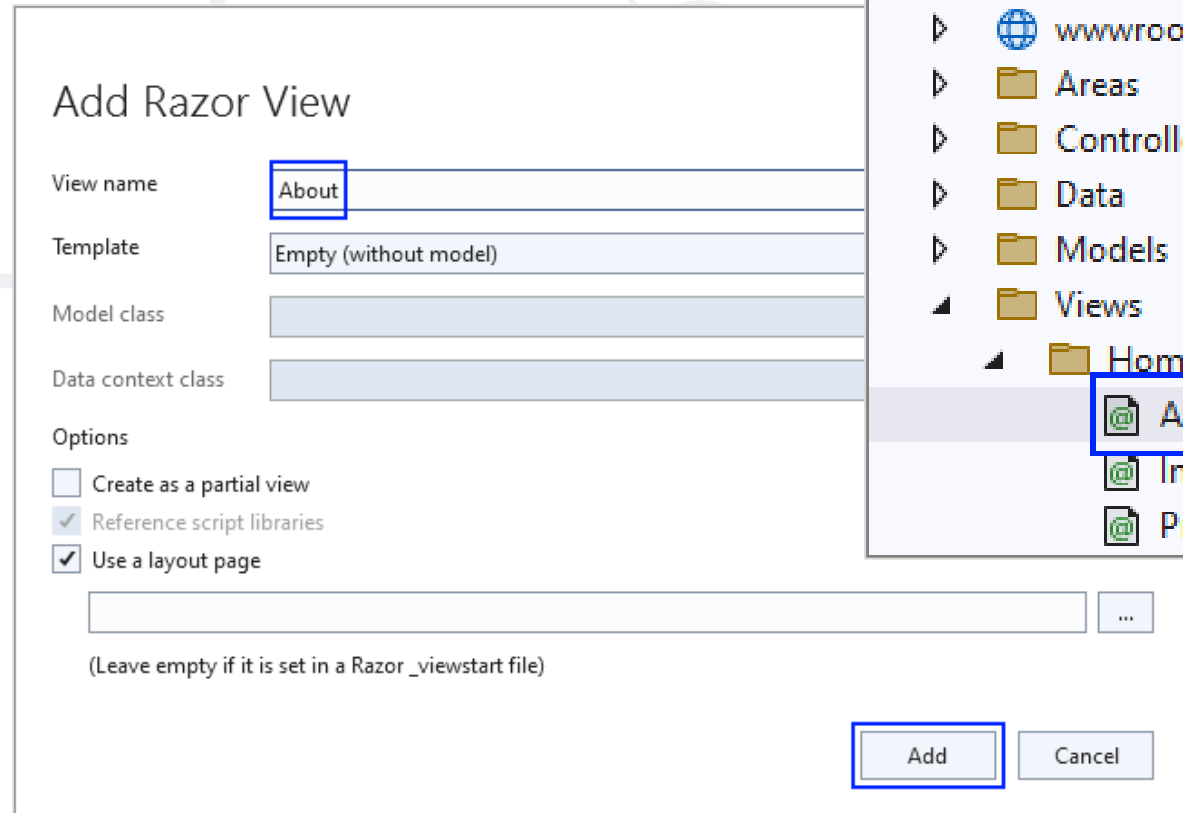
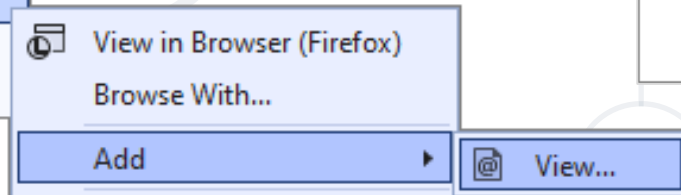
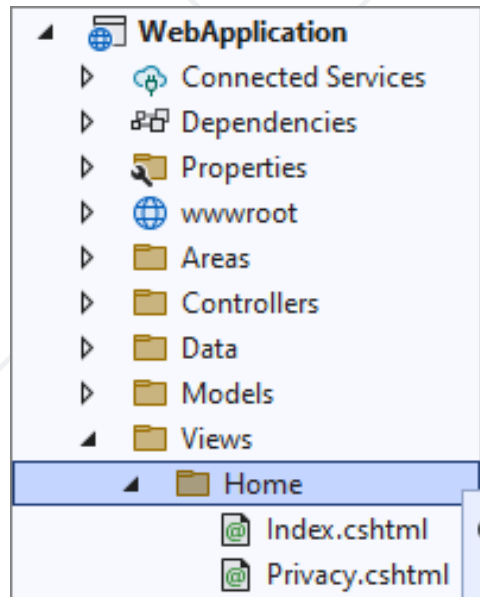
```
public class HomeController : Controller
{
    public ActionResult About()
    {
        ViewBag.Message = "This is an ASP.NET Core MVC app.";
        return View();
    }
}
```

Renders

**`\Views\Home\About.cshtml`**



- **Views** render the **HTML code** for the invoked action
- Create **About.cshtml** view



- ASP.NET MVC uses Razor view engine
- Views combine HTML with C# code

\Views\Home>About.cshtml

```
@{  
    ViewBag.Title = "About";  
}
```

@ { ... } inserts C# code block

```
<h2>@ViewBag.Title</h2>  
<h3>@ViewBag.Message</h3>
```

@Something  
prints a C# variable

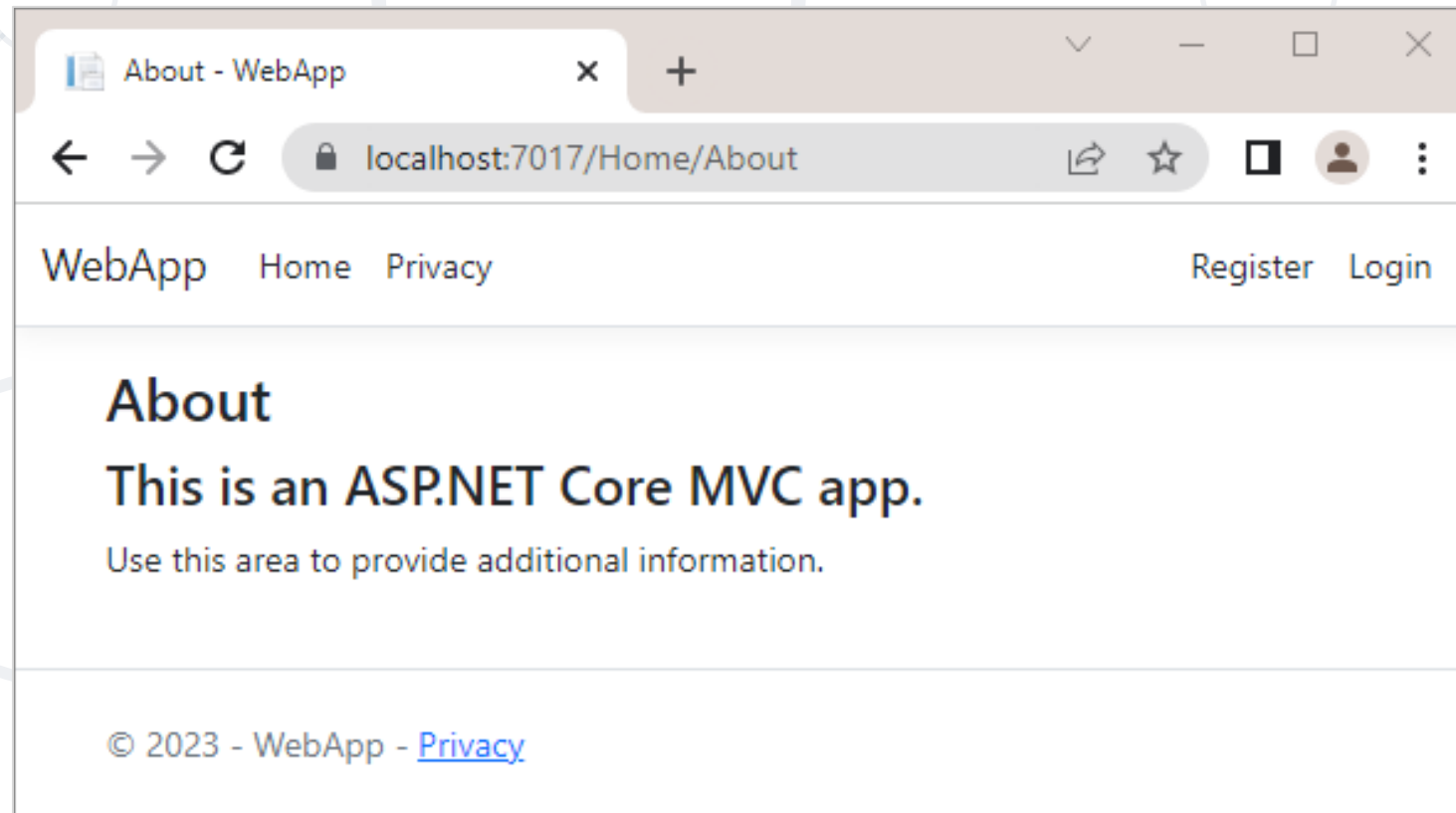
Everything else is  
HTML code

```
<p>Use this area to provide additional information.</p>
```

# The "About" Page in the Browser

- Run the app, by pressing **[Ctrl + F5]**
  - Open the "**About**" page on <https://localhost:44364/Home/About>

The **port number** is auto-generated





# **Controllers and Actions**

- All controllers should be in the "**Controllers**" folder
- Controller naming standard should be **{name}Controller**
- Every controller should inherit the **Controller** class
  - Access to **Request**, **Response**, **HttpContext**, **RouteData**, **TempData**, etc.
- Routes select Controllers in every request

`\Controllers\UsersController.cs`

```
public class UsersController : Controller
{
    public IActionResult All() => View();
}
```

Mapped to URL  
**`"/Users/All"`**

- **Actions** are the ultimate **Request** destination
  - Public controller methods
  - Non-static
  - No return value restrictions
- Actions typically return an **ActionResult**

```
public IActionResult Details(int id)
{
    var viewModel = this.dataService.GetById(id).To<DetailsViewModel>();
    return this.View(viewModel);
}
```

- **Action result** == controller's response to a browser request
  - Represent various **HTTP status codes**
- Inherit from the base **ActionResult** class

```
public IActionResult Index()
{
    return Json(_dataService.GetData());
}
```

```
public IActionResult GetFile()
{
    return File(fileStream, mimeType, fileName);
}
```

```
private const string AppVersion = "v.1.0.0";
public IActionResult Version()
{
    return Content(AppVersion);
}
```

```
public IActionResult LoginConfirm(string username,
    string password)
{
    return Redirect("/Home/Index");
}
```

# Action Results

Name	Framework Behavior	Helping Method
<b>StatusCodeResult</b>	Returns an HTTP Response Result with given Status	<b>StatusCode()</b> / <b>Ok()</b> <b>BadRequest()</b> / <b>NotFound()</b>
<b>JsonResult</b>	Returns data in JSON format	<b>Json()</b>
<b>RedirectResult</b>	Redirects the client to a new URL	<b>Redirect()</b> / <b>RedirectPermanent()</b>
<b>RedirectToRouteResult</b>	Redirect to another action, or another controller's action	<b>RedirectToRoute()</b> / <b>RedirectToAction()</b>
<b>ViewResult</b> <b>PartialViewResult</b>	Response is the responsibility of a view engine	<b>View()</b> / <b>PartialView()</b>
<b>ContentResult</b>	Returns a string literal	<b>Content()</b>
<b>EmptyResult</b>	No response, no content-type header	
<b>FileContentResult</b> <b>FilePathResult</b> <b>FileStreamResult</b>	Return the contents of a file	<b>File()</b> / <b>PhysicalFile()</b>



- **ActionName**(string name)
- **AcceptVerbs**
  - **HttpPost**
  - **HttpGet**
  - **HttpDelete**
  - **HttpOptions**
  - ...
- **NonAction**
- **RequireHttps**
- **etc.**

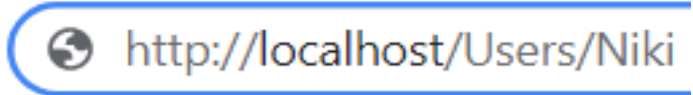
```
public class UsersController : Controller
{
    [ActionName("UserLogin")]
    [HttpPost]
    [RequireHttps]
    public IActionResult Login(
        string username, string password)
    {
        return Content("Logged in!");
    }
}
```

Selectors' **order**  
doesn't matter

- **ASP.NET Core** maps the **data** from the **HTTP request** to action parameters in few ways

- **Routing engine** can pass parameters to actions

- **Routing pattern:** Users/{**username**}



- URL query string can contain parameters

- /Users/ByUsername?**username=NikolayIT**



- HTTP post data can also contain parameters

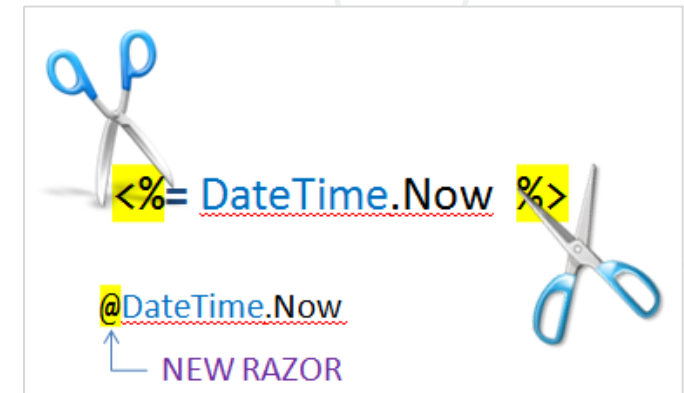
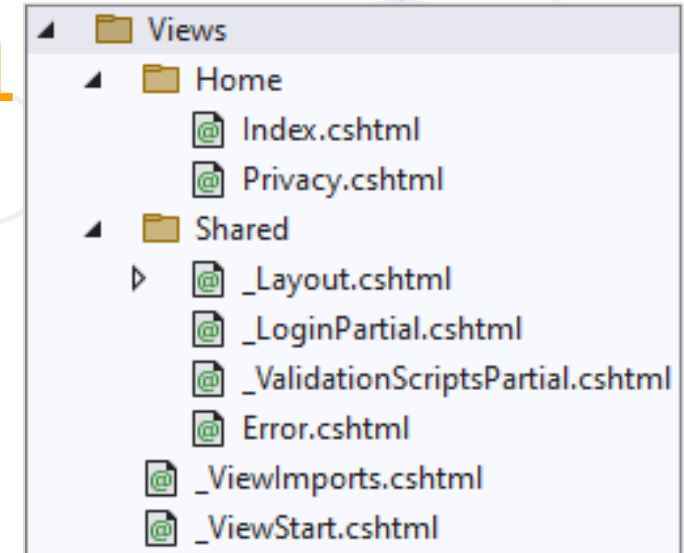
```
public IActionResult
    ByUsername(string username)
{
    return Content(username);
}
```



# **Views and Razor View Engine**

Passing Data to a View

- **Views** render the **HTML code** for the invoked action
- View naming standard is **{ActionName}.cshtml**
- Views should be placed in folder **"/Views/{ControllerName}"**
- A lot of **view engines** available
  - View engines execute code and provide HTML
  - Provide a lot of helpers to easily generate HTML
  - The most popular is **Razor View Engine**

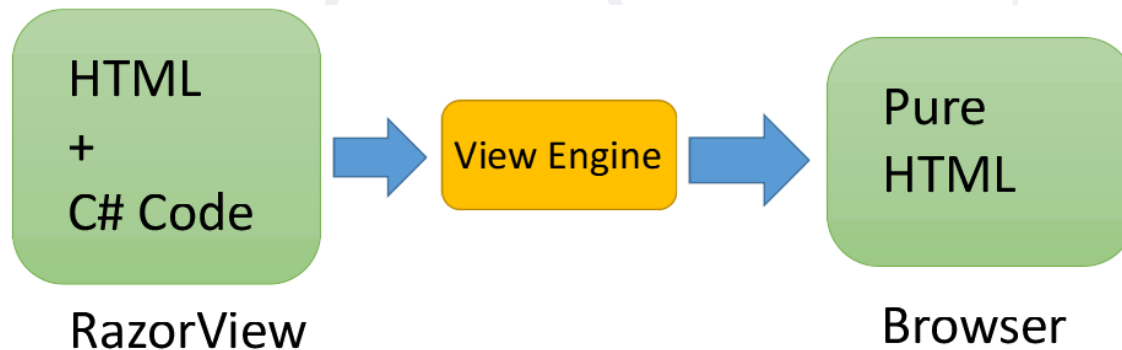


- **Razor** is a markup syntax which helps us write **HTML** and **server-side code** using **C#**
- **Razor View Engine**: use **Razor** with **MVC** to produce HTML
  - **Code blocks** start with a **@** character and don't require explicit closing

```
<div>
    @{
        for (int count = 0; count < 3; count++)
        {
            <p>Count is: @count</p>
        }

        string[] nameArray = { "Mandy", "Peter" };
        foreach (var name in nameArray)
        {
            <p>Your name is: @name</p>
        }
    }
</div>
```

Count is: 0  
Count is: 1  
Count is: 2  
Your name is: Mandy  
Your name is: Peter



# Razor View Engine: Example

- HTML mixed with C# code (@ switches to C#):

```
<div class="row">
  @foreach(var article in Model)
  {
    <article>
      <h2>@article.Title</h2>
      <p>@article.Content</p>
      <small>--@article.Author.FullName</small>
    </article>
  }
</div>
```

HTML  
Syntax

C# code

C# foreach

C# code

# Passing Data to a View – Weakly Typed

- With **ViewBag** (dynamic type):
  - Action: **ViewBag.Message** = "Hello!";
  - View: **@ViewBag.Message**
- With **ViewData** (dictionary)
  - Action: **ViewData["message"]** = "Hello!";
  - View: **@ViewData["message"]**



# ViewBag – Example

`\Controllers\HomeController.cs`

```
public IActionResult Index()
{
    ViewBag.Message = "Hello World!";
    return View();
}
```

`@{ ... }` inserts C# code block

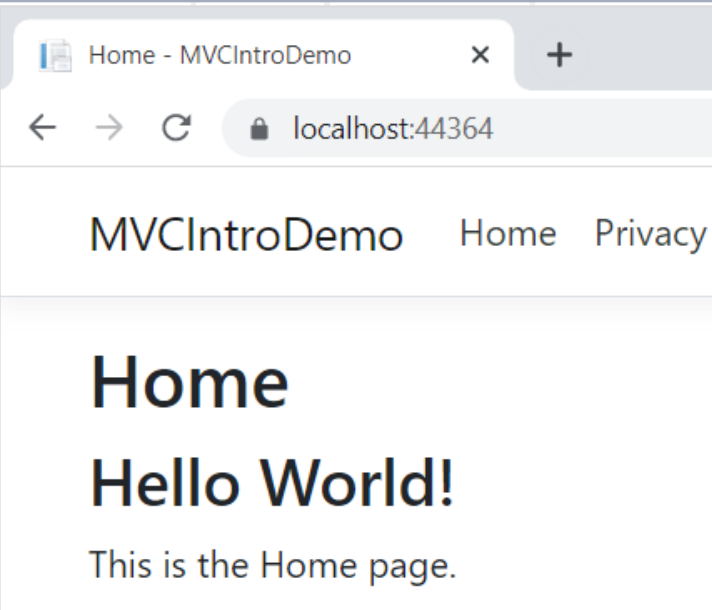
`@Something` prints a C# variable

Everything else is HTML code

`\Views\Home\Index.cshtml`

```
@{
    ViewBag.Title = "Home";
}

<h2>@ViewBag.Title</h2>
<h3>@ViewBag.Message</h3>
<p>This is the Home page.</p>
```





# Passing Data to a View – Strongly Typed – Example

\Controllers\CustomerController.cs

```
public IActionResult Show()
{
    CustomerViewModel customer =
        new CustomerViewModel()
        {
            Name = "Pesho",
            Age = 20
        };
    return View(customer);
}
```

\Models\CustomerViewModel.cs

```
public class CustomerViewModel
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

The **@model** directive makes the **model** available to the **view**

\Views\Customer\Show.cshtml

```
@model CustomerViewModel

@{ViewBag.Title = "View Customer";}

<h2>Current customer: @Model.Name
    (@Model.Age years old).</h2>
```

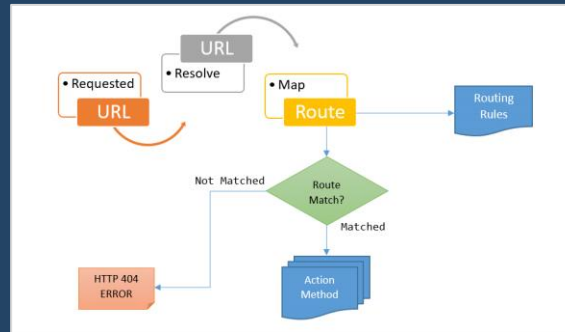
**@Model.Property** prints a model property

View Customer - MVCIntroDemo x +

localhost:44364/Customer/Show

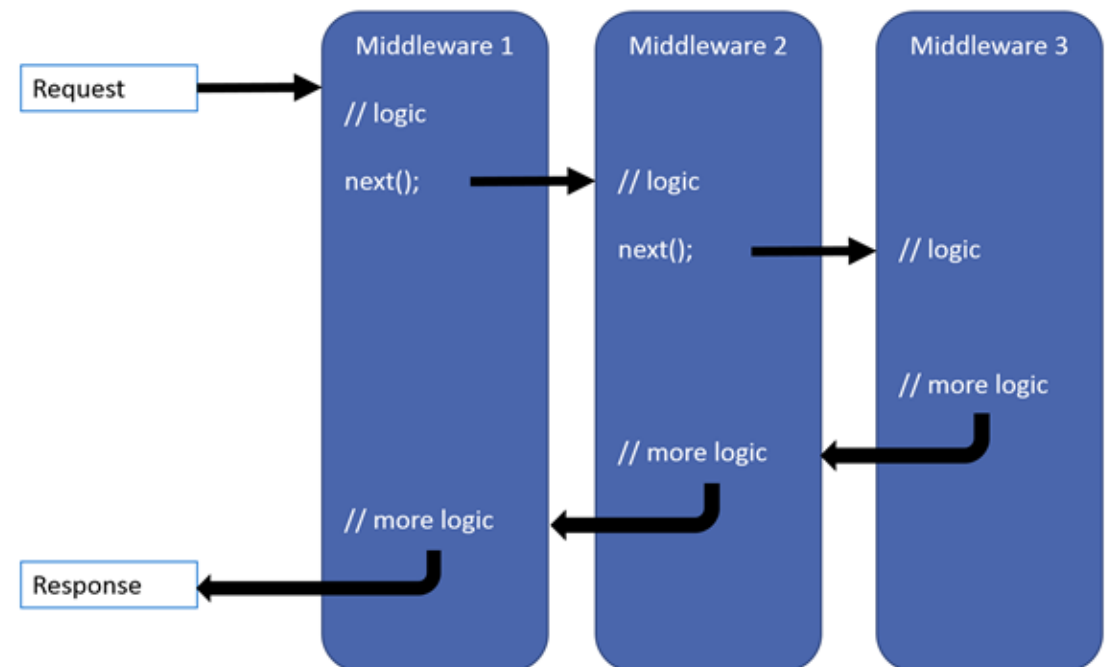
MVCIntroDemo Home Privacy About Numbers NumbersToN Produ

Current customer: Pesho (20 years old).



# ASP.NET Core MVC Routing

- **ASP.NET Core MVC** uses a **middleware** for **Routing** on client requests
  - **Routes** describe how request URL paths should be mapped to **Controller Actions**
  - There are 2 types of Action routing
    - Conventional
    - Attribute



# Conventional Routing (Used by Default)

- Called **Conventional** because it establishes a **convention** for URL paths

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

- Will match a route like **"/Cats/Show/1"**
- Will extract the route values

```
{  
    controller = "Cats",  
    action = "Show",  
    id = "1"  
}
```



**Static Files**

- **Static files** are a necessity for a web application to work
  - Files such as HTML, CSS, JS and different Assets can be served directly to Clients with **ASP.NET Core**

```
app.UseStaticFiles();
```

This will tell the ASP.NET Core App to serve the static files in the "**wwwroot**" directory



# Dependency Injection

## Overview

# What is Dependency Injection?

- **Dependency injection** injects objects at runtime
  - **Register** some service class in the **Program** class

```
services.AddTransient<DataService>();
```

- Later, **inject** the registered class in your controllers

```
public class ProductController : Controller
{
    public ProductController(DataService ds) {
        // Use the injected object "ds"
    }
}
```





**Model Binding**

- **Model binding** in ASP.NET Core MVC maps data from **HTTP requests** to **action method parameters**
  - The parameters may be primitive types or complex types
  - Implemented abstractly, paving the way for reusability in different apps
- The framework binds request data to action parameters by **name**
  - The value of each parameter will be searched, using the **parameter name**
  - **Classes** are mapped using the names of the **public settable properties**



- **Model binding** can look through several **data sources** per Request
  - **Form values** – POST Request parameters
  - **Route values** – The set of Route values provided by the Routing
  - **Query strings** – The query string parameters in the URL
  - Even in headers, cookies, session, etc. in custom model binders
  - Data from these sources are stored as **name-value** pairs
- The framework checks each of the data sources for a parameter value
  - If there is no parameter in the data source, the next in order is checked
  - The data sources are checked in the **order** specified above

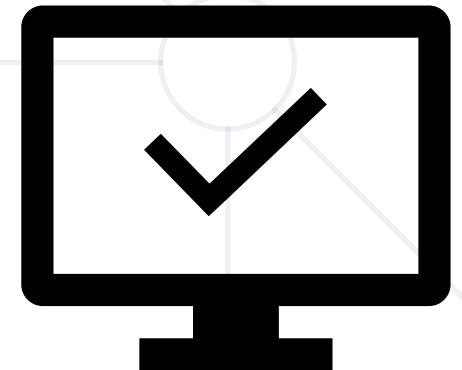
- If binding **fails**, the framework does **not** throw an **error**
  - Every action, accepting **user input**, should check if binding was successful
  - This is done through the **ModelState.IsValid** property
- Each entry in the **controller's ModelState** property is a **ModelStateEntry**
  - Each **ModelStateEntry** contains an **Errors** property
  - It's rarely necessary to query this collection, though
- Default **Model binding** works great for most development scenarios
  - It is also extensible, and you can customize the built-in behavior

- You can easily **iterate** over the errors in the **ModelState**

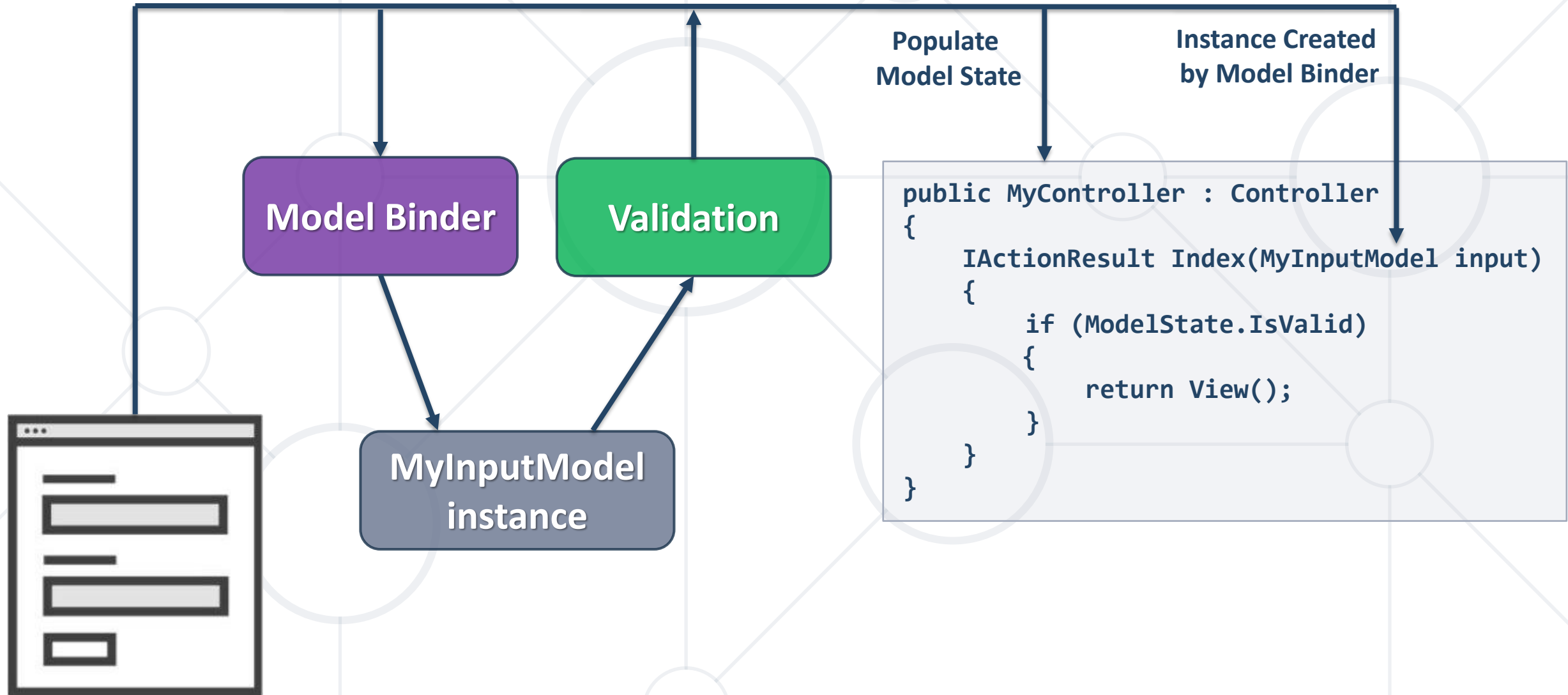
```
public class UsersController : Controller
{
    public IActionResult Register(RegisterUserBindingModel model)
    {
        if(!ModelState.IsValid)
        {
            foreach (var error in ModelState.Values.SelectMany(v => v.Errors))
            {
                DoSomething(error);
            }

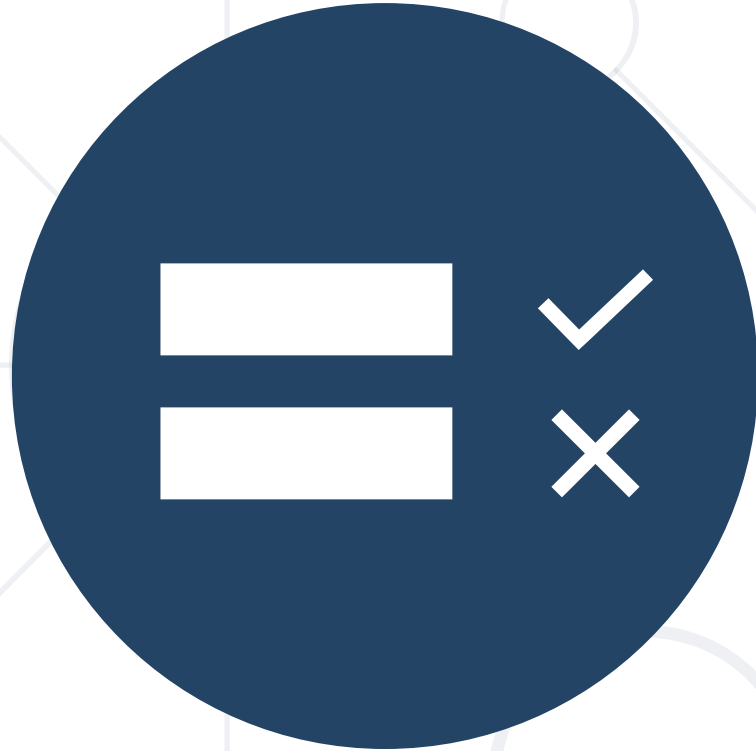
            // TODO: Return Error Page
        }

        return Ok("Success!");
    }
}
```



# Incoming Request to MVC





**Model Validation**

- **Validation** is absolutely necessary before persisting data
  - There may be potential security threats
  - There may be malformed data (**type, size, data constraints**)
- In **ASP.NET Core MVC**, validation happens both on **client** and **server**

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
  <div asp-validation-summary="ModelOnly"></div>
  Email: <input asp-for="Email"/><br />
  <span asp-validation-for="Email"></span><br />
  <div asp-validation-summary="ModelOnly"></div>
  Password: <input asp-for="Password" /><br />
  <span asp-validation-for="Password"></span><br />
  <button type="submit">Register</button>
</form>
```

```
public class RegisterViewModel
{
    [Required]
    [EmailAddress]
    [Display(Name = "Email Address")]
    0 references
    public string Email { get; set; } = null!;

    [Required]
    [DataType(DataType.Password)]
    0 references
    public string Password { get; set; } = null!;
}
```



- .NET provides us with an abstracted validation through **attributes**
  - Some attributes configure model validation by **constraint**
    - Similar to validation on **database fields**
  - Other apply patterns to data to enforce **business rules**
    - **Credit Cards, Phone Numbers, Email Addresses** etc.
- Validation attributes make enforcing these requirements simple
  - They are specified at the **property** or **parameter** level

```
[Required]  
[StringLength(100)]  
0 references  
public string Title { get; set; } = null!;
```

```
[Range(0, 999.99)]  
0 references  
public decimal Price { get; set; }
```

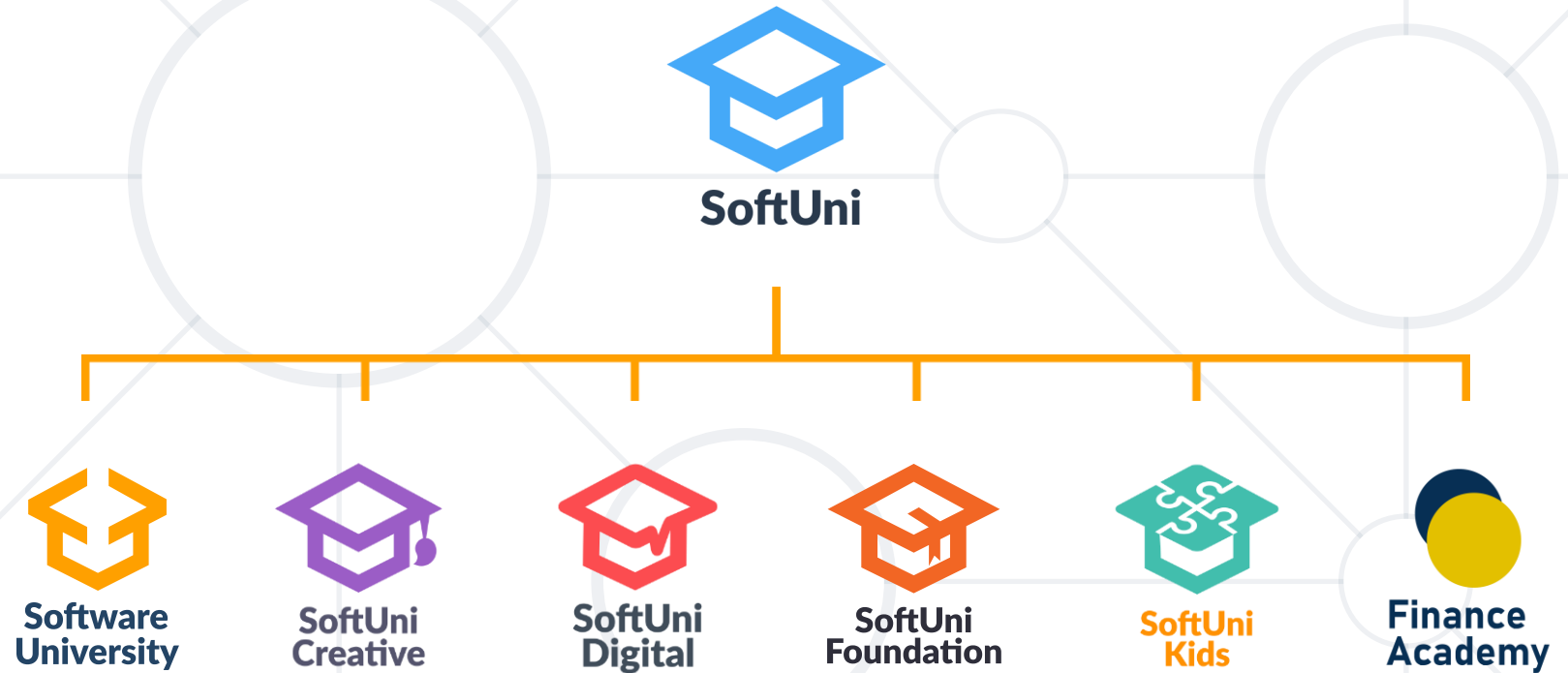
```
public IActionResult SaveUser(  
    [Required, EmailAddress] string Email,  
    [Required, StringLength(1000)] string Username)
```

Attribute	Description
[CreditCard]	Validates the property has a <b>credit card</b> format
[Compare]	Validates 2 properties in a model <b>match</b> . (Useful for <b>password confirmation</b> )
[EmailAddress]	Validates the property has an <b>email</b> format
[Phone]	Validates the property has a <b>telephone</b> format
[Range]	Validates the property <b>value</b> falls within the <b>given range</b>
[RegularExpression]	Validates the data <b>matches</b> the specified <b>regular expression</b>
[Required]	Makes the property <b>required</b> . Value <b>cannot</b> be <b>null</b>
[StringLength]	Validates that a string property has at most the <b>given maximum length</b>
[Url]	Validates the property has a <b>URL</b> format

- **ASP.NET Core** is a great platform for developing Web apps
- MVC **Controllers** and **Actions**
- MVC **Views** and **Razor**
- **Routing**
- **Static Files**
- Dependency Injection
- **Model Binding** and **Model Validation**



# Questions?



# SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)



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

