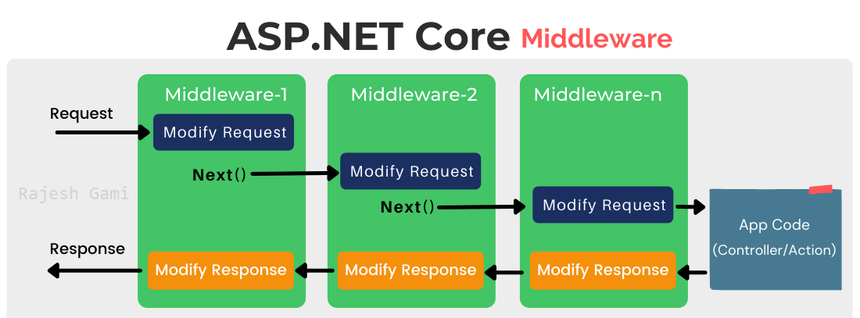
**Asp.Net Core 6**

**MiddleWare:**

Middleware in .NET Core 6 is a key component of the request/response pipeline. It processes HTTP requests as they travel through the pipeline and can either pass the request to the next middleware or generate a response. Middleware is essential for handling tasks such as authentication, logging, error handling, and more

**Key Concepts of Middleware**

1. **Pipeline Execution**:
   1. Middleware components are executed sequentially in the order they are added to the pipeline.
   2. Each middleware can either:
      1. Perform some work and pass control to the next middleware.
      2. Short-circuit the pipeline and produce a response without calling the next middleware.
2. **Request Delegates**:
   1. Middleware is implemented as a delegate that takes an HttpContext and returns a Task.
3. **Built-in Middleware**:
   1. .NET Core provides several built-in middleware, such as:
      1. UseStaticFiles: Serves static files.
      2. UseRouting: Matches incoming requests to route handlers.
      3. UseAuthentication and UseAuthorization: Handles authentication and authorization.
4. **Custom Middleware**:
   1. Developers can create custom middleware to handle specific tasks.



**Routing:**

Core Concept:

* + The process of matching incoming URL requests to the appropriate controller actions within your application.
  + Acts like a traffic director, guiding requests to the correct destination.

Key Components:

* Route Templates:
  + Define URL patterns with placeholders (e.g., {controller}, {id}) for dynamic segments.
* Controller Actions:
  + Methods within controllers that handle specific requests.
* Middleware:
  + A pipeline of components that process requests and responses. The routing middleware is crucial for matching URLs to actions.

Types of Routing:

1. Conventional Routing:
   * Configured in program.cs using MapControllerRoute.
   * Follows conventions to determine URLs based on controller and action names.

**Example:**

C#

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

});

1. Attribute Routing:
   * Defined using attributes like [Route] on controllers or actions.
   * Provides more control over URL patterns for customization.

**Example:**

[Route("api/[controller]")]

public class ProductsController : ControllerBase

{

[HttpGet("{id}")]

public IActionResult GetProduct(int id) { /\* ... \*/ }

}

Benefits of Routing:

* Clean URLs: Creates user-friendly and SEO-friendly URLs.
* Maintainability: Improves code organization and maintainability.
* Flexibility: Supports customization and RESTful API implementation.

Choosing the Right Type:

* Conventional: Suitable for simpler scenarios with default conventions.
* Attribute: Preferred for complex scenarios, RESTful APIs, and when precise URL control is needed.

Key Considerations:

* Clarity: Prioritize creating clear and meaningful URLs.
* Maintainability: Choose a routing approach that enhances code maintainability.
* Performance: Consider the impact of complex routes on application startup time.

**In Summary:**

Routing is essential for building well-structured and efficient ASP.NET Core applications. By understanding and effectively using routing mechanisms, you can create a seamless and user-friendly experience for your application's users.s

|  |  |  |
| --- | --- | --- |
| Type | Example | Purpose |
| Static Routes | /home | Fixed URL paths. |
| Parameterized Routes | /product/{id} | Capture dynamic data in the URL. |
| Route Constraints | /user/{id:int} | Restrict parameter types. |
| Attribute Routing | [HttpGet("{id}")] | Route definitions in attributes. |
| Conventional Routing | {controller}/{action}/{id?} | Common pattern-based routing in MVC. |
| Endpoint Routing | endpoints.MapGet("/") | Centralized route definitions. |
| Catch-All Routes | /files/{\*filePath} | Capture remaining segments in the URL. |
| Conditional Routing | UseWhen() | Apply logic to determine route handling. |
| API Routing | [Route("api/[controller]")] | Routing for REST APIs. |
| Razor Pages Routing | /Pages/About.cshtml | Implicit routing based on file paths. |
| Custom Routing | Middleware handling /custom | Fully customized routing logic. |

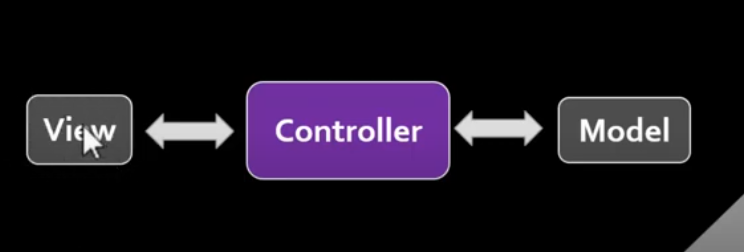
These routing types make .NET Core highly versatile for building web applications of any complexity.

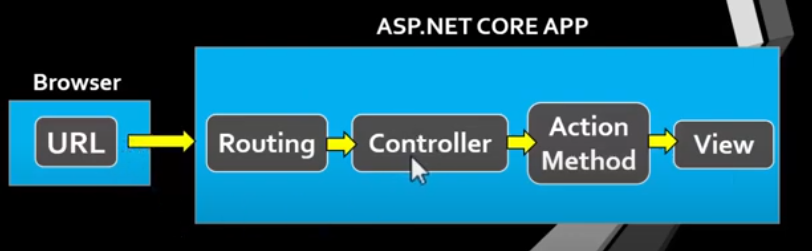
**Controllers:**

Controllers are fundamental building blocks in ASP.NET Core 6 applications. They act as the intermediaries between incoming HTTP requests and the responses generated by your application.

Core Responsibilities

* Handling HTTP Requests: Controllers receive incoming HTTP requests from the web browser or other clients.
* Processing Data: They process the data associated with the request, which might involve interacting with models to retrieve or update data from databases or other sources.
* Selecting Views (MVC) or Returning Responses (Web API):
  + In ASP.NET Core MVC applications, controllers typically select appropriate views to render the user interface (UI) based on the processed data.
  + In ASP.NET Core Web API applications, controllers focus on returning data (often in JSON or XML format) directly to the client.





Key Characteristics

* Inheritance: Controllers inherit from the base class Controller in the Microsoft.AspNetCore.Mvc namespace.
* Action Methods: These are public methods within controllers that correspond to specific HTTP verbs (GET, POST, PUT, DELETE, etc.). They handle incoming requests and generate responses.
* Attributes: Controllers and action methods can be decorated with attributes to define routes, specify HTTP verbs, and manage other aspects of request handling.

In essence, controllers serve as the coordinators, directing the flow of data and responses within your ASP.NET Core application. They bridge the gap between user interactions and the underlying data layer, ensuring a well-structured and maintainable application architecture.

**Action Methods:**

In ASP.NET Core 6, action methods are the workhorses of controllers. They are public methods within a controller class that handle incoming HTTP requests and generate the corresponding responses.

Core Responsibilities

* Responding to HTTP Requests: Action methods are specifically designed to handle requests based on the HTTP verb (GET, POST, PUT, DELETE, etc.) associated with the request.
* Processing Data: They process the data received from the request, which might involve:
  + Extracting data from the request body, query string, or route parameters.
  + Interacting with models to retrieve or update data from databases or other sources.
  + Performing calculations or other business logic.
* Returning Responses: Action methods return an appropriate response object that gets sent back to the client. This response can take various forms:
  + In ASP.NET Core MVC applications, the response typically specifies a view to render the UI.
  + In ASP.NET Core Web API applications, the response often contains data formatted in JSON or XML.

Key Characteristics

* Public Methods: Action methods must be public methods within a controller class.
* HTTP Verb Attributes: They are typically decorated with attributes that specify the supported HTTP verb (e.g., [HttpGet], [HttpPost]). This helps map the action method to the appropriate request type.
* Return Types: Action methods can return various types of objects depending on the application's needs. Some common return types include:
  + IActionResult: The base interface for MVC action methods, providing flexibility for different response scenarios (views, redirects, HTTP status codes, etc.).
  + Specific data types (e.g., string, int, custom models) for returning data directly in Web API applications.

Example

public class ProductsController : Controller

{

[HttpGet]

public IActionResult Index()

{

// Get product data from a model

var products = ProductService.GetProducts();

// Pass the data to the view

return View(products);

}

[HttpPost]

public IActionResult Create(Product product)

{

// Validate and save the product using a model

ProductService.SaveProduct(product);

// Redirect to a different action or page

return RedirectToAction("Index");

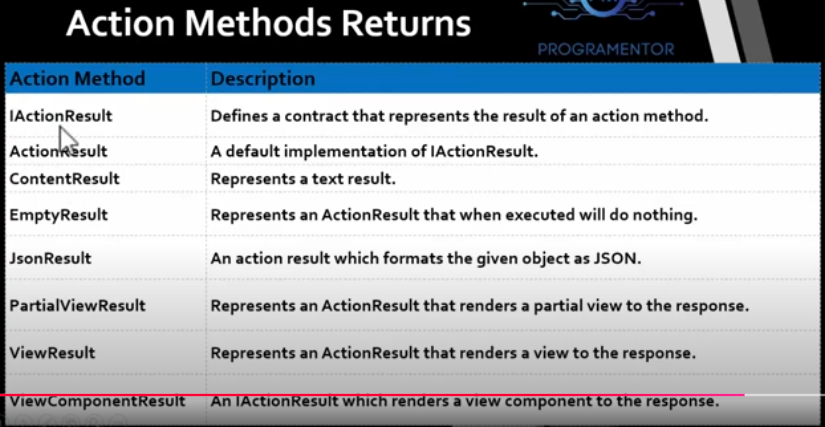
}

}

In this example:

* The Index action method handles GET requests and returns an IActionResult, likely to render a view with product data.
* The Create action method handles POST requests for creating new products and returns a redirection after saving the product.

**In summary, action methods are the heart of controllers in ASP.NET Core 6. They provide a structured way to handle different types of HTTP requests, process data, and generate appropriate responses for your web application.**



**View**

**Razor in ASP.NET Core 6**

**Overview**

* Razor is a view engine used to create dynamic web pages by combining C# code with HTML.
* It is employed in **Razor Pages**, **MVC Views**, and **Razor Components** (Blazor).
* Uses the @ symbol to transition between server-side code and HTML.

**Key Features**

* Lightweight and syntax-rich.
* HTML output is encoded by default to prevent XSS.
* Integrates with ASP.NET Core features like Tag Helpers and Dependency Injection.
* Supports reusable components (e.g., layouts, partial views).

**Razor Syntax**

1. **Inline Code**: Use @ to embed C# in HTML.

<p>@DateTime.Now.ToString("F")</p>

1. **Code Blocks**:

@{

var name = "Manoj";

}

<p>Hello, @name!</p>

1. **Control Structures**:

@for (int i = 0; i < 5; i++)

{

<p>Number: @i</p>

}

@if (DateTime.Now.Hour < 12)

{

<p>Good Morning!</p>

}

else

{

<p>Good Evening!</p>

}

**Types of Razor Implementations**

1. **Razor Pages**:
   * Page-centric model.
   * Each page has a .cshtml file (UI) and a .cshtml.cs file (PageModel).
2. **Razor Views**:
   * Used in MVC applications to render the "View" in MVC.
3. **Razor Components**:
   * Used in Blazor for building reusable and interactive UI.

**Special Razor Features**

1. **Layouts**:
   * Shared structure for pages/views.

<!-- \_Layout.cshtml -->

<html>

<body>

@RenderBody()

</body>

</html>

**Index.cshtml**

html

Copy code

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

1. **Partial Views**:
   * Reusable views embedded within other views.

@Html.Partial("\_PartialViewName")

1. **Tag Helpers**:

<form asp-controller="Home" asp-action="Submit">

<button type="submit">Submit</button>

</form>

1. **Model Binding**:
   * Use @model to bind data to the view.

@model Product

<h1>@Model.Name</h1>

**Razor Pages Example**

**Index.cshtml**

@page

@model IndexModel

<h1>Hello, @Model.Name!</h1>

**Index.cshtml.cs**

public class IndexModel : PageModel

{

public string Name { get; set; }

public void OnGet() => Name = "Manoj";

}

**Advantages**

1. Clean separation of concerns.
2. Easy-to-learn syntax.
3. Strong integration with ASP.NET Core.
4. Built-in HTML encoding for security.
5. Supports asynchronous programming.

These notes summarize Razor's functionality and features in ASP.NET Core 6, making it a powerful tool for building dynamic and secure web applications.

**Layouts in ASP.NET Core Razor**

**Layouts** in Razor allow you to define a shared structure for multiple views or pages, promoting consistency and reducing redundancy. A layout typically includes the common HTML elements like headers, footers, navigation menus, and scripts that are shared across multiple views or pages.

**Key Features of Layouts**

1. **Centralized Structure**:
   * Define the common design (header, footer, sidebar, etc.) once and reuse it across pages.
2. **Dynamic Content Rendering**:
   * Use @RenderBody() or @RenderSection() to include specific content from individual views.
3. **Flexibility**:
   * Multiple layouts can be used in an application for different sections or themes.
4. **Consistency**:
   * Enforces uniformity in UI design across views.

**Creating and Using Layouts**

**Step 1: Create a Layout File**

* Layout files are usually placed in the Views/Shared folder and named \_Layout.cshtml.

**Example: \_Layout.cshtml**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

<title>@ViewData["Title"] - My Application</title>

<link rel="stylesheet" href="site.css" />

</head>

<body>

<header>

<h1>My Application</h1>

<nav>

<a href="/">Home</a>

<a href="/About">About</a>

</nav>

</header>

<main>

@RenderBody()

</main>

<footer>

<p>&copy; @DateTime.Now.Year - My Application</p>

</footer>

</body>

</html>

**Step 2: Specify Layout in a View**

Set the layout for individual views by using the Layout property.

**Example: Index.cshtml**

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Welcome to Razor Layouts!</h2>

<p>This is the content of the Index page.</p>

Alternatively, set the layout globally in the \_ViewStart.cshtml file.

**ViewStart in ASP.NET Core 6**

The \_ViewStart.cshtml file in ASP.NET Core 6 is used to define common settings or logic that apply to all Razor views in an application. It simplifies the management of repetitive tasks, such as specifying a layout, ensuring consistency across views.

**Purpose of \_ViewStart.cshtml**

1. **Set Default Layout**:
   * Automatically applies a layout to all views unless explicitly overridden.
2. **Reduce Redundancy**:
   * Eliminates the need to specify a layout or perform common initialization logic in every view.
3. **Maintain Consistency**:
   * Ensures a uniform configuration across views.

**Location of \_ViewStart.cshtml**

* It is typically placed in the Views folder of the application.
* You can also create a \_ViewStart.cshtml file in subfolders to define folder-specific settings, overriding the parent \_ViewStart.cshtml.

**Basic Syntax of \_ViewStart.cshtml**

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**How \_ViewStart.cshtml Works**

1. \_ViewStart.cshtml is executed before rendering any view in the same folder or its subfolders.
2. If no \_ViewStart.cshtml is present in a subfolder, the one in the root Views folder applies.
3. A specific view can override the layout set in \_ViewStart.cshtml.

**Examples**

**1. Default Usage**

**\_ViewStart.cshtml**

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**Index.cshtml**

<h1>Welcome to the Home Page</h1>

* The Index.cshtml view will automatically use the layout specified in \_ViewStart.cshtml.

**2. Overriding the Layout in a View**

**\_ViewStart.cshtml**

@{

Layout = "~/Views/Shared/\_Layout.cshtml";

}

**Index.cshtml**

@{

Layout = "~/Views/Shared/\_CustomLayout.cshtml";

}

<h1>Home Page with a Custom Layout</h1>

* The Index.cshtml view uses \_CustomLayout.cshtml instead of the layout specified in \_ViewStart.cshtml.

**3. Folder-Specific \_ViewStart.cshtml**

* You can add a \_ViewStart.cshtml in subfolders for different layouts or logic.

**Views/Shared/\_AdminViewStart.cshtml**

@{

Layout = "~/Views/Shared/\_AdminLayout.cshtml";

}

**Admin/Index.cshtml**

<h1>Welcome to the Admin Dashboard</h1>

* Views in the Admin folder will use the \_AdminLayout.cshtml.

# **Pass Data From Controller To View**

Types:

* View Data
* View Bag
* Temp Data
* Strongly Type View

1. **ViewData**

**ViewData** is a dictionary used to pass data from a controller (or Razor Page) to a view in ASP.NET Core applications. It provides a way to transfer data that is not strongly typed and is accessible using a key-value pair mechanism.

**Characteristics of ViewData**

1. **Type**:
   * ViewData is of type ViewDataDictionary.
2. **Scope**:
   * Available only for the current request.
   * Data is passed from the controller to the view.
3. **Weakly Typed**:
   * Data is stored as object and requires casting to retrieve.
4. **Useful for Small Data Transfers**:
   * Often used for dynamic or short-lived data like titles or messages.

**Using ViewData with Layouts**

Pass data from a controller to a layout view via ViewData.

**In Controller:**

ViewData["AppTitle"] = "My Application";

**In Layout (\_Layout.cshtml):**

<title>@ViewData["AppTitle"]</title>

1. **ViewBag**

**ViewBag** is a dynamic object provided by ASP.NET Core for passing data from a controller to a view. It uses the dynamic keyword, allowing developers to add and retrieve properties without explicitly defining them.

**Characteristics of ViewBag**

1. **Type**:
   * ViewBag is a dynamic object.
2. **Scope**:
   * It is valid only for the current request.
   * The data is passed from the controller to the view.
3. **Weakly Typed**:
   * It does not provide compile-time type checking, which can lead to runtime errors.
4. **Convenient Syntax**:
   * ViewBag properties are accessed like standard properties (e.g., ViewBag.Title).

**Basic Syntax**

**Setting Data in Controller**

public IActionResult Index()

{

ViewBag.Title = "Home Page";

ViewBag.Message = "Welcome to ASP.NET Core!";

return View();

}

**Accessing Data in View**

<h1>@ViewBag.Title</h1>

<p>@ViewBag.Message</p>

**TempData**

**TempData** is a dictionary used to store data that needs to persist between two requests in ASP.NET Core. It is particularly useful for scenarios like redirection where you want to pass temporary data from one action to another.

**Characteristics of TempData**

1. **Type**:
   * TempData is of type ITempDataDictionary.
2. **Scope**:
   * Data persists until it is read or the request ends.
   * Primarily used for scenarios involving redirection.
3. **Temporary Storage**:
   * Data in TempData is automatically removed after it is read.
4. **Underlying Storage**:
   * Uses cookies or session to store data.

**Basic Syntax**

**Assigning Data in Controller**

public IActionResult Index()

{

TempData["Message"] = "Welcome to TempData!";

return RedirectToAction("About");

}

**Accessing Data in Another Action**

public IActionResult About()

{

var message = TempData["Message"];

return View();

}

**Accessing Data in View**

<p>@TempData["Message"]</p>

**Features**

1. **Data Persistence**:
   * Data persists across actions or redirects, unlike ViewData or ViewBag.
2. **Automatically Cleared**:
   * Once accessed, the data is removed from TempData.
3. **Manual Retention**:
   * Use TempData.Keep() to retain data for another request.
   * Use TempData.Peek() to read data without clearing it.

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | TempData | ViewData | ViewBag |
| Type | ITempDataDictionary | ViewDataDictionary | Dynamic |
| Scope | Across requests (until read) | Current request | Current request |
| Persistence | Redirection or multiple requests | Single request | Single request |
| Strong Typing | No | No | No |
| Usage | Temporary data between actions | Short-lived data for views | Short-lived data for views |
|  | Need to Typecast | Need to Typecast | No need to Typecast |

**Strongly Typed Views**

A **Strongly Typed View** in ASP.NET Core is a Razor view that is associated with a specific model class. This association enables the view to access the properties of the model in a type-safe manner, providing IntelliSense support and compile-time checking.

**Benefits of Strongly Typed Views**

1. **Type Safety**:
   * Compile-time checking prevents runtime errors due to mismatched property names or types.
2. **IntelliSense**:
   * Provides auto-completion for model properties in the Razor view.
3. **Clean Code**:
   * Eliminates the need for casting or using loosely typed data structures like ViewBag or ViewData.
4. **Enhanced Readability**:
   * Clearly defines what data the view expects.

**Passing Data to Strongly Typed Views**

1. **Single Model Instance**:
   * Pass a single object of the model type to the view.

var product = new Product { Name = "Phone", Price = 699.99m };

return View(product);

1. **List of Models**:
   * Pass a collection of models to the view.

**Controller Action:**

public IActionResult Index()

{

var products = new List<Product>

{

new Product { Id = 1, Name = "Laptop", Price = 999.99m },

new Product { Id = 2, Name = "Phone", Price = 699.99m }

};

return View(products);

}

**View:**

@model IEnumerable<Product>

<h1>Product List</h1>

<ul>

@foreach (var product in Model)

{

<li>@product.Name - $@product.Price</li>

}

</ul>

**Models**

**Purpose of Models**

* Represent the application’s data and business logic.
* Facilitate data transfer between layers (Controller ↔ View ↔ Database).
* Support validation and encapsulate rules.

**Types of Models**

1. **Domain Models**:
   * Represent core business entities.
   * Example: Product, User, Order.
2. **View Models**:
   * Designed for the presentation layer.
   * Used to pass data from controllers to views.
   * Example: ProductViewModel.
3. **DTOs (Data Transfer Objects)**:
   * Simplified objects for transferring data between layers or services.
   * Example: UserDTO.
4. **Input Models**:
   * Capture data from forms or user input.
   * Example: LoginModel.

**Defining a Model**

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

public string Description { get; set; }

}

**Using Models**

1. **Passing Data to Views**:
   * Controllers use models to send data to views.

public IActionResult Index()

{

var product = new Product { Name = "Laptop", Price = 999.99m };

return View(product);

}

1. **Data Binding**:
   * Automatically maps HTTP request data to model properties in actions.
2. **Validation**:
   * Models can include attributes for validation, such as [Required] and [Range].

**Validation with Data Annotations**

Common attributes:

* [Required] – Ensures the property is not null or empty.
* [StringLength(50)] – Limits the length of a string.
* [Range(1, 100)] – Specifies a range of values.
* [EmailAddress] – Validates an email address.

**Example:**

public class User

{

[Required]

[StringLength(50)]

public string Name { get; set; }

[EmailAddress]

public string Email { get; set; }

[Range(18, 99)]

public int Age { get; set; }

}

**View Models**

* Used to shape data specifically for the view.
* Example:

public class ProductViewModel

{

public string ProductName { get; set; }

public decimal Price { get; set; }

public string Category { get; set; }

}

**Best Practices**

1. **Use View Models**:
   * Prevent exposing domain models directly to the view.
2. **Keep Models Simple**:
   * Focus on properties and basic validation.
   * Avoid embedding business logic in models.
3. **Validation**:
   * Use data annotations or custom validation.
4. **Automate Mapping**:
   * Use libraries like **AutoMapper** to map between models and view models.
5. **Separation of Concerns**:
   * Keep domain models, view models, and DTOs distinct.

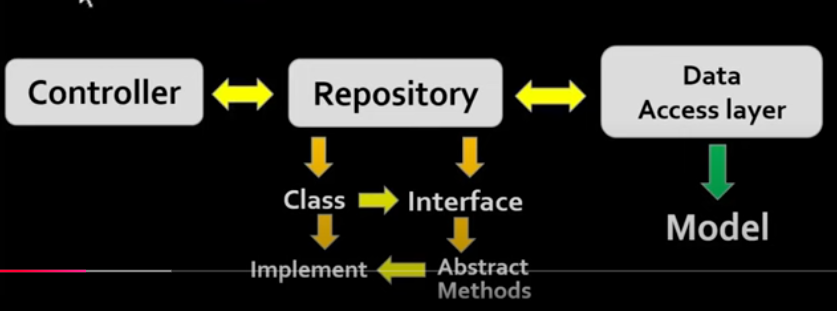
**Key Points**

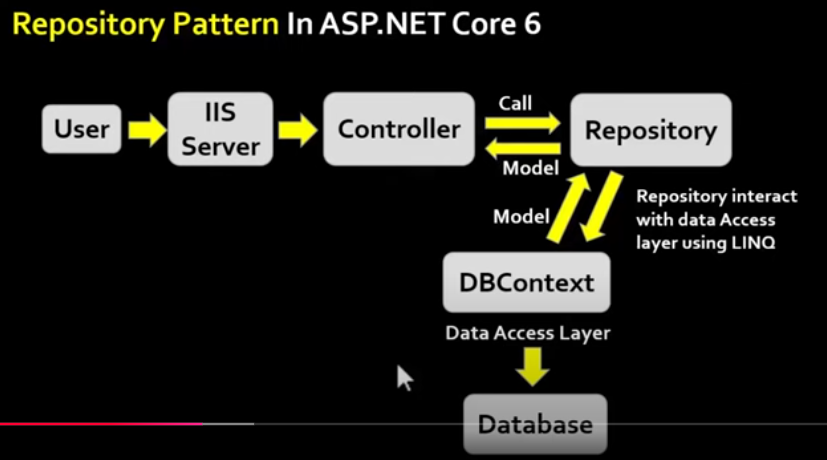
* Models act as a bridge between the controller, view, and database.
* Use appropriate models for specific tasks:
  + Domain Models: Core business entities.
  + View Models: Presentation-layer data.
  + DTOs: Simplified, task-specific objects.

By following best practices and leveraging the flexibility of models, you can maintain a clean and scalable architecture in your ASP.NET Core applications.

**Repository Pattern**

The **Repository Pattern** is a design pattern that separates the data access logic from the business logic of an application. It provides a centralized abstraction for managing data, which promotes maintainability, testability, and separation of concerns.





**Why Use the Repository Pattern?**

1. **Separation of Concerns**:
   * Keeps data access logic separate from business logic.
2. **Testability**:
   * Easier to test business logic by mocking the repository.
3. **Abstraction**:
   * Hides the details of data access (e.g., Entity Framework, raw SQL) from the service layer.
4. **Reusability**:
   * Promotes code reuse by centralizing data access methods.

**Key Components**

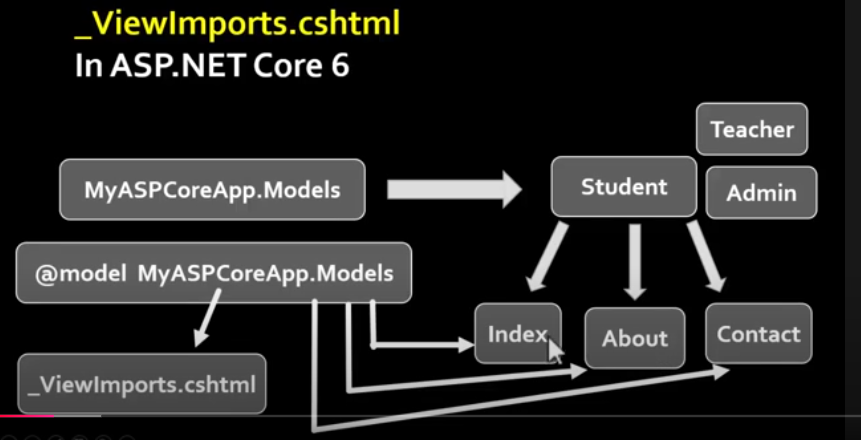
1. **Entity**:
   * Represents the data model (e.g., Product, Order).
2. **Repository Interface**:
   * Defines the contract for data access methods (e.g., GetAll, GetById, Add).
3. **Repository Implementation**:
   * Implements the data access logic, typically using Entity Framework Core.
4. **Service/Controller**:
   * Uses the repository to access and manipulate data.

# How To Install Bootstrap, Tailwind, jQuery Or Other Libraries In ASP.NET Core 6

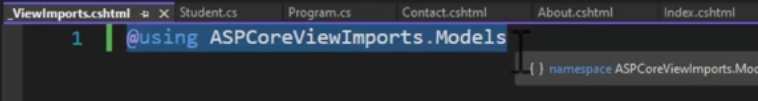


**\_ViewImports.cshtml**

• In ASP.NET Core MVC Application, the \_ViewImports.cshtml file provides a mechanism to include the directives globally for Razor Pages / Views so that we don't have to add the individually in each and every page.



**Declaring Student model globally**



• ViewImports.cshtml file supports the following directives:

* + @addTagHelper
  + @tagHelperPrefix
  + @removeTagHelper
  + @namespace
  + @inject
  + @model
  + @using

# Tag Helpers In ASP.NET Core 6

• Tag helpers are basically special attributes provided by Asp.net Core.

• Tag Helpers enable server-side components to participate in creating and rendering HTML elements in Views.

• Tag helpers are a new feature and similar to HTML helpers (MVC 5), which help us render HTML.

• There are many built-in Tag Helpers for common tasks, such as creating forms, hyperlinks, loading assets etc.

• Tag Helpers are authored in C#, and they target HTML elements based on the element name.

• For example, the built-in LabelTagHelper can target the HTML <label> element when the LabelTagHelper attributes are applied.

Before start working with tag helpers, make sure you have included namespace for tag helpers in your ViewImports file.

• Microsoft.AspNetCore.Mvc.TagHelpers.

Add this line in in your ViewImports file.

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

Example:

<a href="/Home/Contact">Contact Page 1</a>

<br/>

@Html.ActionLink("Contact Page 2", "Contact", "Home") // HTML helpers (MVC 5)

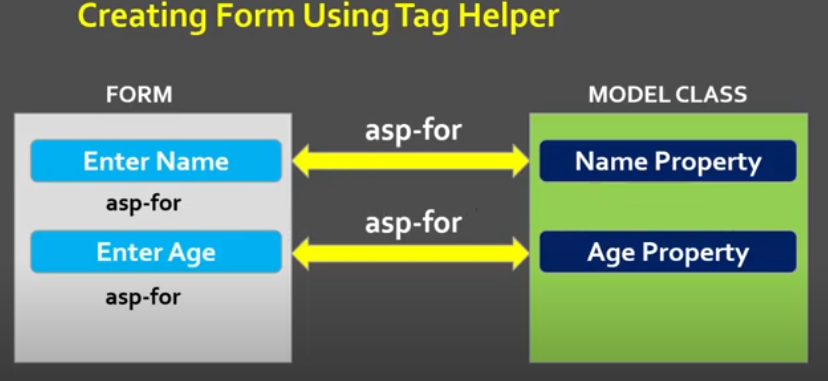
<br/>

<a href="@Url.Action("Contact", "Home")">Contact Page 3</a>// HTML helpers (MVC 5

<br/>

<a asp-controller="Home" asp-action="Contact">Contact Page 4</a> // Tag Helper

**Creating Form Using Tag Helpers In ASP.NET Core 6**



**Asp-For Tag helper**

* The asp-for attribute is arguably one of the most common tag helpers that you'll encounter, and its primary purpose is to handle binding a specific property to the element that it decorates.
* asp-for tag helper is used for an <input>, it sets the name attribute, so that it can be bound to model class.
* e.g. asp-for="Movie.Year" when retrieving data from a POST request.
* asp-for sets the id, name and validation related attributes.

**Basic Form Tag Helper Syntax**

<form asp-controller="ControllerName" asp-action="ActionName" method="post">

<!-- Form inputs go here -->

<input asp-for="Name" class="form-control" />

</form>

* asp-controller: Specifies the controller name.
* asp-action: Specifies the action method.
* method: Specifies the HTTP method (default is post).

Creating Form Using Tag Helper

* The Form Tag Helper.
* The Form Action Tag Helper.
* The Input Tag Helper.
* The Textarea Tag Helper.
* The Label Tag Helper.
* The Validation Tag Helpers.
* The Select Tag Helper

Creating DropDownList, Radio Buttons, TextArea, and Labels in ASP.NET Core 6

**1. DropDownList**

The Select Tag Helper generates a dropdown list.

**Example:**

**Model**:

public class User

{

public string SelectedCountry { get; set; }

public List<SelectListItem> Countries { get; set; }

}

**Controller**:

public IActionResult Index()

{

var model = new User

{

Countries = new List<SelectListItem>

{

new SelectListItem { Value = "1", Text = "India" },

new SelectListItem { Value = "2", Text = "USA" },

new SelectListItem { Value = "3", Text = "UK" }

}

};

return View(model);

}

**View**:

@model User

<form method="post">

<label asp-for="SelectedCountry"></label>

<select asp-for="SelectedCountry" asp-items="Model.Countries" class="form-control"></select>

<button type="submit" class="btn btn-primary">Submit</button>

</form>

**Generated HTML**:

html

Copy code

<select id="SelectedCountry" name="SelectedCountry" class="form-control">

<option value="1">India</option>

<option value="2">USA</option>

<option value="3">UK</option>

</select>

**2. Radio Buttons**

The Input Tag Helper is used to generate radio buttons.

**Example:**

**Model**:

public class GenderModel

{

public string Gender { get; set; }

}

**Controller**:

public IActionResult Index()

{

return View();

}

**View**:

@model GenderModel

<form method="post">

<label>

<input asp-for="Gender" type="radio" value="Male" /> Male

</label>

<label>

<input asp-for="Gender" type="radio" value="Female" /> Female

</label>

<button type="submit" class="btn btn-primary">Submit</button>

</form>

**Generated HTML**:

html

Copy code

<label>

<input id="Gender" name="Gender" type="radio" value="Male" /> Male

</label>

<label>

<input id="Gender" name="Gender" type="radio" value="Female" /> Female

</label>

**3. TextArea**

The TextArea Tag Helper is used to create multi-line text input fields.

**Example:**

**Model**:

public class CommentModel

{

public string Comment { get; set; }

}

**View**:

@model CommentModel

<form method="post">

<label asp-for="Comment"></label>

<textarea asp-for="Comment" class="form-control" rows="5"></textarea>

<button type="submit" class="btn btn-primary">Submit</button>

</form>

**Generated HTML**:

<label for="Comment">Comment</label>

<textarea id="Comment" name="Comment" class="form-control" rows="5"></textarea>

**4. Labels**

The Label Tag Helper generates <label> elements for form fields.

**Example:**

**Model**:

public class User

{

public string Name { get; set; }

}

**View**:

@model User

<form method="post">

<label asp-for="Name"></label>

<input asp-for="Name" class="form-control" />

<button type="submit" class="btn btn-primary">Submit</button>

</form>

**Generated HTML**:

<label for="Name">Name</label>

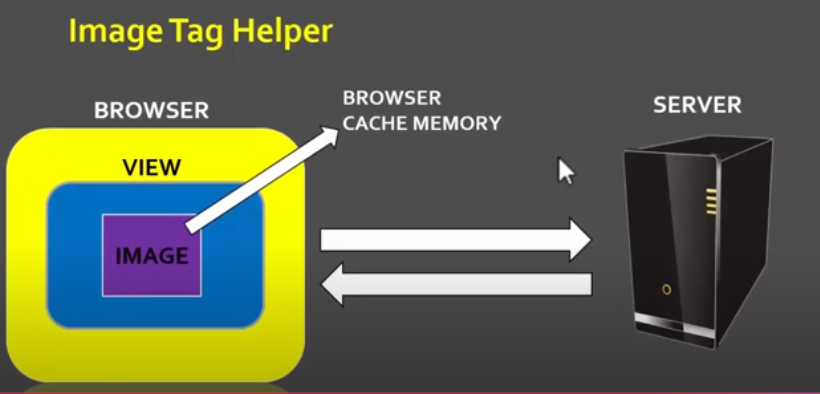
<input id="Name" name="Name" class="form-control" type="text" />

**Image Tag Helper**In ASP.NET Core 6

• In general, when you visit a web page and if that web page contains some image, then most of the modern web browser cache the images for later use.

• In the future, when you revisit to that web page, then the browser loads the images from cache instead of downloading the images from the server.

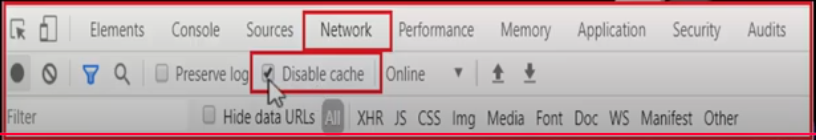
• In most of the caches, this is not an issue as the images are not changes that often for a web page.



How to Disable Browser Cache?

Press F12 key to launch the Browser Developer Tools.

Then click on the "Network" tab and finally, check then "Disable Cache" checkbox as shown in the below image



How to use Image Tag Helper in ASP.NET Core?

• In order to use the Image Tag Helper in ASP.NET Core Application, you need to add the asp-append-version attribute to the <img> tag and need to set the value to true.

• Image tag helper provides cache busting behavior within asp.net core application.

**Example:**

<img asp-src="images/sample.jpg" asp-append-version="true" alt="Sample Image" />

**Generated Output**:

<img src="/images/sample.jpg?v=5f6d6c7a9b" alt="Sample Image" />

v=5f6d6c7a9b: The version string is based on the file's hash, ensuring cache invalidation when the file changes.

Model Binding

• Model binding in ASP.NET Core MVC maps data from HTTP requests to action method parameters.

• The parameters may be simple types such as strings, integers, or floats, or they may be complex types like Student, Employee, Customer, Order etc.

How model binding works

• When MVC receives an HTTP request, it routes it to a specific action method of a controller.

• It determines which action method to run based on what is in the route data, then it binds values from the HTTP request to that action method's parameters.

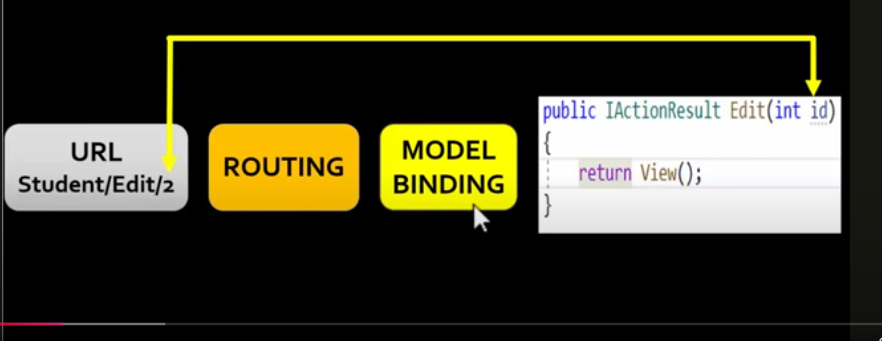
• In addition to route values MVC will bind data from various parts of the request and it does so in a set order. Below is a list of the data sources in the order that model binding looks through them:

1. Form values: These are form values that go in the HTTP request using the POST method.

2. Route values: The set of route values provided by routing.

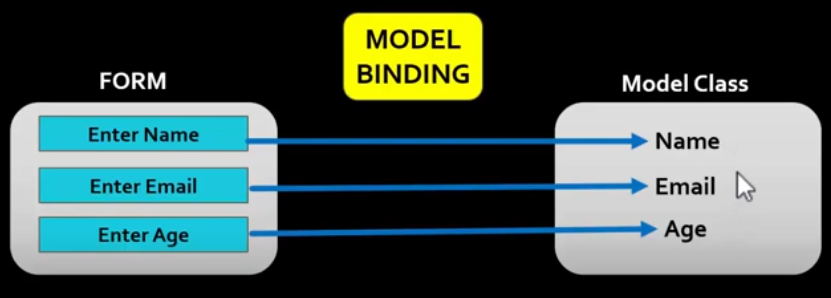
3. Query strings: The query string part of the URI.

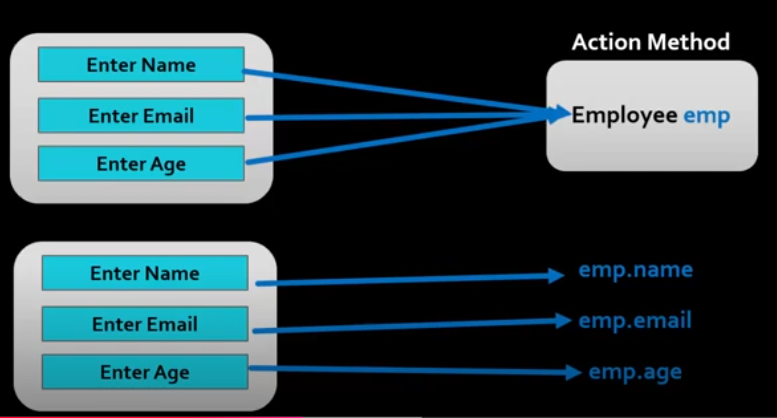
• NOTE: Form values, route data, and query strings are all stored as name-value pairs.



**Model Validation**

* Often the data entered by the user is not valid and cannot be saved into the database.
* The entered data may contain a typo or user may intentionally enter the inappropriate data.
* Hence, we need to validate the user input before storing it in the database.





• The ASP.NET Core gives us Model Validator, which uses the validation attributes to validate the model, which makes our task easier.

• We also take a look at ModelState and how to use it. Finally, we look at the list of Validation attributes.

• The Form Data is posted to Controller action is automatically mapped to the action parameter by the Model Binder. We already looked at how Model Binding works in ASP.NET Core.

• The Model needs to be validated for the correctness.

• These validations can be done at the client side before sending data to the server or at the server side when the data is received from the client.

Explicitly Validating a Model

Once you received the model in the controller, you can validate the model programmatically as shown below:

if (string.IsNullOrEmpty(model.Name))

{

//Validation Failed

//Send the list of errors to client

}

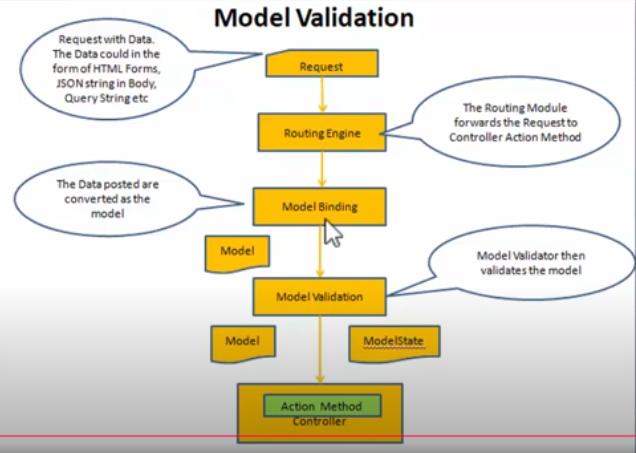
How Model Validation works

• We learnt how Model binding works in ASP.NET Core applications. When the HTTP Request arrives Model binder is invoked before passing the control to controller action method.

• The Model binder not only binds the value to the action parameter but also validates them by using the Model Validator.

• The Model Validator runs after the model binding and runs a series of validations on each property of the model based on the attributes that you set on the model property.

• These attributes are called Validation attributes and cont in the code, which is used by the Model Validator to validate the model.



**ModelState**

**ModelState** in ASP.NET Core is a feature of the **Controller** class that represents the state of model binding and validation. It plays a crucial role in processing and validating incoming data in an application.

**Key Components of ModelState**

1. **Model Binding**:
   * Populates action method parameters or properties of complex types using request data (e.g., form values, query strings, route values, etc.).
2. **Validation**:
   * Ensures that the data provided adheres to the validation rules defined in the model using **Data Annotations** or custom validation.

**Properties of ModelState**

* **ModelState.IsValid**:
  + Returns true if the model has no validation errors; otherwise, false.
* **ModelState.Values**:
  + Contains all the values submitted to the action.
* **ModelState.Errors**:
  + Stores validation error messages.



All these validation attributes are located in

System.ComponentModel.DataAnnotations;

**1. Required Attribute**

The [Required] attribute ensures that a property has a value. If the property is null, empty, or whitespace, validation fails.

[Required(ErrorMessage = "This field is required.")]

**2. StringLength Attribute**

The [StringLength] attribute specifies the maximum and/or minimum length of a string property.

**Syntax**

[StringLength(maximumLength, MinimumLength = minimumLength, ErrorMessage = "Custom error message")]

**Email and Range Validation in ASP.NET Core 6**

ASP.NET Core 6 provides built-in validation attributes, including **[EmailAddress]** and **[Range]**, to validate specific types of user input. These attributes ensure that input matches a valid email format or falls within a specified numeric range.

**3. EmailAddress Attribute**

The **[EmailAddress]** attribute validates that a property contains a valid email address format.

**Syntax**

[EmailAddress(ErrorMessage = "Invalid email address.")]

4. **Range Attribute**

The **[Range]** attribute ensures that a property's value falls within a specified range.

**Syntax**

[Range(minimumValue, maximumValue, ErrorMessage = "Custom error message.")]

5. **Regular Expression**

The **[RegularExpression]** validation attribute is used in ASP.NET Core to validate input against a specified pattern. This attribute allows developers to enforce custom rules for data input using **regular expressions (regex)**.

**Syntax**

[RegularExpression("regex pattern", ErrorMessage = "Custom error message.")]

**6. [Compare] Attribute**

**Purpose:**

Validates that two properties in a model have the same value.

**Use Case:**

* Confirm password validation.

**Syntax:**

[Compare("OtherProperty", ErrorMessage = "Fields must match.")]

**7. [URL] Attribute**

**Purpose:**

Ensures that a property contains a valid URL format.

**Use Case:**

* Validating website or resource URLs.

**Syntax:**

[Url(ErrorMessage = "Invalid URL format.")]

**8. [MaxLength] Attribute**

**Purpose:**

Specifies the maximum number of characters allowed for a string.

**Use Case:**

* Limiting input size for fields like name, description, or comments.

**Syntax:**

[MaxLength(length, ErrorMessage = "Custom error message.")]

**9. [MinLength] Attribute**

**Purpose:**

Specifies the minimum number of characters required for a string.

**Use Case:**

* Ensuring minimum input length, such as a strong username or comment.

**Syntax:**

[MinLength(length, ErrorMessage = "Custom error message.")]

**10. [Display] Attribute**

**Purpose:**

Specifies the display name for a property (used in labels, validation messages, etc.).

**Use Case:**

* Customizing field labels and validation error messages.

**Syntax:**

[Display(Name = "Custom Name")]

**Validation Tag Helpers**

Validation Tag Helpers simplify the integration of client-side and server-side validation in Razor views by automatically generating the necessary HTML for displaying validation messages and summaries.

**Types of Validation Tag Helpers**

1. **<span asp-validation-for="PropertyName">**:
   * Displays a validation error message for a specific property.
2. **<div asp-validation-summary="All" | "ModelOnly">**:
   * Displays a summary of all validation errors.

**Usage of Validation Tag Helpers**

**1. asp-validation-for**

* **Purpose**: Display validation errors for a specific model property.

**Syntax**:

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

**Example**:

<div>

<label asp-for="Name"></label>

<input asp-for="Name" class="form-control" />

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

</div>

**2. asp-validation-summary**

* **Purpose**: Display all validation errors in a single location.
* **Attributes**:
  + **All**: Shows errors for both model-level and property-level validation.
  + **ModelOnly**: Shows errors for model-level validation only.

**Syntax**:

<div asp-validation-summary="All" class="text-danger"></div>

**Example**:

<form method="post">

<div asp-validation-summary="All" class="text-danger"></div>

<div>

<label asp-for="Name"></label>

<input asp-for="Name" class="form-control" />

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

</div>

<button type="submit" class="btn btn-primary">Submit</button>

</form>

**Entity Framework Core In ASP.NET Core 6**

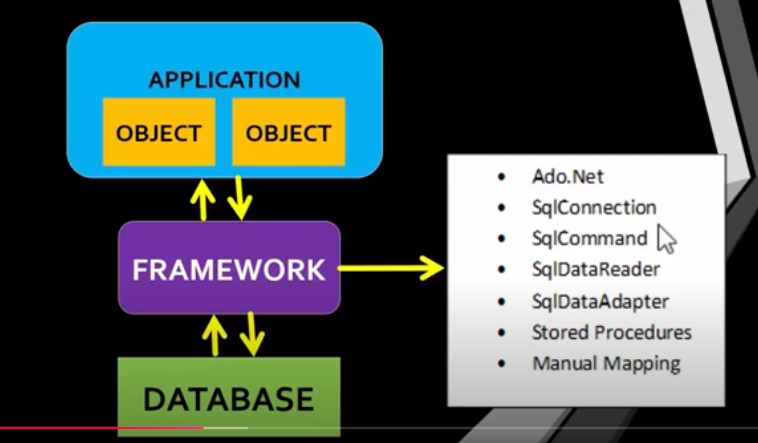
ORM

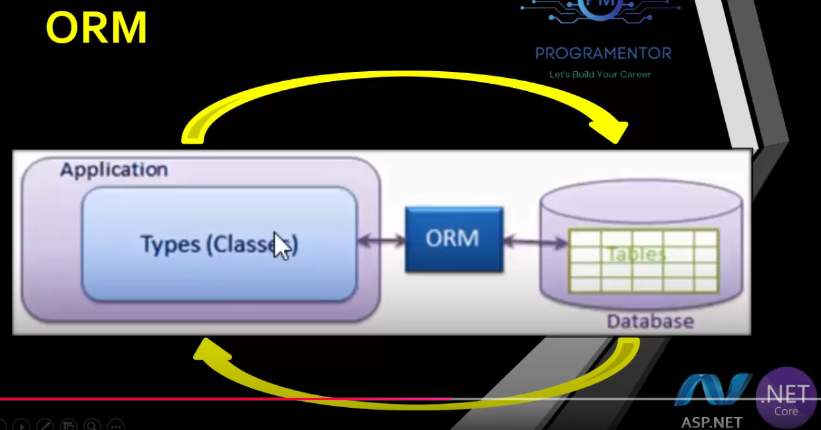
• To address the data access requirements of ASP.NET CORE MVC application, you can use an ORM (OBJECT RELATIONAL MAPPING) framework.

An ORM framework

• Simplifies the process of accessing data from applications.

• ORM is a tool for storing data from domain objects(Model class) to relational database(SQL SERVER).





Entity Framework Core is the new version of Entity Framework after EF 6 but it is redesigned.

• It is open-source, lightweight, extensible and a cross-platform version of Entity Framework.

• Entity Framework Core is a data access technology.

• Entity Framework is an Object/Relational Mapping (ORM) framework.

• It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing & storing the data in the database.

• EF Core is intended to be used with.NET Core applications. However, it can also be used with standard.NET 4.5+ framework based applications.

EF Core supports two development approaches

• 1) Code-First

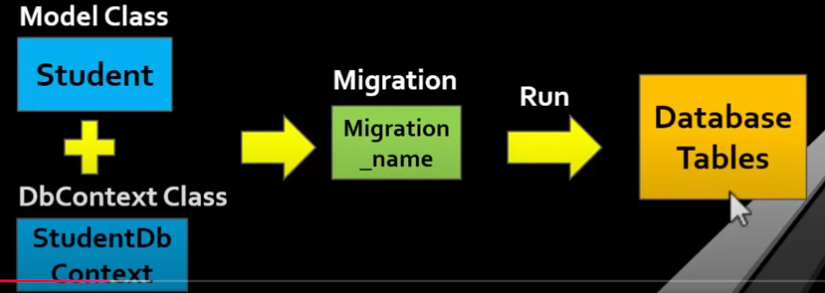
• 2) Database-First.  
  
1) Code-First  
• In the code-first approach the Entity Framework core creates database objects based on model classes that you create to represent application data.



• Is the most common approach implemented in ASP.NET CORE MVC Framework.

• Allows you to develop your application by coding model classes and properties and delegate the process of creating the database objects to the Entity Framework Core.  
• The code-first approach allows you to define your own model by creating custom classes.

• Then, you can create database based on the models.



Steps

1st Step:

• Install 3 Packages in your ASP.NET Core MVC application.

1. Microsoft.EntityFrameworkCore.SqlServer

2. Microsoft.EntityFrameworkCore.Tools They're primarily used to manage Migrations and to scaffold a DbContext.

3. Microsoft.EntityFrameworkCore.Design contains all the design-time logic for Entity Framework Core.

2nd Step:

* Create a Model Class.
* Create a DbContext Class.

Syntax:

using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext

{

public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

public DbSet<Product> Products { get; set; }

}

DbContext

• The DbContext class is an integral part of Entity Framework.

• This is the class that we use in our application code to interact with the underlying database.

• It is this class that manages the database connection and is used to retrieve and save data in the database.

• An instance of DbContext represents a session with the database which can be used to query and save instances of your entities to a database.

• DbContext is a combination of the Unit Of Work and Repository patterns.

• DbContext can be used to define the database context class after creating a model class.

• DbContext co-ordinates with Entity Framework and allows you to query and save the data in the database.

• Uses the DbSet <T> type to define one or more properties where, T represents the type of an object that needs to be stored in the database

DbContextOptions in Entity Framework Core

* For the DbContext class to be able to do any useful work, it needs an instance of the DbContextOptions class.
* The DbContextOptions instance carries configuration information such as the connection string, database provider to use etc.

3rd step:

Add the connection string to appsettings.json

{

"ConnectionStrings": {

"DefaultConnection": "Server=.;Database=MyAppDb;Trusted\_Connection=True;"

}

}

4th Step:

Registering Connection String In Program.cs File.

Example:

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

// Configure EF Core

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();

(or)

var provider builder.Services. BuildServiceProvider();

var config = provider.GetRequiredService<IConfiguration>();

builder. Services.AddDbContext<StudentDBContext> (item => item. UseSqlServer(config.GetConnectionString("DefaultConnection")));

5th step:

Add a migration and Run the migration

Step

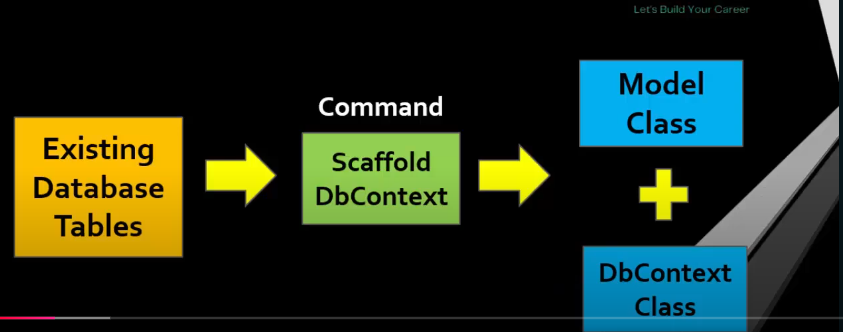
* add-migration “Name”
* Update-database

2. **Database First Approach**

**• In the database-first approach the Entity Framework core creates model classes and properties corresponding to the existing database objects, such as tables and columns.**

**• The database-first approach is applicable in scenario where a database already exists for the application.**





Steps:

1st Step:

• Install 3 Packages in your ASP.NET Core MVC application

2nd Step:

Execute a command for Scaffold DbContextss

Scaffold-DbContext "server=ServerName;database DatabaseName;trusted\_connection=true"

Microsoft.EntityFramework Core.SqlServer-OutputDir Models

•Above command will generate model class and DbContext class automatically.

• If we update our database then how we can update our Model and DbContext.

By using this command:

Scaffold-DbContext "server=ServerName;database DatabaseName;trusted\_connection=true"

Microsoft.EntityFramework Core.SqlServer-OutputDir Models -force

3rd Step:

• Move Connection String from DbContext Class To appsettings.json file.

4th Step:

• Registering Connection String In Program.cs File.

**[ValidateAntiForgeryToken] in ASP.NET Core [Important]**

The [ValidateAntiForgeryToken] attribute is used to protect against **Cross-Site Request Forgery (CSRF)** attacks in ASP.NET Core applications. CSRF is a type of attack where unauthorized commands are transmitted from a user that the web application trusts.

**How Does It Work?**

* When you use [ValidateAntiForgeryToken], ASP.NET Core expects every HTTP POST request to include a valid antiforgery token.
* The antiforgery token is usually embedded in forms as a hidden input or in a request header.

### ****Usage****

#### **1. Adding Antiforgery Token in a Form**

In a Razor view, use the @Html.AntiForgeryToken() helper inside the form:

<form asp-action="Submit" method="post">

@Html.AntiForgeryToken()

<div class="form-group">

<label for="Name">Name</label>

<input type="text" id="Name" name="Name" class="form-control" />

</div>

<button type="submit" class="btn btn-primary">Submit</button>

</form>

This generates a hidden input with the antiforgery token:

<input name="\_\_RequestVerificationToken" type="hidden" value="TOKEN\_VALUE" />

**2. Enforcing Validation in the Controller**

Add the [ValidateAntiForgeryToken] attribute to the POST action method:

using Microsoft.AspNetCore.Mvc;

public class HomeController : Controller

{

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Submit(string name)

{

// Handle form submission

return View();

}

}

**How to Enable Antiforgery by Default**

By default, ASP.NET Core includes antiforgery token validation in Razor Pages and MVC:

1. **Razor Pages**: Antiforgery is automatically applied to all form POST requests.
2. **MVC**: Use [AutoValidateAntiforgeryToken] on the controller or globally to enforce antiforgery token validation for all POST methods:

[AutoValidateAntiforgeryToken]

public class MyController : Controller

{

public IActionResult Index() => View();

[HttpPost]

public IActionResult Submit(string data)

{

// Antiforgery token will be validated automatically

return RedirectToAction("Index");

}

}

**Excluding Certain Actions**

If you want to skip CSRF validation for specific actions, you can use the [IgnoreAntiforgeryToken] attribute.

[HttpPost]

[IgnoreAntiforgeryToken]

public IActionResult UnprotectedAction()

{

// This action will not validate antiforgery tokens

return Ok();

}

**Session In ASP.NET Core 6**

Step:1

• Add below line before build,

"builder.Services.AddSession();"

• Add below line after build,

"app.UseSession();"

Step:2

• Create Session Variable.

• HttpContext.Session.SetString("MyKey", "Programentor");

• We can set different type of values in a session.

Step:3s

• Access Session Variable.

HttpContext.Session.GetString("MyKey ");

Step:4

• If you want to access session directly in view, not in action method then you have to use HttpContextAccessor.

Example:

Program.cs

builder.Services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>;

View pages:

@using Microsoft.AspNetCore.Http

@inject Microsoft.AspNetCore.Http. IHttpContextAccessor accessor

<h1>About</h1>

<h3>@accessor.HttpContext.Session.GetString("MyKey")</h3>

What Is Session ?

• It is a state management technique.

• Session state is an ASP.NET Core scenario for storage of user data while the user browses a web app.

• Session state uses a store maintained by the app to persist data across requests from a client.

• The session data is backed by a cache and considered ephemeral data.

• Critical application data should be stored in the user database and cached in session only as a performance optimization.

• The session is specific to the browser, Sessions aren't shared across browsers.

• Session are deleted when the browser session ends.

• Sessions are Server-Side.

• Session is also used to pass data within the ASP.NET Core MVC application and unlike TempData.

• It persist for its expiration time (default time is 20 minutes but it can be increased or decreased).

• When many users access an application simultaneously, then, each of these users will have a different session state.

• Every session has a unique session id.

**Session Timeout**

The IdleTimeout property defines the duration a session remains active without activity. The default timeout is 20 minutes.

builder.Services.AddSession(options => {

options.IdleTimeout = TimeSpan.FromMinutes(30); // Set session timeout

});

**Clear and Remove Session Data**

1. **Remove Specific Key**:

HttpContext.Session.Remove("Username");

1. **Clear All Session Data**:

HttpContext.Session.Clear();

**Summary**

* **Configuration**: Add session middleware and configure timeout and storage.
* **Usage**: Store and retrieve data using HttpContext.Session.
* **Scalability**: Use distributed cache (SQL Server or Redis) for production.
* **Security**: Mark session cookies as HttpOnly and Secure.

Sessions in ASP.NET Core 6 provide a simple way to manage user-specific data between requests, ensuring a smooth and stateful user experience.

**Difference Between SelectList and SelectListItem in ASP.NET Core**

Both SelectList and SelectListItem are used to represent data for dropdown lists in ASP.NET Core, but they serve different purposes and are used in different contexts.

**1. SelectListItem**

SelectListItem represents a single item in a dropdown list. It is used when you manually define a collection of items for your dropdown.

**Key Properties:**

* Text: The text displayed in the dropdown.
* Value: The value associated with the item.
* Selected: A boolean indicating whether the item is selected by default.

**Example Usage:**

csharp

Copy code

var items = new List<SelectListItem>

{

new SelectListItem { Text = "United States", Value = "US" },

new SelectListItem { Text = "India", Value = "IN", Selected = true },

new SelectListItem { Text = "Australia", Value = "AU" }

};

This creates a collection of individual items for a dropdown.

**2. SelectList**

SelectList is a class used to represent an entire list of items in a dropdown. It is more convenient for populating dropdowns when you already have a collection of objects or entities (e.g., from a database or API).

**Key Features:**

* SelectList can automatically map object properties to Value and Text.
* It eliminates the need to manually create SelectListItem objects for each dropdown item.

**Constructors:**

* SelectList(IEnumerable items, string dataValueField, string dataTextField)
  + items: The data source.
  + dataValueField: The property to use for the value of each item.
  + dataTextField: The property to use for the display text.

**Example Usage:**

var countries = new List<Country>

{

new Country { Id = 1, Name = "United States" },

new Country { Id = 2, Name = "India" },

new Country { Id = 3, Name = "Australia" }

};

var selectList = new SelectList(countries, "Id", "Name");

Here, the Id property is used for the value, and the Name property is used for the text.

**When to Use SelectListItem vs. SelectList**

| **Feature** | **SelectListItem** | **SelectList** |
| --- | --- | --- |
| **Use Case** | Manually define a small set of items. | Populate dropdowns from a collection or database. |
| **Complexity** | Simple and straightforward for static or small lists. | Better for dynamic data sources or larger lists. |
| **Data Binding** | You must explicitly create each SelectListItem. | Automatically binds properties from a data source using field mappings. |
| **Dynamic Data** | Not ideal for larger or dynamic data sources, as it requires manual item creation. | Ideal for dynamic or database-driven dropdown lists. |
| **Default Selection** | Set the Selected property of the individual SelectListItem. | Use a constructor overload of SelectList or bind the selected value via the model (asp-for). |
| **Example** | csharp new SelectListItem { Text = "Option1", Value = "1", Selected = true } | csharp new SelectList(countries, "Id", "Name") |

**Combining Both**

In some scenarios, you might use both together. For instance, you can create a list of SelectListItem objects and pass it to a SelectList for additional functionality.

**Example**:

var items = new List<SelectListItem>

{

new SelectListItem { Text = "United States", Value = "US" },

new SelectListItem { Text = "India", Value = "IN" },

new SelectListItem { Text = "Australia", Value = "AU" }

};

var selectList = new SelectList(items, "Value", "Text");

**Conclusion**

* **SelectListItem**: Best for static or small lists where you manually define items.
* **SelectList**: Best for dynamic or large lists where data comes from an external source (e.g., database or API).

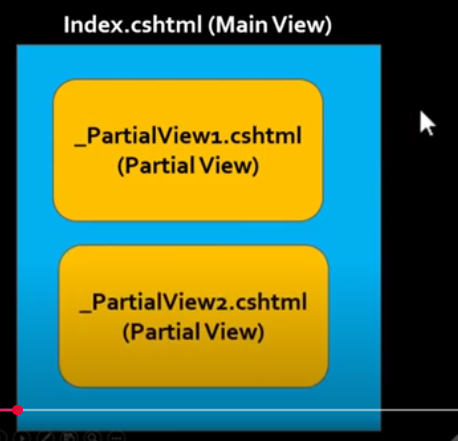
Both are integral to rendering dropdowns in ASP.NET Core, and the choice depends on your specific use case.

**Partial Views**

• Partial view represents a sub-view / nested view / inner view of a main view.

• Partial view allows you to reuse common markups across the different views of the application.

• We can use partial views in different views.



• Partial views cannot be used separately, we have to attach partial in some other view.

• Partial view extension is .cshtml like a view.

• When we have to use some html markup on some pages not all pages then we can use partial view.

View:

<partial name="\_EmployeeNames"/>

**Partial View with Model**

A **Partial View** in ASP.NET Core is a reusable component for rendering a part of a web page. It can be used to display dynamic data, and it can accept a **model** to make it dynamic and reusable.

**Use the Partial View in a Parent View**

You can call the partial view in a parent view using the Html.PartialAsync or Html.RenderPartialAsync helper methods.

@model List<YourNamespace.Models.User>

<h2>User List</h2>

@foreach (var user in Model)

{

@await Html.PartialAsync("\_UserPartial", user)

(Or)

<partial name="\_EmployeeNames" model=”user”/>

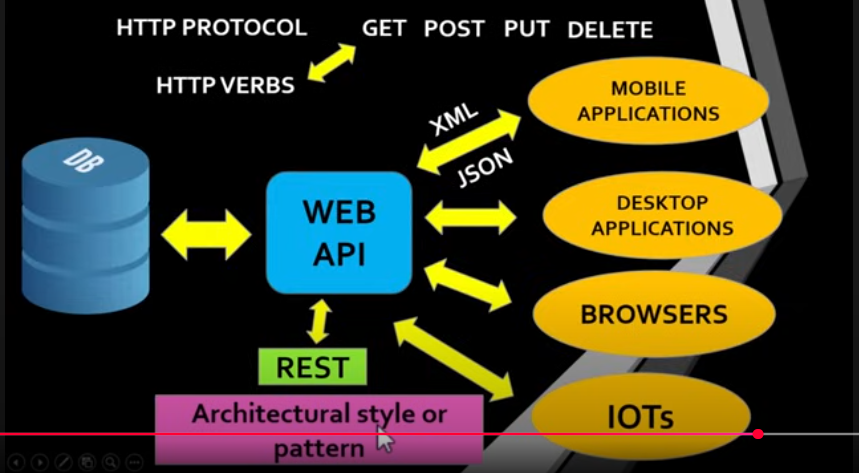
}

**Introduction To ASP.NET Core Web API**

API

• Before we understand what is Web API, let's see what is an API (Application Programing Interface).

• As per Wikipedia's Definition of API: In computer programming, (API) is a set of subroutine definitions, protocols, and tools for building software and applications.



Web API

Web API is an application programming interface (API) that is used to enable communication or interaction with software components with each other.

Web API is the enhanced form of the web application to provide services on different devices like laptop, mobile, and others.

• A Web API (Application Programming Interface) is a set of endpoints that allow different software applications to communicate with each other over the internet.

ASP.NET Core Web API

• ASP.NET Core Web API is a framework developed by Microsoft that allows you to build and create RESTful APIs (Application Programming Interfaces) using the ASP.NET Core platform.

• It's designed to facilitate the development of lightweight, fast, and efficient APIs that can be consumed by various clients, such as web applications, mobile apps, or other services.

• Web API allows users to access a particular resource using HTTP protocol.

• We can build Web API using different technologies like .NET, PHP, JAVA, PYTHON etc

• In.Net Microsoft has created a framework for web API called ASP.NET Core WEB API.

• ASP.NET Core Web API is a framework that makes it easy to build HTTP web services (Restful HTTP services)

* It is provided by Dot Net Framework.
* That web services reaches a broad range of clients, including browser, mobile applications, Desktop application and IOTS.
* It is very similar to ASP.NET MVC since it contains the MVC features.

• Web API is a programming interface/application type that provides communication or interaction between software applications.

• Web API is often used to provide an interface for web sites and client applications to have data access.

• Web APIs can be used to access data from a database and save data back to the database.

RESTFUL Architecture:

• ASP.NET Core Web API follows the principles of Representational State Transfer (REST).

• It uses HTTP methods like GET, POST, PUT, and DELETE interact with resources identified by URLs (Uniform Resource Locators).

• Rest Stands for "Representational state transfer".

• Its introduce in 2000 by Roy Fielding.

• In REST architecture, a REST Server simply provides access to resources and the REST client accesses and presents the resources.

• Here each resource is identified by URIs/ Global IDs.

• A resource is any information that has name like image, entity, document etc.

• REST uses various representations to represent a resource like Text, JSON and XML.

• JSON is now the most popular format being used in Web Services.

RESTFUL Architecture:

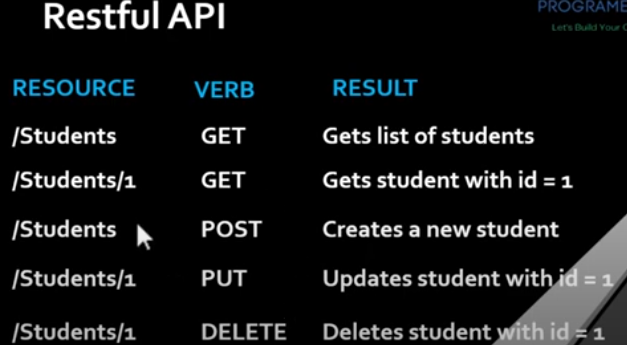
A Restful system consists of:

* client who requests for the resources.
* server who has the resources.

What is Restful Services

REST architectural pattern specifies a set of constraints that a system adhere to.

* Uniform Interface
* Client Server
* Stateless
* Cacheable
* Layered system
* Code on demand



What Is Swagger?

• Swagger is used to describe Restful APIs.

• Swagger allows developer to create interactive and human-readable API documentation.

• Swagger is used to Test the APIs.

What Is ControllerBase?

• The ControllerBase class is a base class for controllers in ASP.NET Core that handles HTTP requests.

• It provides a set of common properties and methods controllers use to handle HTTP requests and generate HTTP responses.

[ApiController] attribute

• The [ApiController] attribute enables a few features including attribute routing requirement, automatic model validation and binding source parameter inference.

**ASP.NET Core Web API CRUD Operations Using Entity Framework Core And SQL Server**

Objectives

First Part:

• Create ASP.NET Core Web API CRUD Operations Using Entity Framework Core & SQL Server. (API Server)

Second Part:

• Consume ASP.NET Core Web API for CRUD Operations In ASP.NET Core MVC Web Application. (Client)

StringContent class

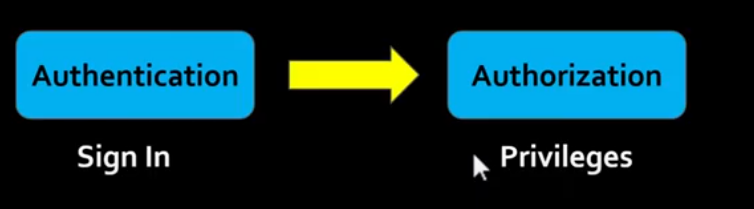
• StringContent class creates a formatted text appropriate for the http server/client communication.

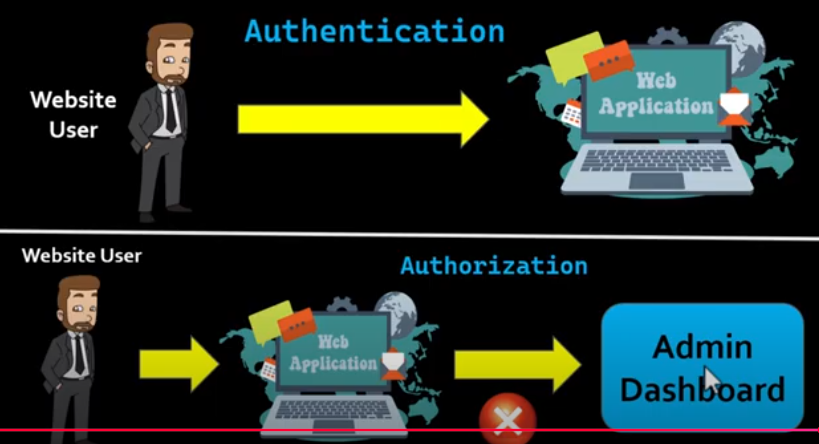
• After a client request, a server will respond with a HttpResponseMessage and that response will need a content, that can be created with the String Content class.

**Authentication And Authorization**

* Authentication is process of checking the identity of the user that accessing our application.
* Authorization is process of validating privileges to access a resource of the application.

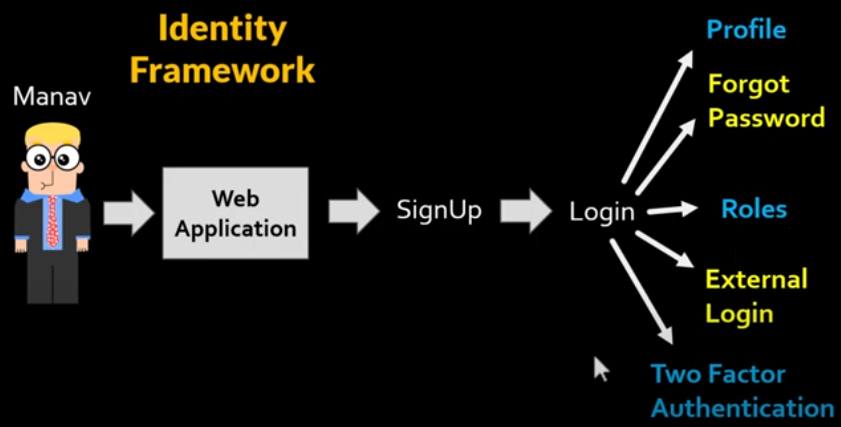
• After successful login to the application (authentication), authorization mechanism checks whether login user has privileges to access the application resource.





**Identity Framework**

* ASP.NET Core Identity is a built-in membership system.
* It is used to Create, Read, Update and Delete User Accounts.
* Account Information.
* Authentication & Authorization
* PassWord Recovery.



• Two factor authentication with SMS.

• It supports External Login Providers like Microsoft, Facebook, Google Gmail etc.

• ASP.Net Core provides identity membership system that enable us to add login functionality to our application.

• It is an API that supports User Interface (UI) Login Functionality.

Manage users, password, profile data, roles, tokens, email confirmation.

• And much more....

**Register / Signup, Login And LogOut Using Identity Framework In ASP.NET Core 6 MVC**

Step 1:

Add Identity To Our Project.

* + Add Layout Page (Master Page)
  + Select Features you want to add in Identity.
  + Create a Data Context Class which is used to communicate with our database.
  + Create a User Class

After Adding Identity there is a folder called "Areas" appears.

Step: 2

• Add Register / Login Links in the Navbar.

• We have to call a "\_LoginPartial.cshtml" partial view in our Layout page by using partial tag.

• "\_LoginPartial.cshtml" partial view is located in Views/Shared folder, and that is added by identity.

Step: 3

• Add Some Properties In Application User class.

• Register these properties OR configure In ApplicationDbContext class.

Step: 4

• Now Add Connection String inside appsettings.json

• Add-Migration.

• Update-Database.

Step: 5

Now Add TextBoxes for FirstName And LastName In Register Page.

AJAX Method

* The ajax() method is used to perform an AJAX (asynchronous HTTP) request.
* It can call any action method of the controller.
* It can pass values to action parameters from the view.
* It can get the response from the action method and display it on the view.

All these actions happen without page reload.

**Image Uploading And Retrieving In ASP.NET Core**

Step 1:

Create database and table.

Step 2:

Install 3 Packages in your ASP.NET Core MVC application.

* Microsoft.EntityFrameworkCore.SqlServer
* Microsoft. EntityFrameworkCore.Tools
* Microsoft. EntityFrameworkCore. Design

STEP 3:

Execute a command for Scaffold DbContext.

• Scaffold-DbContext "server=ServerName;database-DatabaseName;trusted\_connection=true"

Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

• Move Connection String from DbContext Class To appsettings.json file.

• Registering Connection String In Program.cs File.

STEP 4:

• Create a Controller, action method and write code for image uploading in action method and create views.

STEP 5:

• Add validations while uploading image.

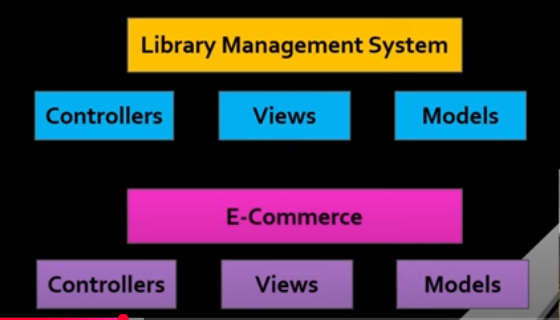
**Area In ASP.NET Core**

* Areas provide a way to partition an ASP.NET Core Web app into smaller functional groups, each with its own set of Razor Pages, controllers, views, and models.
* An area is effectively a structure inside an app.
* Areas are an ASP.NET feature used to organize related functionality into a group as a separate:
* Namespace for routing.
* Folder structure for views and Razor Pages.
  + In an ASP.NET Core web project, logical components like Pages, Model, Controller, and View are kept in different folders.
  + The ASP.NET Core runtime uses naming conventions to create the relationship between these components.
  + For instance, an e-commerce app with multiple business units, such as checkout, billing, and search.
  + Each of these units have their own area to contain views, controllers, Razor Pages, and models.

Consider using Areas in a project when:

• The app is made of multiple high-level functional components that can be logically separated.

• You want to partition the app so that each functional area can be worked on independently.



Steps To Work With Areas

• Adding Folder Structure For Areas.

• Adding Area Specific Routes.

• Decorate Area Attribute On Area's Controller.

• Anchor Tag Helper Attribute. (asp-area)

JSON Web Token

What is a JWT?

A JWT stands for JSON Web Token.

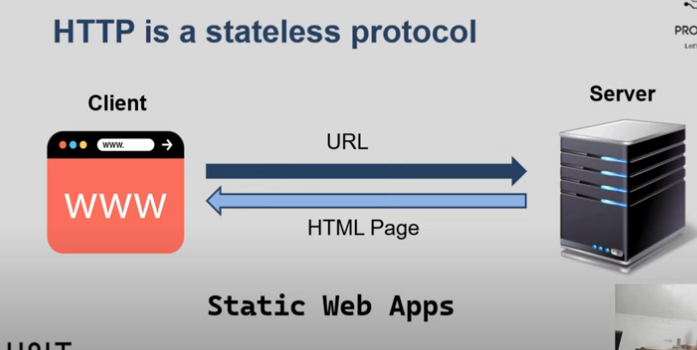
It's a commonly used standard for securely transmitting information between a client (like a web browser or mobile app) and a server.

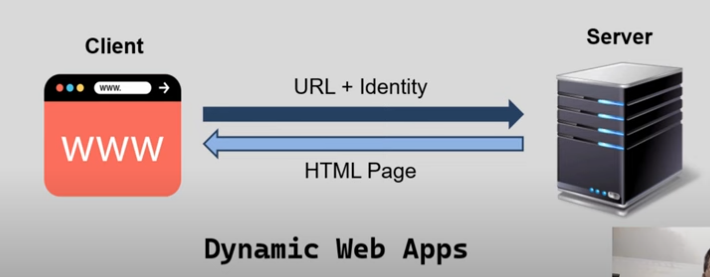
JWTs are used for user authentication and authorization

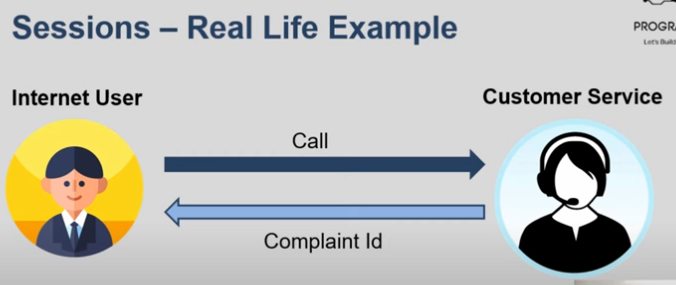
2 Ways

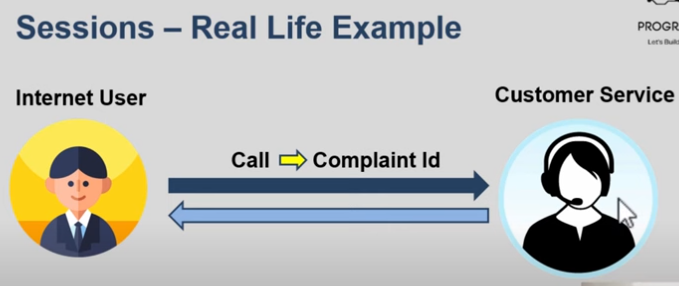
• Sessions

• JSON Web Token (JWT)









Sessions Vs JWT

• Sessions -> Reference Tokens

• JWT → Value Tokens

