## 📘 ASP.NET Core MVC

**Converting Static HTML + Bootstrap into MVC**

---

### 1. Objective of This Stage

- Take an existing **static HTML + Bootstrap website**
- Convert it into a **structured ASP.NET Core MVC application**
- Understand **how MVC consumes HTML assets**
- Learn **where each file goes and why**

This stage focuses on **structure, not business logic**.

---

### 2. Starting Point: Static HTML Project

A typical static website contains:

/index.html

/about.html

/contact.html

/css/bootstrap.min.css

/css/site.css

/js/bootstrap.bundle.min.js

/js/site.js

/images/

All files are **static** and loaded directly by the browser.

---

### 3. MVC Project: Key Concept Shift

In MVC:

- **HTML files become Razor Views**
- **Bootstrap assets move to wwwroot**
- **Navigation becomes controller-based**
- **Pages are rendered via controllers**

Static → Dynamic (server-rendered).

---

**4. MVC Folder Mapping (Static → MVC)**

| Static Website | MVC Project |
|---|---|
| index.html | Views/Home/Index.cshtml |
| about.html | Views/Home/About.cshtml |
| contact.html | Views/Home/Contact.cshtml |
| /css | wwwroot/css |
| /js | wwwroot/js |
| /images | wwwroot/images |

---

**5. wwwroot Folder (Static Assets)**

In MVC, **all static files** must be placed inside wwwroot.

wwwroot/

├── css

│ ├── bootstrap.min.css

│ └── site.css

├── js

│ ├── bootstrap.bundle.min.js

│ └── site.js

└── images

Files outside wwwroot are **not publicly accessible**.

---

**6. Enabling Static Files (Program.cs)**

MVC enables static file serving using middleware:

app.UseStaticFiles();

Without this line:

- CSS
- JS
- Images
  will **not load** in browser.

---

**7. Converting index.html → Razor View**

**Original HTML**

```
<!DOCTYPE html>
<html>
<head>
   <link href="css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
   <h1>Welcome</h1>
</body>
</html>
```

---

**MVC View (Index.cshtml)**

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

<h1>Welcome</h1>
```

Key change:

- <html>, <head>, <body> move to **Layout**
- View contains **page-specific content only**

---

**8. Layout Concept (Very Important)**

MVC uses a **Layout page** as a master template.

Views/Shared/_Layout.cshtml

Layout contains:

- <html>
- <head>
- Navbar
- Footer
- Script references

---

**9. _Layout.cshtml Structure**

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewData["Title"]</title>

    <link rel="stylesheet" href="~/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>

    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
        <a class="navbar-brand" asp-controller="Home" asp-action="Index">MySite</a>
    </nav>

    <div class="container mt-4">
        @RenderBody()
    </div>

    <script src="~/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js"></script>
</body>
</html>
```

---

**10. @RenderBody() (Critical Concept)**

- Placeholder for view content
- Injects the current .cshtml page
- Every view is rendered **inside layout**

---

**11. URL Paths: ~/ in MVC**

~/ means **application root**.

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

This resolves to:

https://domain.com/css/site.css

Avoid hardcoding paths like /wwwroot/css/....

---

**12. Converting Navigation Links**

**Static HTML**

<a href="about.html">About</a>

---

**MVC (Tag Helpers)**

<a asp-controller="Home" asp-action="About">About</a>

Benefits:

- No hardcoded URLs
- Route-safe
- Automatically updates if routes change

---

**13. HomeController (Basic Setup)**

```
public class HomeController : Controller
{
   public IActionResult Index()
   {
      return View();
   }

   public IActionResult About()
   {
      return View();
   }

   public IActionResult Contact()
```

```
    {
        return View();
    }
}
```

Each action maps to a view with the **same name**.

---

**14. View Resolution Rule**

MVC resolves views like this:

Controller: HomeController

Action: About

View path: Views/Home/About.cshtml

If not found → runtime error.

---

**15. Page Title Handling**

In view:

```
@{
    ViewData["Title"] = "Home";
}
```

In layout:

```
<title>@ViewData["Title"]</title>
```

This keeps title **dynamic per page**.

---

**16. Partial Conversion Strategy (Recommended)**

Convert static site gradually:

1. Move CSS/JS/images → wwwroot
2. Create _Layout.cshtml
3. Convert index.html → Index.cshtml
4. Convert navbar links to MVC routes
5. Convert remaining pages one by one

This avoids breaking UI.

---

**17. Common Conversion Mistakes**

- Leaving <html> tags in views
- Wrong CSS paths
- Forgetting UseStaticFiles()
- Hardcoded .html links
- Placing assets outside wwwroot

---

**18. Summary**

- MVC separates **layout** and **page content**
- Static assets live in wwwroot
- HTML pages become Razor views
- Navigation uses controller/action routing
- Layout controls shared UI
- Views render inside layout via @RenderBody()

---

**Exercises for Students**

1. Convert a static Bootstrap homepage into MVC Index.cshtml.
2. Create _Layout.cshtml and move navbar + footer into it.
3. Convert all .html links into MVC Tag Helpers.
4. Add a new page Services using MVC routing.
5. Change Bootstrap theme and verify asset loading.

---

## 📘 Razor Syntax Basics (ASP.NET Core MVC)

---

### 1. What is Razor?

**Razor** is a server-side view engine that allows mixing **HTML** with **C#** code in .cshtml files.

Key characteristics:

- Executes on the **server**
- Generates **HTML output**
- Uses @ to transition from HTML to C#

---

### 2. Razor File Types

| File | Purpose |
|---|---|
| .cshtml | Razor view |
| _Layout.cshtml | Master layout |
| _ViewImports.cshtml | Common imports (namespaces, tag helpers) |
| _ViewStart.cshtml | Default layout assignment |

---

### 3. Razor Transition Symbol (@)

The @ symbol tells Razor: **C# code starts here**.

Examples:

<h2>@DateTime.Now</h2>

@ViewData["Title"]

Rendered output is plain HTML.

---

### 4. Razor Code Blocks

#### 4.1 Inline Expression

Total: @total

Used for simple values.

---

#### 4.2 Code Block

@{

```
    var year = DateTime.Now.Year;
    var name = "MVC";
}
```

- No HTML output
- Used for variable declarations and logic

---

**5. Razor Comments**

```
@* This is a Razor comment *@
```

- Removed completely from output HTML
- Not visible in browser source

---

**6. Control Structures in Razor**

**6.1 if Statement**

```
@if (isLoggedIn)
{
    <p>Welcome back</p>
}
else
{
    <p>Please login</p>
}
```

HTML is written **inside** C# blocks.

---

**6.2 for Loop**

```
@for (int i = 1; i <= 5; i++)
{
    <li>Item @i</li>
}
```

---

**6.3 foreach Loop**

```
@foreach (var name in names)
```

```
{
   <p>@name</p>
}
```

---

**7. Razor and HTML Mixing Rules**

Correct:

```
@if (true)
{
   <h3>Hello</h3>
}
```

Incorrect:

```
@if (true)
   <h3>Hello</h3>  <!-- ❌ -->
```

HTML must be inside { } blocks.

---

**8. Escaping @ Symbol**

To display @ in HTML:

```
<p>Email: support@@company.com</p>
```

or

```
<p>Email: support&#64;company.com</p>
```

---

**9. ViewData, ViewBag (Intro Level)**

**ViewData (Dictionary)**

```
ViewData["Title"] = "Home";
```

Usage in view:

```
<h2>@ViewData["Title"]</h2>
```

---

**ViewBag (Dynamic)**

```
ViewBag.Message = "Welcome";
```

Usage:

```
<p>@ViewBag.Message</p>
```

Both are **temporary** and scoped to the current request.

---

**10. Layout Interaction in Razor**

In view:

```
@{
   ViewData["Title"] = "About";
   Layout = "_Layout";
}
```

In layout:

```
<title>@ViewData["Title"]</title>
@RenderBody()
```

---

**11. _ViewStart.cshtml**

Purpose: set default layout for all views.

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

Location:

Views/_ViewStart.cshtml

Individual views can override it if needed.

---

**12. _ViewImports.cshtml**

Purpose:

- Import namespaces
- Enable tag helpers

Typical content:

```
@using MyMvcApp
@using MyMvcApp.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Applies to all views in the folder tree.

---

**13. HTML Encoding (Security)**

Razor **HTML-encodes output by default**.

@userInput

Prevents XSS attacks.

To output raw HTML (use carefully):

@Html.Raw(htmlContent)

---

**14. Common Razor Errors**

| Error | Cause |
|---|---|
| Unexpected { | HTML not wrapped properly |
| CSHTML not rendering | Missing layout / wrong path |
| @ showing in output | Escaping mistake |
| Layout not applied | _ViewStart.cshtml missing |

---

**15. Best Practices**

- Keep logic minimal in views
- Use views for **presentation only**
- Avoid database or service calls in views
- Prefer strongly typed views (next topic)
- Use partial views for reusable UI

---

**16. Summary**

- Razor mixes HTML and C#
- @ marks C# expressions
- Code blocks use @{ }
- Control structures render HTML
- Layouts control shared UI
- _ViewStart and _ViewImports reduce repetition
- Output is HTML-encoded by default

---

**Exercises for Students**

1. Display current date and time using Razor.
2. Use if to show different messages based on a boolean.
3. Render a list of names using foreach.
4. Set page title using ViewData.
5. Create _ViewStart.cshtml and apply layout globally.

---

# 📘 Controllers & IActionResult (ASP.NET Core MVC)

---

### 1. What is a Controller?

A **Controller** is a C# class responsible for:

- Receiving HTTP requests
- Executing application logic (or delegating it)
- Returning a response (View, JSON, Redirect, Status code)

Controllers act as the **entry point** for MVC requests.

---

### 2. Controller Naming & Location

**Rules:**

- Class name must end with Controller
- Located inside the Controllers folder

Example:

```
public class HomeController : Controller
{
}
```

URL mapping uses the class name **without** the Controller suffix.

---

### 3. Controller Base Classes

#### 3.1 Controller

Used for **view-based MVC**.

```
public class HomeController : Controller
{
}
```

Provides:

- View()
- RedirectToAction()
- ViewData, ViewBag, TempData
- ModelState helpers

---

### 3.2 ControllerBase

Used for **API-only controllers** (no views).

```
public class StudentsController : ControllerBase
{
}
```

Provides:

- Ok(), NotFound(), BadRequest()
- No view support

---

### 4. Action Methods

An **action method**:

- Is a public method in a controller
- Handles a request
- Returns a result to the framework

Example:

```
public IActionResult Index()
{
   return View();
}
```

---

### 5. IActionResult (Core Concept)

IActionResult represents **any HTTP response** a controller can return.

It allows a single action to return:

- HTML view
- JSON data
- Redirect
- HTTP status codes

---

### 6. Common IActionResult Types

#### 6.1 ViewResult

Returns an HTML view.

return View();

or

return View("About");

---

**6.2 ContentResult**

Returns plain text or simple content.

return Content("Hello MVC");

---

**6.3 JsonResult**

Returns JSON data.

return Json(new { Name = "Ravi", Age = 20 });

---

**6.4 RedirectResult**

Redirects to a URL.

return Redirect("/Home/Index");

---

**6.5 RedirectToActionResult**

Redirects to another action.

return RedirectToAction("Index");

or

return RedirectToAction("Details", new { id = 5 });

---

**6.6 Status Code Results**

return NotFound();

return BadRequest();

return Unauthorized();

return Ok();

---

**7. IActionResult vs Specific Return Types**

**Using IActionResult (Flexible)**

```
public IActionResult Details(int id)
{
   if (id <= 0)
      return BadRequest();

   return View();
}
```

**Using Specific Type (Strict)**

```
public ViewResult Index()
{
   return View();
}
```

**Preferred:** IActionResult for flexibility.

---

**8. View Selection Rules**

**Default View Resolution**

```
public IActionResult About()
{
   return View();
}
```

MVC looks for:

Views/Home/About.cshtml

---

**Explicit View Name**

```
return View("Contact");
```

Looks for:

Views/Home/Contact.cshtml

---

**9. Passing Data to Views (Intro)**

**ViewData**

```
ViewData["Message"] = "Welcome";
```

return View();

---

**ViewBag**

ViewBag.Message = "Welcome";

return View();

(Both are temporary and request-scoped.)

---

**10. HTTP Verb Attributes**

Controllers support **HTTP method mapping**.

```
[HttpGet]
public IActionResult Index()
{
    return View();
}
[HttpPost]
public IActionResult Save()
{
    return RedirectToAction("Index");
}
```

---

**11. Route Attributes (Controller Level)**

```
[Route("students")]
public class StudentsController : Controller
{
    [Route("list")]
    public IActionResult List()
    {
        return View();
    }
}
```

URL:

/students/list

---

**12. Action Parameters & Model Binding (Preview)**

public IActionResult Details(int id)

{

return View();

}

URL:

/Home/Details/5

id is automatically bound from the route.

(Deep model binding is covered in the next topic.)

---

**13. TempData (One-Request Data)**

Used for redirects.

TempData["Success"] = "Saved successfully";

return RedirectToAction("Index");

Accessible on the next request only.

---

**14. Controller Lifecycle (High Level)**

Request

↓

Routing

↓

Controller instantiated

↓

Action method executed

↓

Result returned

↓

Response sent

Controller instances are **created per request**.

---

**15. Common Controller Mistakes**

- Business logic inside controller
- Database calls directly in controller
- Returning views from ControllerBase
- Hardcoded URLs in redirects
- Too much logic in actions

---

**16. Best Practices**

- Keep controllers thin
- Delegate logic to services
- Use IActionResult
- Prefer RedirectToAction() over Redirect()
- One responsibility per action

---

**17. Summary**

- Controllers handle requests
- Actions are public methods
- IActionResult represents responses
- MVC supports views, JSON, redirects, status codes
- Routing maps URLs to controllers/actions
- Controllers coordinate, not compute

---

**Exercises for Students**

1. Create a controller with three actions: Index, About, Contact.
2. Return a view from one action and JSON from another.
3. Use RedirectToAction() after a POST.
4. Return NotFound() when an id is invalid.
5. Map a custom route using attributes.

---

## 📘 Strongly Typed Views (ASP.NET Core MVC)

---

### 1. What is a Strongly Typed View?

A **strongly typed view** is a Razor view that is **bound to a specific C# model type** using the @model directive.

This allows:

- Compile-time checking
- IntelliSense support
- Clean and safe data access
- Elimination of magic strings (ViewBag, ViewData)

---

### 2. Why Strongly Typed Views Are Important

Using strongly typed views:

- Reduces runtime errors
- Improves maintainability
- Makes refactoring safe
- Clearly defines what data a view expects

They are the **recommended approach** for production MVC applications.

---

### 3. Basic Syntax: @model

At the **top of a Razor view**:

@model Student

or with namespace:

@model MyMvcApp.Models.Student

This declares that the view expects a Student object.

---

### 4. Example Model

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
```

```
    public int Age { get; set; }
}
```

---

**5. Passing Model from Controller to View**

**Controller Action**

```
public IActionResult Details()
{
    var student = new Student
    {
        Id = 1,
        Name = "Ravi",
        Age = 20
    };

    return View(student);
}
```

The object passed to View() becomes the **Model** for the view.

---

**6. Using Model in View**

**Details.cshtml**

```
@model Student

<h2>Student Details</h2>

<p>Id: @Model.Id</p>
<p>Name: @Model.Name</p>
<p>Age: @Model.Age</p>
```

Key rules:

- Model refers to the object passed from controller
- Property names are checked at compile time

---

**7. Strongly Typed View vs ViewBag/ViewData**

| Feature | Strongly Typed View | ViewBag / ViewData |
|---|---|---|
| Compile-time safety | Yes | No |
| IntelliSense | Yes | No |
| Refactoring safety | High | Low |
| Readability | High | Medium |
| Recommended | Yes | No (for main data) |

---

**8. Strongly Typed Views with Collections**

**Controller**

```
public IActionResult List()
{
   var students = new List<Student>
   {
      new Student { Id = 1, Name = "Ravi", Age = 20 },
      new Student { Id = 2, Name = "Amit", Age = 22 }
   };

   return View(students);
}
```

---

**View (List.cshtml)**

```
@model List<Student>

<h2>Students</h2>

<ul>
@foreach (var s in Model)
{
   <li>@s.Name (@s.Age)</li>
```

}

</ul>

---

**9. Using IEnumerable<T> (Preferred)**

Instead of List<T>:

@model IEnumerable<Student>

Reason:

- More flexible
- Supports any collection type
- Follows abstraction principles

---

**10. Strongly Typed Views with Layout**

No special configuration required.
The model works seamlessly with _Layout.cshtml.

---

**11. Null Safety in Views**

Avoid null reference issues:

@if (Model != null)

{

@Model.Name

}

Or ensure controller always passes a valid model.

---

**12. ViewModels (Concept Introduction)**

Sometimes a view needs **more than one model**.

Instead of using multiple ViewBags, create a **ViewModel**.

Example:

public class StudentDetailsViewModel

{

public Student Student { get; set; } = new();

public string CourseName { get; set; } = "";

}

View:

@model StudentDetailsViewModel

This keeps views strongly typed and clean.

---

**13. Common Mistakes**

- Forgetting @model directive
- Passing wrong object type to View()
- Using ViewBag for main data
- Using database entities directly without ViewModels
- Changing model but not updating view

---

**14. Best Practices**

- Always prefer strongly typed views
- Use IEnumerable<T> for lists
- Use ViewModels for complex views
- Keep views focused on presentation only
- Avoid business logic in views

---

**15. Summary**

- Strongly typed views bind Razor views to C# models
- @model defines the expected type
- Model provides compile-time safe access
- Collections are supported naturally
- ViewModels enable complex data scenarios
- Strongly typed views are the MVC standard

---

**Exercises for Students**

1. Create a strongly typed view to display a single Student.
2. Create a strongly typed view to list multiple students.
3. Convert a ViewBag-based view into a strongly typed view.
4. Create a ViewModel combining Student and an extra field.
5. Modify a model property and observe compile-time errors in view.

---