
📘 Partial Views (ASP.NET Core MVC)

---

1. What is a Partial View?

A **Partial View** is a reusable Razor view that represents **a fragment of UI**, not a complete page.

It is used to:

- Avoid repeating markup
- Break large views into smaller components
- Share UI blocks across multiple pages

Partial views **do not contain <html>, <head>, or <body>**.

---

2. Why Partial Views Are Needed

Without partial views:

- Same HTML is duplicated across views
- Changes must be made in multiple places
- Layout files become too large

Partial views promote:

- Reusability
- Clean separation of UI concerns
- Easier maintenance

---

3. Common Use Cases

Typical UI fragments implemented as partial views:

- Header / Navbar
- Footer
- Sidebar
- Login status panel
- Alerts / Messages
- Reusable tables or cards

---

4. Naming Convention

Partial view files are usually prefixed with an underscore (_):

_Header.cshtml

_Footer.cshtml

_Sidebar.cshtml

The underscore indicates:

- This view is not a full page
- It is meant to be included in other views

---

5. Location of Partial Views

Partial views can be placed in:

5.1 Shared Location (Most Common)

Views/Shared/_Header.cshtml

Views/Shared/_Footer.cshtml

Accessible from **any view**.

5.2 Controller-Specific Location

Views/Home/_HomeBanner.cshtml

Accessible only within the same controller's views (by convention).

---

6. Creating a Simple Partial View

Example: _Footer.cshtml

```
<footer class="text-center mt-4">
    <hr />
    <p>&copy; 2025 - My MVC App</p>
</footer>
```

No layout, no HTML wrapper—only the fragment.

---

7. Rendering a Partial View

7.1 Using <partial> Tag Helper (Recommended)

```
<partial name="_Footer" />
```

Advantages:

- Simple syntax
- Async by default
- Clean and readable

7.2 Using Html.Partial()

@Html.Partial("_Footer")

Returns HTML immediately (synchronous).

7.3 Using Html.PartialAsync()

@await Html.PartialAsync("_Footer")

Preferred if the partial is heavy or data-driven.

---

8. Using Partial Views Inside Layout

Partial views are commonly used in _Layout.cshtml.

```
<body>
   <partial name="_Header" />

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

   <partial name="_Footer" />
</body>
```

This keeps layout clean and modular.

---

9. Partial Views with Strongly Typed Models

A partial view can be **strongly typed**.

Partial View: _StudentCard.cshtml

```
@model Student

<div class="card mb-2">
```

```
  <div class="card-body">
    <h5>@Model.Name</h5>
    <p>Age: @Model.Age</p>
  </div>
</div>
```

Parent View

```
@model IEnumerable<Student>

@foreach (var s in Model)
{
  <partial name="_StudentCard" model="s" />
}
```

Each iteration passes one Student to the partial.

---

10. Partial View vs Layout

Feature	Layout	Partial View
Purpose	Page skeleton	UI fragment
Uses @RenderBody()	Yes	No
Reusable	Yes	Yes
Full HTML	Yes	No
Typical usage	Master page	Navbar, footer, widgets

---

11. Partial View vs ViewComponent (Conceptual)

Partial View	ViewComponent
Simple UI reuse	Complex reusable UI
No C# logic	Has C# class
Data passed from view	Fetches its own data
Lightweight	More powerful

(ViewComponents are covered later.)

---

12. Passing Additional Data to Partial Views

Using anonymous object (limited)

<partial name="_Info" model="new Info { Message = 'Hello' }" />

Preferred: pass a proper model

Create a small ViewModel for clarity.

---

13. Common Mistakes

- Putting <html> or <body> in partial views
- Accessing ViewBag heavily inside partials
- Hardcoding data inside partials
- Naming conflicts without underscore
- Placing partials outside Views folder

---

14. Best Practices

- Use partial views for reusable UI only
- Keep partials small and focused
- Prefer strongly typed partial views
- Store shared partials in Views/Shared
- Avoid business logic in partials

---

15. Summary

- Partial views represent reusable UI fragments
- They reduce duplication and improve maintainability
- Rendered using <partial> or Html.Partial
- Can be strongly typed
- Widely used in layouts and list rendering
- Essential for clean MVC UI architecture

Exercises for Students

1. Create a _Header.cshtml partial and include it in _Layout.cshtml.
2. Create a _Footer.cshtml partial with copyright text.
3. Create a strongly typed partial view to display a student card.
4. Render the student card partial inside a loop.
5. Move repeated HTML from a view into a partial.

Below are **clean, structured notes** for **Topic 5: Model Binding in MVC (Deep)**, continuing the MVC sequence after Partial Views.

📘 Model Binding in ASP.NET Core MVC (Deep)

---

1. What is Model Binding?

Model Binding is the mechanism by which ASP.NET Core MVC **automatically maps incoming request data** to action method parameters and model properties.

It converts:

- Route values
- Query strings
- Form fields
- Headers
- Request body (JSON)

into **C# parameters or objects**.

---

2. Why Model Binding Matters

Model binding:

- Eliminates manual parsing of request data
- Supports complex object graphs
- Integrates with validation
- Enables clean controller actions

It is a **core feature** of MVC and underpins form handling and APIs.

---

3. Data Sources Used by Model Binding

MVC checks data sources in a **defined order**:

1. Route values
2. Query string
3. Form data
4. Request body
5. Headers

Example URL:

/Students/Details/5?show=true

Bindings:

- id → route
- show → query string

---

4. Simple Parameter Binding

Action Method

```
public IActionResult Details(int id)
{
   return View();
}
```

URL

/Students/Details/10

Result:

- id = 10

---

5. Binding from Query String

```
public IActionResult Search(string name, int age)
{
   return View();
}
```

URL:

/Students/Search?name=Ravi&age=20

Model binding maps:

- name = "Ravi"

- age = 20

---

6. Binding to a Model (Complex Type)

Model

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

Action Method

```
[HttpPost]
public IActionResult Create(Student student)
{
    return View(student);
}
```

Form fields:

```
<input name="Id" />
<input name="Name" />
<input name="Age" />
```

MVC binds inputs to Student automatically.

---

7. Model Binding with Forms (POST)

View

```
<form asp-action="Create" method="post">
    <input asp-for="Name" />
    <input asp-for="Age" />
    <button type="submit">Save</button>
</form>
```

Controller

```
[HttpPost]
public IActionResult Create(Student student)
{
   return RedirectToAction("Index");
}
```

Binding works because:

- Input name attributes match property names

---

8. Binding Using Attributes

[FromRoute]

```
public IActionResult Details([FromRoute] int id)
```

[FromQuery]

```
public IActionResult Search([FromQuery] string keyword)
```

[FromForm]

```
[HttpPost]
public IActionResult Save([FromForm] Student student)
```

[FromBody] (API-style)

```
[HttpPost]
public IActionResult Save([FromBody] Student student)
```

Used mainly for JSON requests.

---

9. Binding Collections

Query String Collection

URL:

```
/Tags?items=a&items=b&items=c
```

Action:

```
public IActionResult Tags(List<string> items)
{
```

```
    return View(items);
}
```

Form Collection

```
<input name="Subjects[0]" />
<input name="Subjects[1]" />
public IActionResult Save(List<string> subjects)
```

---

10. Nested Model Binding

Model

```
public class Student
{
    public string Name { get; set; } = "";
    public Address Address { get; set; } = new();
}

public class Address
{
    public string City { get; set; } = "";
    public string Pincode { get; set; } = "";
}
```

Input Names

```
<input name="Name" />
<input name="Address.City" />
<input name="Address.Pincode" />
```

MVC binds nested objects correctly.

---

11. ModelState

ModelState tracks:

- Binding success
- Validation errors

```
if (!ModelState.IsValid)
{
   return View(student);
}
```

Used heavily with validation (next topic).

---

12. Optional Parameters & Defaults

```
public IActionResult Index(int page = 1)
```

If page not provided, default value is used.

---

13. Binding Precedence Rules

If same parameter exists in multiple places:

Priority order:

1. Route
2. Query
3. Form
4. Body

Route values override query string values.

---

14. Common Model Binding Failures

Issue	Cause
Null model	Input names mismatch
Zero values	Type conversion failed
Missing data	Field not submitted
Collection empty	Incorrect index naming
Nested object null	Missing parent prefix

---

15. Best Practices

- Use strongly typed models
- Match input name with property names

- Prefer ViewModels for forms
- Avoid [FromBody] in view-based MVC
- Always validate ModelState

---

16. Summary

- Model binding maps request data to C# objects
- Supports primitives, objects, collections, nested models
- Uses route, query, form, and body data
- Integrates tightly with validation
- Central to MVC form handling

Exercises for Students

1. Bind query string parameters to an action method.
2. Create a form and bind it to a model.
3. Bind a list of values from query string.
4. Create a nested model and bind it using a form.
5. Intentionally break input names and observe ModelState errors.
