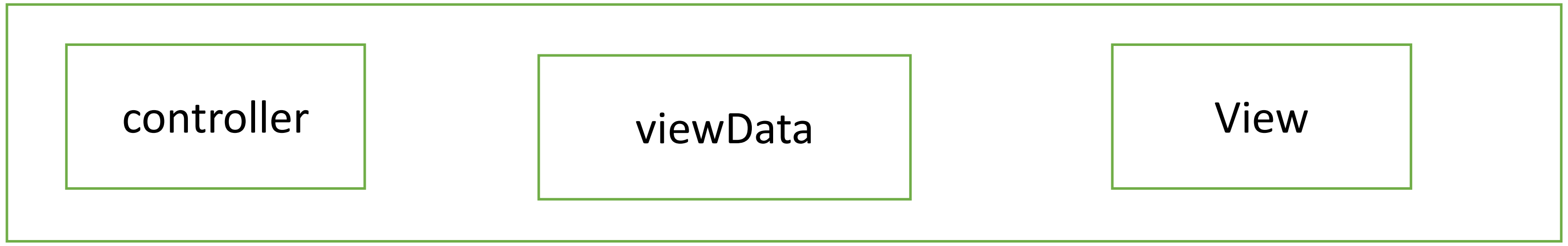
# Asp.NET MVC Core

Trainer : Priyanka R Rangole

Email: priyanka.rangole@sunbeaminfo.com

## ViewData vs ViewBag vs TempData vs Session: An Overview

In **ASP.NET MVC** there are three ways - ViewData, ViewBag, and TempData to pass data from the controller to view and in the next request. Like WebForm, you can also use Session to persist data during a user session. Now the question is when to use ViewData, VieBag, TempData, and Session. Each of them has its importance



ViewData is a dictionary object that is derived from the ViewDataDictionary class.

```
public ViewDataDictionary ViewData { get; set; }
```

1. ViewData is a property of the ControllerBase class.
2. ViewData is used to pass data from the controller to the corresponding view.
3. Its life lies only during the current request.
4. If redirection occurs then its value becomes null.
5. It requires typecasting for getting data and checking for null values to avoid errors.

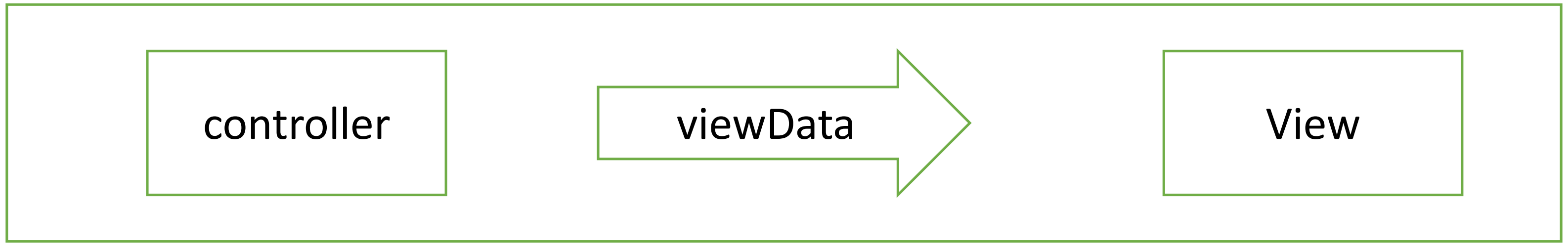
## ViewData Example

```
//Controller Code
public ActionResult Index()
{
      List Employee= new List();
      Employee.Add("Komal");
      Employee.Add("Vishal");
      Employee.Add("Rahul");

      ViewData["Employee"] = Employee;
      return View();
}
//page code

    <% foreach (var employee in ViewData["Employee"] as List)
        { %>
    <%: employee%>

    <% } %>
```



1.ViewBag is a dynamic property that takes advantage of the new dynamic features in C# 4.0.
2.It is a wrapper around the ViewData and is also used to pass data from the controller to the corresponding view.

```
public Object ViewBag { get; }
```

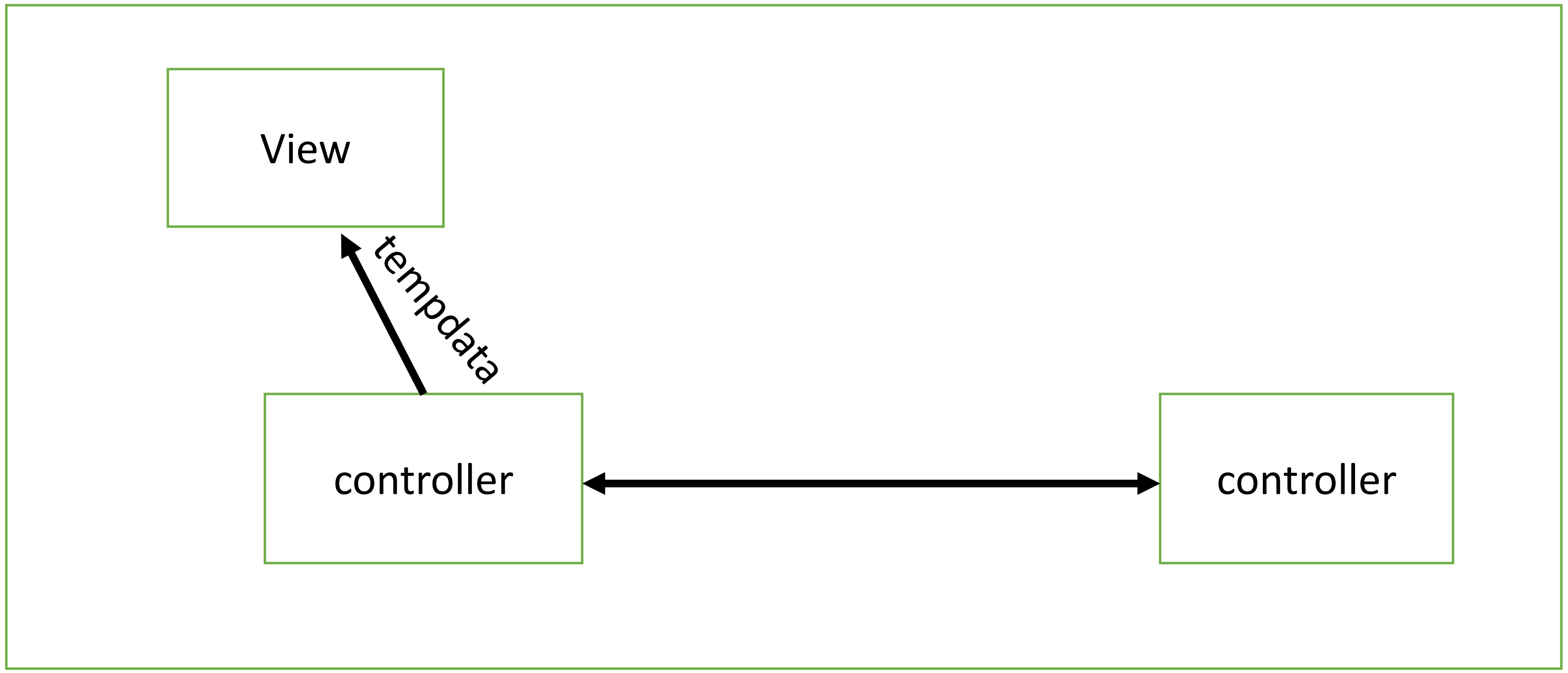
1.ViewBag is a property of the ControllerBase class.
2.Its life also lies only during the current request.
3.If redirection occurs then its value becomes null.
4.It doesn’t require typecasting for getting data.

```
//Controller Code
public ActionResult Index()
{
     List Employee= new List();
     Employee.Add("Komal");
     Employee.Add("Vishal");
     Employee.Add("Rahul");

     ViewBag.Employee= Employee;
     return View();
}
//page code

   <% foreach (var student in ViewBag.Employee)
      { %>
   <%: employee%>

   <% } %>
```



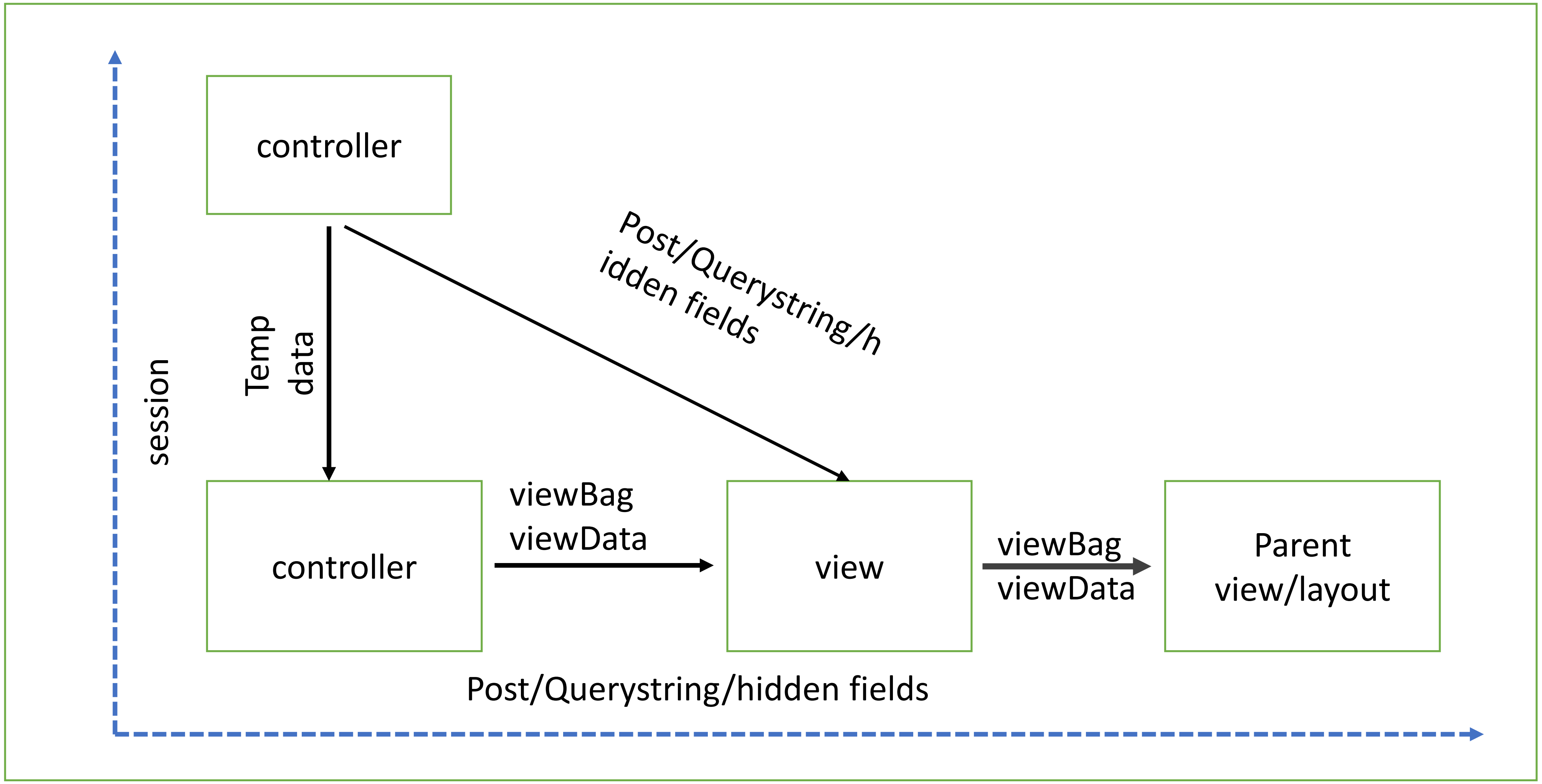
TempData is a dictionary object that is derived from the TempDataDictionary class and stored in a short-lived session.

```
public TempDataDictionary TempData { get; set; }
```

1. TempData is a property of the ControllerBase class.
2. TempData is used to pass data from the current request to subsequent requests (which means redirecting from one page to another).
3. Its life is very short and lies only till the target view is fully loaded.
4. It required typecasting for getting data and checking for null values to avoid errors.
5. It is used to store only one-time messages like error messages, and validation messages. To persist data with TempData refer to this article: Persisting Data with TempData

```
// Populating session
public ActionResult Index()
{
   Session["SomeKey"] = "Welcome to ScholarHat";
   return RedirectToAction("Error");
}

// Retriving session value
public ActionResult Error()
{
   var msg = Session["SomeKey"];
   // Do Something
}
```


controller
Temp data
session
Post/Querystring/hidden fields
controller
viewBag
viewData
view
viewBag
viewData
Parent view/layout
Post/Querystring/hidden fields

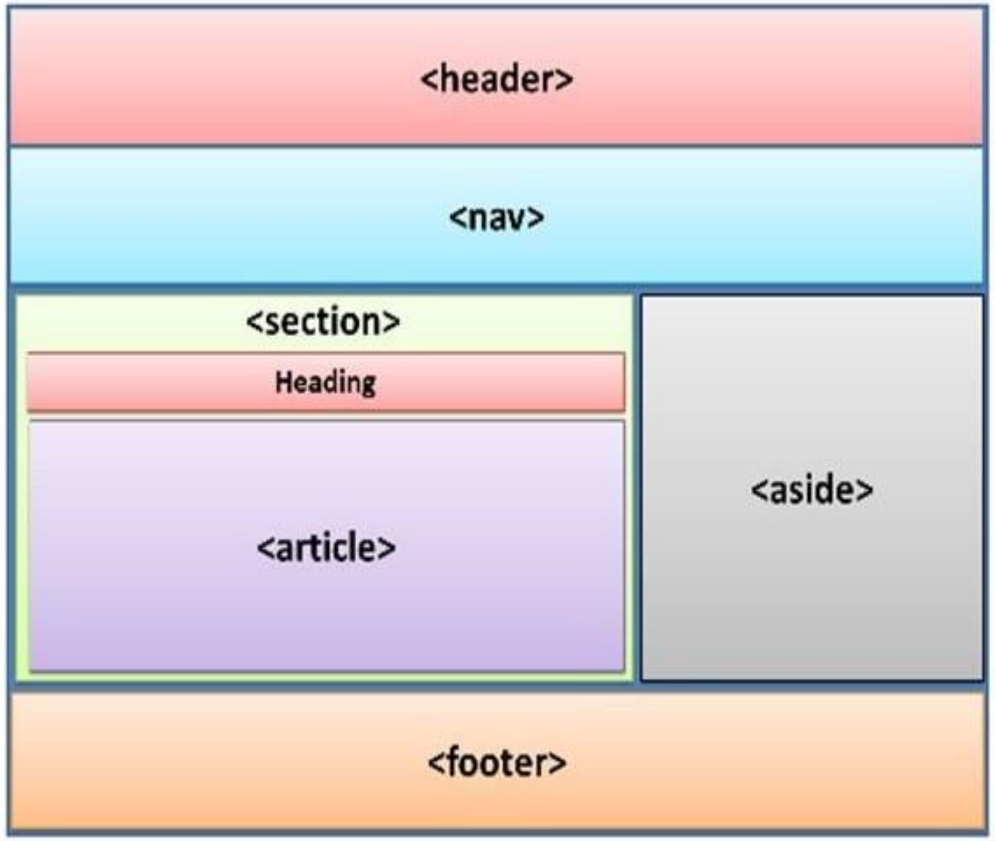
## ViewData Vs ViewBag Vs TempData Vs Session

| **ViewData** | **ViewBag** | **TempData** | **Session** |
|---|---|---|---|
| Is a key-value dictionary derived from ViewDataDictionary. | Is a Dynamic property. It is a wrapper around ViewData. | Is a key-value dictionary derived from TempDataDictionary. | Is a key-value dictionary derived from TempDataDictionary. |
| Un-typed. So, needs type-casting for complex data. | Type casting is not required. | Un-typed: Needs type-casting for complex data type. | Un-typed: Needs type-casting and null checking. |
| Used to pass data between controller and view | Used to pass data between controller and view | Used to pass data between requests. I.e. it helps to pass data from one controller to another controller | Used to store a small amount of data for the duration of the user visiting the website |
| The lifespan is only for the current request | The lifespan is only for the current request | Lifespan is for the current and subsequent requests. The lifespan of TempData can be increased beyond the first redirection using TempData.Keep() method | lifespan of a session persists till it is forcefully destroyed by the server or the user |
| On redirection, the value in the ViewData becomes Null | On redirection, the value in the ViewData becomes Null | The data stored in TempData persists only during redirection | The data stored in Session persists during any number of redirection |
| Does not provide compile-time error-checking | Does not provide compile-time error-checking | Does not provide compile-time error-checking | Does not provide compile-time error-checking |

| ViewData | ViewBag | TempData | Session |
|---|---|---|---|
| ViewData is safe to use in the webfarm environment as they are not dependent on session | It is safe to use ViewBag in the webfarm environment as they are not dependent on session | TempData is not reliable in webfarm with a cluster of servers as the TempData uses ASP.NET Session for storage. The workaround is to set Session State Mode to Out-of-Process and make the data stored in the TempData serializable | Sessions are not reliable in web farm as they are stored on the server's memory. In a webfarm scenario, if a server creates a session and the return request goes to another server in the cluster, then the session will be lost. The workaround is to set Session State Mode to Out-of-Process |
| •**Applications**To pass a list of data to render a drop-down list.<br>•To pass a small amount of data to be rendered in the view.<br>•Not ideal for complex data involving multiple data sources. | •**Applications**To pass a list of data to render a drop-down list.<br>•To pass a small amount of data to be rendered in the view.<br>•Not ideal for complex data involving multiple data sources. | •**Applications**Useful for storing one-time messages like error messages and validation messages.<br>•Used in scenarios to pass small data from one action to another action or one controller call to another controller call. | |

## What is a Layout

Most web apps have a common layout that provides the user with a consistent experience as they navigate from page to page. The layout typically includes common user interface elements such as the app header, navigation or menu elements, and footer.



Common HTML structures such as scripts and stylesheets are also frequently used by many pages within an app. All of these shared elements may be defined in a *layout* file, which can then be referenced by any view used within the app. Layouts reduce duplicate code in views.

by convention, the default layout for an ASP.NET Core app is named **_Layout.cshtml**. The layout files for new ASP.NET Core projects created with the templates are:

Controller with views: Views/Shared/_Layout.cshtml

**Specifying a Layout**

Razor views have a Layout property. Individual views specify a layout by setting this property

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

The layout specified can use a full path

/Views/Shared/_Layout.cshtml) or a partial name (example: _Layout). When a partial name is provided, the Razor view engine searches for the layout file using its standard discovery process. The folder where the handler method (or controller) exists is searched first, followed by the *Shared* folder. This discovery process is identical to the process used to discover partial views.

By default, every layout must call `RenderBody`. Wherever the call to `RenderBody` is placed, the contents of the view will be rendered.

## Sections

A layout can optionally reference one or more *sections*, by calling RenderSection. Sections provide a way to organize where certain page elements should be placed. Each call to RenderSection can specify whether that section is required or optional:

```
<script type="text/javascript"
src="~/scripts/global.js"></script>

@RenderSection("Scripts", required: false)
```

**Layout and Sections in Razor**

•A **Layout** defines the common UI (header, footer, scripts, etc.)
•**Sections** allow views to inject content into specific places of the layout
•Sections help **organize page-specific content** like scripts or styles

**What is RenderSection**?

```
@RenderSection("Scripts", required: false)
```

**RenderSection Method**

•Used inside **_Layout.cshtml**
•Specifies **where a section's content will appear**

•required: true → Section must exist in the view
•required: false → Section is optional

## Defining a Section in a View

**@section Syntax**

•Views define sections using @section
•Content inside section is rendered in layout

```
@section Scripts {
   <script src="~/scripts/main.js"></script>
}
```

•Adds main.js only to that specific page
•Other pages may **skip this section**

**Layout File (_Layout.cshtml )**

```
<script src="~/scripts/global.js"></script>
@RenderSection("Scripts", required: false)
```

**View File (Index.cshtml)**

```
@section Scripts {
   <script src="~/scripts/pageSpecific.js"></script>
}
```

✓ Global script loads for all pages
✓ Page-specific script loads only where needed

## Using Partial Inside a Section

### Validation Scripts Example

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

•Commonly generated by **scaffolding**
•Loads client-side validation scripts
•Clean and reusable approach

**Important Rules About Sections**
**Key Rules**
•Sections are available **only to the immediate layout**
•Sections:
❌ Cannot be accessed from partial views
❌ Cannot be accessed from view components
•Defined section **must be rendered or ignored**
•Otherwise → runtime error

## Ignoring Sections

**IgnoreSection Method**

•Used when layout does NOT want to render a section

```
@IgnoreSection("Scripts")
```

•Prevents Razor engine error
•Useful for conditional layouts

Ignoring Body Content

```
@IgnoreBody()
```

•instructs Razor to ignore @RenderBody()
•Rarely used
•Mostly applicable in special layout scenarios

**Razor View Engine Behavior**

•Razor **tracks**:

- Body rendering
- Section rendering

•Every section must be:

✓ Rendered
✓ Or explicitly ignored

**Otherwise:** InvalidOperationException

✓ Layout controls page structure
✓ Sections control **where view content appears**
✓ RenderSection() defines section placeholder
✓ Sections can be **required or optional**
✓ Sections exist only between **view and its layout**
✓ Razor enforces rendering or ignoring sections

## Error Handling Overview

•ASP.NET Core provides middleware to handle exceptions globally.
•Key approaches:
**1.Developer Exception Page** – detailed info during development.
**3.Exception Handler Lambda** – inline handling with custom logic.
**2.Exception Handler Page** – custom error pages for production.

## Developer Exception Page

•Displays detailed info about unhandled exceptions:
•Stack trace, headers, cookies, query string, endpoint metadata.
•Enabled by default in **Development environment**.
•Middleware: DeveloperExceptionPageMiddleware
•**Important**: Never enable in production.

## Exception Handler Page

•Use app.UseExceptionHandler("/Error") in **production**.
•Catches and logs exceptions.
•Re-executes request using the alternate path.
•Razor Pages: Error.cshtml + ErrorModel.
•MVC: Error action in controller + view.
•Access exception: IExceptionHandlerPathFeature.

## Handling Exceptions by HTTP Method

•Razor Pages: multiple handlers (OnGet, OnPost) for GET/POST errors.
•MVC: apply [HttpGet], [HttpPost] attributes for actions.
•Ensure anonymous access if error page should be public.

**Exception Handler Lambda**

•Inline error handling via lambda in UseExceptionHandler.
•Allows custom response based on exception type.
•Example: set StatusCode, return plain text message.
•Useful for APIs or dynamic responses.

## Introduction to Error Handling

•ASP.NET Core provides multiple approaches to handle errors in web apps:
•Developer Exception Page
•Exception Handler Page
•Exception Handler Lambda
•IExceptionHandler interface
•Status code pages (UseStatusCodePages)
•Problem Details responses
•Always avoid exposing sensitive information in production.

**Developer Exception Page**

•Shows detailed information about unhandled exceptions.
•Enabled by default in **Development** environment.
•Provides:

- Stack trace
- Query string
- Cookies
- Headers

•Runs early in the middleware pipeline.
•**Do not use in Production**.

## Exception Handler Page

- Configured via app.UseExceptionHandler("/Error").
- Handles unhandled exceptions in **Production**.
- Re-executes the request using an alternate pipeline (/Error).
- Key points:
  - Scoped services remain unchanged.
  - Request method preserved (GET, POST, etc.).
  - Avoid HTTP method restrictions if all requests should reach error page.
- Access exception info via IExceptionHandlerPathFeature.

**Exception Handler Lambda**

- Alternative to a separate error page.
- Provides inline handling of exceptions.

```
app.UseExceptionHandler(app => {
    app.Run(async context => {
        context.Response.StatusCode = 500;
        context.Response.ContentType = Text.Plain;
        await context.Response.WriteAsync("An exception was
thrown.");
    });
});
```

- Useful for APIs or custom responses.

## IExceptionHandler Interface

•Centralized callback for handling exceptions.
•Registered with builder.Services.AddExceptionHandler<T>().
•Multiple handlers can be chained.
•Example:

```
public class CustomExceptionHandler : IExceptionHandler
{
    public ValueTask<bool> TryHandleAsync(HttpContext ctx,
Exception ex, CancellationToken token) { ... }
}
```

### Status Code Pages

•by default, ASP.NET Core returns empty responses for HTTP errors (400–599).
•Enable text-only handlers: app.UseStatusCodePages().
•Options:
1.**Format string**: "Status Code: {0}"
2.**Lambda**: Custom response writing
3.**Redirects**: UseStatusCodePagesWithRedirects("/StatusCode/{0}")
4.**Re-execute**: UseStatusCodePagesWithReExecute("/StatusCode/{0}")
•Can disable for specific actions with [SkipStatusCodePages] or IStatusCodePagesFeature.