Passing Data

In ASP.NET Core MVC, you can pass data to views in several ways. Here are the common approachs:

1. Using ViewData

ViewData is a dictionary of key-value pairs.
It’s used to pass data from the controller to the view.

```
// In the Controller
public IActionResult Index()
{
    ViewData["Message"] = "Hello from ViewData!";
    return View();
}

// In the View (Index.cshtml)
<p>@ViewData["Message"]</p>
```

Pros:

- Simple to use.
- Useful for small pieces of data.

Cons:

- Data is loosely typed (you need to cast types manually).

2. Using ViewBag

ViewBag is a dynamic object that allows you to pass data to the view.

```
// In the Controller
public IActionResult Index()
{
    ViewBag.Message = "Hello from ViewBag!";
    return View();
}

// In the View (Index.cshtml)
<p>@ViewBag.Message</p>
```

Pros:

- Easy to use and dynamic.
- No need for type casting like ViewData.

Cons:

- Less type-safe compared to strongly typed models

3. Using Strongly-Typed Models

You can pass a strongly-typed model to the view. This is the most common and recommended approach.

Pros:

- Type-safe.
- Clear and structured.

Cons:

- Requires more setup

(a model class and @model directive).

```
// Model
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

// In the Controller
public IActionResult Index()
{
    var person = new Person { Name = "John Doe", Age = 30 };
    return View(person);
}

// In the View (Index.cshtml)
@model Person
<p>Name: @Model.Name</p>
<p>Age: @Model.Age</p>
```

4. Using TempData

TempData is used to store data temporarily. It persists across a single request and is often used for redirects.

Pros:

- Ideal for temporary data,

such as success or error messages.

Cons:

- Data is only available for one request.

```
// In the Controller
public IActionResult RedirectWithMessage()
{
    TempData["Message"] = "Hello from TempData!";
    return RedirectToAction("Index");
}

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

// In the View (Index.cshtml)
<p>@TempData["Message"]</p>
```

5. Using Dependency Injection

You can inject services or repositories directly into your controller and pass the data they provide to the view.

How Dependency Injection Works Here

1. **Service Registration:** The ProductService is registered in the DI container.

2. **Service Injection:** ASP.NET Core automatically injects an instance of ProductService into the HomeController when it is instantiated.

3. **Data Passing:** The controller action (Index) retrieves data from the service and passes it to the view as the model.

Step 1: Create a Service

First, create a service that provides the data you want to display in your view.

```
// Services/IProductService.cs
public interface IProductService
{
    IEnumerable<string> GetProducts();
}

// Services/ProductService.cs
public class ProductService : IProductService
{
    public IEnumerable<string> GetProducts()
    {
        return new List<string> { "Product A", "Product B", "Product C" };
    }
}
```

Step 2: Register the Service in Program.cs

Register the service in the **dependency injection container**.

```
// Program.cs (ASP.NET Core 6+)
var builder = WebApplication.CreateBuilder(args);

// Register the ProductService
builder.Services.AddScoped<IProductService, ProductService>();

var app = builder.Build();

app.MapControllers();
app.Run();
```

Step 3: Inject the Service into the Controller

Inject the service into a controller to use it.

```
// Controllers/HomeController.cs
using Microsoft.AspNetCore.Mvc;

public class HomeController : Controller
{
    private readonly IProductService _productService;

    public HomeController(IProductService productService)
    {
        _productService = productService;
    }

    public IActionResult Index()
    {
        var products = _productService.GetProducts();
        return View(products);
    }
}
```

Step 4: Pass Data to the View

The Index action in the controller passes the product list to the view as its model.

```
// Views/Home/Index.cshtml
@model IEnumerable<string>

<h1>Product List</h1>
<ul>
    @foreach (var product in Model)
    {
        <li>@product</li>
    }
</ul>
```

This approach ensures loose coupling, easier testing, and better separation of concerns. If you decide to change the implementation of IProductService, you can do so without modifying the controller or view logic.

When to Use Each Method

- **ViewData/ViewBag**: For small, temporary data that doesn’t need strong typing.
- **Strongly-Typed Models**: For structured and reusable data. Recommended for most use cases.
- **TempData**: For one-time messages, especially with redirects.
- **Dependency Injection**: When working with services or repositories.