

Backend Development

Chapter 2: ASP.NET Core

TABLE CONTENT

Razor Syntax

Dependency Injection

Razor syntax

What is Razor?

- A markup syntax for embedding .NET code into webpages.
- Combines Razor markup, C#, and HTML.

File Extensions:

- .cshtml for Razor Pages.
- .razor for Razor Components.

Comparison:

- Similar to templating engines in SPA frameworks like Angular, React, VueJs, and Svelte.

Rendering HTML with Razor

- Default Language: HTML
- Rendering Process:
 - HTML in .cshtml files is rendered by the server unchanged.
- Example: `<p>Hello, World!</p>`
- Rendered HTML: `<p>Hello, World!</p>`

Basic Razor Syntax

- Transition Symbol: @
 - Switches from HTML to C#.
- Razor Expressions:
 - Implicit Expressions: @DateTime.Now
 - Explicit Expressions: @(DateTime.Now) - TimeSpan.FromDays(7)
- Example: <p>@DateTime.Now</p>

Implicit Razor Expressions

- **Usage:**
 - Start with @ followed by C# code.
- **Limitations:**
 - Cannot contain spaces unless the statement clearly ends.
 - Does not support generic methods.
- **Example:** `<p>@DateTime.Now</p>`

Explicit Razor Expressions

- **Usage:**
 - Start with **@()** enclosing C# code.
- **Advantages:**
 - Supports generic methods.
 - Easier to concatenate text with expression results.
- **Example:** `<p>Last week this time: @(DateTime.Now) - TimeSpan.FromDays(7)</p>`

HTML

```
<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>
```


Expression Encoding

- **String Expressions:** HTML encoded.
- **IHtmlContent Expressions:** Rendered directly.
- **Non-IHtmlContent Expressions:** Converted to string and encoded.
- **Example:**
 - `@("Hello World")`
 - **Rendered HTML:** `Hello World`
- **HtmlHelper.Raw:** Renders HTML without encoding.
 - `@Html.Raw("Hello World")`
 - **Rendered HTML:** `Hello World`

Razor Code Blocks

- **Usage:** Start with @{ }.
- **Characteristics:**
 - C# code inside blocks is not rendered.
 - Shares the same scope with expressions.
- **Example:**

```
@{  
    var quote = "The future depends on what you do today. - Mahatma Gandhi";  
}  
  
<p>@quote</p>  
  
@{  
    quote = "Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.";  
}  
  
<p>@quote</p>
```

Content Transitions in Code Blocks

- **Implicit Transitions:** Default to C#.
- **Explicit Delimited Transitions:** Use `<text>` to render HTML within code blocks.
- **Explicit Line Transition:** Use `@:` to render the entire line as HTML.
- Example:

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    @:Name: @person.Name
}
```

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <text>Name: @person.Name</text>
}
```

Conditional Rendering and Loops

- **Conditionals:**

- @if, else if, else
- @switch

- **Loops:**

- @for
- @foreach
- @while
- @do while

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value is odd and small.</p>
}
```

```
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number wasn't 1 or 1337.</p>
        break;
}
```

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

```
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

```
@{ var i = 0; }
@do
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
} while (i < people.Length);
```

```
@{ var i = 0; }
@while (i < people.Length)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
}
```

Directives in Razor

- **Key Directives:**
 - @model
 - @using
 - @inherits
 - @inject
 - @attribute
 - @section
 - @typeparam
- **Functionality:**
 - Modify how views are compiled or behave.

@model Directive

- **Purpose:** Specifies the type of the model passed to the view.
- **Example:**
 - @model LoginViewModel
 - <div>The Login Email: @Model.Email</div>
- **Rendered HTML:**
 - <div>The Login Email: rick@contoso.com</div>

@using Directive

- **Purpose:** Adds C# using directives to the view.
- **Example:**

C#HTML

```
@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>@dir</p>
```

@inherits and @attribute Directives

- **@inherits**: Specifies the base class for the generated view class.

- Example:

CSSHTML

```
@inherits CustomRazorPage<TModel>
```

```
<div>Custom text: @CustomText</div>
```

- **@attribute**: The @attribute directive adds the given attribute to the class of the generated page or view

- Example:

CSSHTML

```
@attribute [Authorize]
```


@inject Directive

Purpose: Injects a service from the service container into the view.

Example:

```
@inject ILogger<MyView> Logger
```

```
<p>@Logger.LogInformation("Hello World")</p>
```

Other Directives

- @section: Used with layouts to render content in different parts of the page.
- @typeparam: Declares a generic type parameter for Razor components.
- @implements: Implements an interface for the generated class.
- Control Structures:
 - @lock
 - @try-catch-finally

Tag Helpers in Razor

Related Directives:

- @addTagHelper
- @removeTagHelper
- @tagHelperPrefix

Functionality: Makes Tag Helpers available to the view and controls their usage.

Comments in Razor

- C# Comments: `/* */`, `//`
- HTML Comments: `<!-- -->`
- Razor Comments: `@* *@` (Not rendered in HTML)

Example:

```
CSHTML
@{
    /* C# comment */
    // Another C# comment
}
<!-- HTML comment -->
```

```
HTML
<!-- HTML comment -->
```

```
CSHTML
@*
    @{
        /* C# comment */
        // Another C# comment
    }
    <!-- HTML comment -->
*@
```

Templated Razor Delegates

Purpose: Define reusable UI snippets.

Example:

C#HTML

```
@using Microsoft.AspNetCore.Html

@functions {
    public static IHtmlContent Repeat(IEnumerable<dynamic> items, int times,
        Func<dynamic, IHtmlContent> template)
    {
        var html = new HtmlContentBuilder();

        foreach (var item in items)
        {
            for (var i = 0; i < times; i++)
            {
                html.AppendHtml(template(item));
            }
        }

        return html;
    }
}
```

Dependency Injection

Introduction to Dependency Injection (DI) in ASP.NET Core

- What is Dependency Injection?
 - A design pattern that allows a class to receive its dependencies from an external source rather than creating them itself.
- Benefits of DI:
 - Enhances testability.
 - Promotes loose coupling.
 - Improves maintainability.
- ASP.NET Core and DI:
 - Built-in support for DI.
 - Controllers explicitly request dependencies via constructors.

Constructor Injection

- Definition:
 - Injecting dependencies through a class constructor.
- How It Works:
 - Services are added as constructor parameters.
 - The runtime resolves the service from the service container.
- Best Practices:
 - Use interfaces to define services.
- Example: IDateTime Service

C#

```
public interface IDateTime
{
    DateTime Now { get; }
}
```


Implementing Constructor Injection

Implementing the IDateTime Interfacecsharp

```
C#  
  
public class SystemDateTime : IDateTime  
{  
    public DateTime Now  
    {  
        get { return DateTime.Now; }  
    }  
}
```

Registering the Service in the Service Container

Note: AddSingleton defines the service lifetime.

```
C#  
  
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddSingleton<IDateTime, SystemDateTime>();  
  
    services.AddControllersWithViews();  
}
```

Using Constructor Injection in Controllers

Example: HomeController

Outcome: Displays a greeting based on the current server time.

C#

```
public class HomeController : Controller
{
    private readonly IDateTime _dateTime;

    public HomeController(IDateTime dateTime)
    {
        _dateTime = dateTime;
    }

    public IActionResult Index()
    {
        var serverTime = _dateTime.Now;
        if (serverTime.Hour < 12)
        {
            ViewData["Message"] = "It's morning here - Good Morning!";
        }
        else if (serverTime.Hour < 17)
        {
            ViewData["Message"] = "It's afternoon here - Good Afternoon!";
        }
        else
        {
            ViewData["Message"] = "It's evening here - Good Evening!";
        }
        return View();
    }
}
```

Action Injection with [FromServices]

- Definition:
 - Injecting services directly into action methods without using constructor injection.
- Usage:
 - Apply the [FromServices] attribute to action parameters.
- Example: About Action Method
- Benefits:
 - Reduces the need for multiple constructor parameters.
 - Useful for injecting dependencies used by specific actions only.

C#

```
public IActionResult About([FromServices] IDateTime dateTime)
{
    return Content( $"Current server time: {dateTime.Now}");
}
```

Action Injection with [FromKeyedServices]

- **Definition:**
 - Accessing keyed services from the DI container.
- **Usage:**
 - Apply the [FromKeyedServices("key")] attribute to action parameters.
- **Setup:**

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
builder.Services.AddControllers();

var app = builder.Build();

app.MapControllers();

app.Run();
```

Action Injection with [FromKeyedServices]

ICache Interface and Implementations:

```
public interface ICache
{
    object Get(string key);
}

public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}
```

Action Injection with [FromKeyedServices]

Custom Controller:

```
[ApiController]
[Route("/cache")]
public class CustomServicesApiController : Controller
{
    [HttpGet("big")]
    public ActionResult<object> GetBigCache([FromKeyedServices("big")] ICache cache)
    {
        return cache.Get("data-mvc");
    }

    [HttpGet("small")]
    public ActionResult<object> GetSmallCache([FromKeyedServices("small")] ICache cache)
    {
        return cache.Get("data-mvc");
    }
}
```

Accessing Settings from a Controller

Importance:

- Accessing application or configuration settings within controllers is a common requirement.

Preferred Approach:

- Use the Options Pattern rather than injecting IConfiguration directly.

Steps:

- Define an Options Class
- Register the Options in the Service Container
- Configure the Application to Read Settings from a JSON File
- Inject and Use the Options in the Controller

Defining the Options Class

Example:

SampleWebSettings

C#

```
public class SampleWebSettings
{
    public string Title { get; set; }
    public int Updates { get; set; }
}
```


Registering the Options Class

- Register in ConfigureServices

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<DateTime, System.DateTime>();
    services.Configure<SampleWebSettings>(Configuration);

    services.AddControllersWithViews();
}
```

Configuring the Application to Read Settings

- Setup in Program.cs

```
C#  
  
public class Program  
{  
    public static void Main(string[] args)  
    {  
        CreateHostBuilder(args).Build().Run();  
    }  
  
    public static IHostBuilder CreateHostBuilder(string[] args) =>  
        Host.CreateDefaultBuilder(args)  
            .ConfigureAppConfiguration((hostingContext, config) =>  
            {  
                config.AddJsonFile("samplewebsettings.json",  
                    optional: false,  
                    reloadOnChange: true);  
            })  
            .ConfigureWebHostDefaults(webBuilder =>  
            {  
                webBuilder.UseStartup<Startup>();  
            });  
    }  
}
```

Sample JSON Configuration
(samplewebsettings.json):

```
{  
    "Title": "My ASP.NET Core App",  
    "Updates": 5  
}
```

Injecting and Using Options in a Controller

- **Example: SettingsController**

C#

```
public class SettingsController : Controller
{
    private readonly SampleWebSettings _settings;

    public SettingsController(IOptions<SampleWebSettings> settingsOptions)
    {
        _settings = settingsOptions.Value;
    }

    public IActionResult Index()
    {
        ViewData["Title"] = _settings.Title;
        ViewData["Updates"] = _settings.Updates;
        return View();
    }
}
```

Summary of Dependency Injection Methods

Constructor Injection:

- Preferred for mandatory dependencies.
- Promotes immutability and testability.

Action Injection with [FromServices]:

- Ideal for optional or action-specific dependencies.

Action Injection with [FromKeyedServices]:

- Useful for resolving multiple implementations of the same service interface.

Accessing Settings:

- Utilize the Options Pattern for configuration settings.

Dependency injection into views in ASP.NET Core

ASP.NET Core supports dependency injection into views. This can be useful for view-specific services, such as localization or data required only for populating view elements. Most of the data views display should be passed in from the controller.

Configuration injection

The values in settings files, such as appsettings.json and appsettings.Development.json, can be injected into a view. Consider the appsettings.Development.json

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "MyRoot": {
    "MyParent": {
      "MyChildName": "Joe"
    }
  }
}
```

The following markup displays the configuration value in a Razor Pages view:

CSHTML

```
@page
@model PrivacyModel
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration
@{
    ViewData["Title"] = "Privacy RP";
}
<h1>@ViewData["Title"]</h1>

<p>PR Privacy</p>

<h2>
    MyRoot:MyParent:MyChildName: @Configuration["MyRoot:MyParent:MyChildName"]
</h2>
```

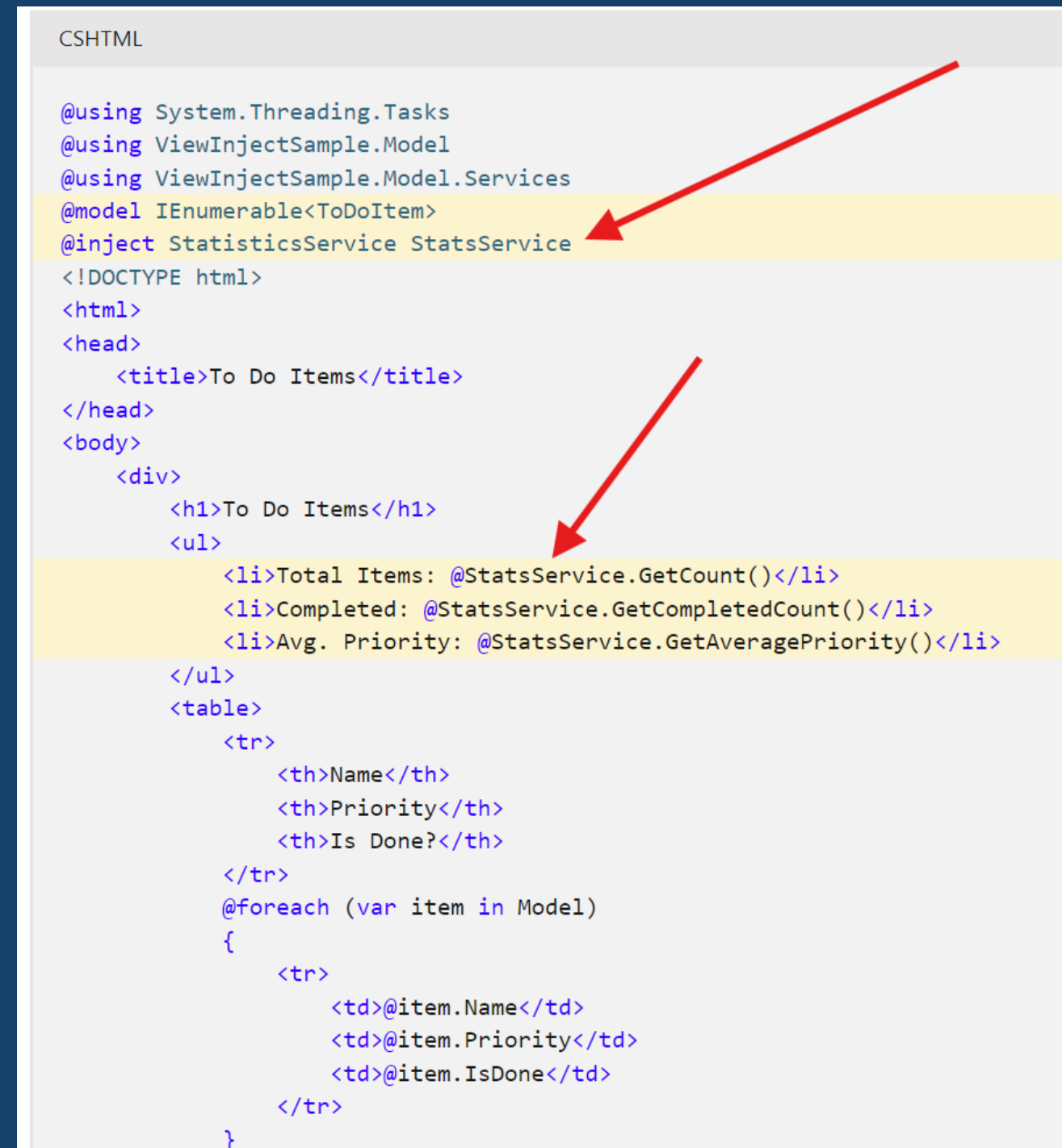
Service injection

A service can be injected into a view using the `@inject` directive.

```
CSHTML

@using System.Threading.Tasks
@using ViewInjectSample.Model
@using ViewInjectSample.Model.Services
@model IEnumerable<ToDoItem>
@Inject StatisticsService StatsService

<!DOCTYPE html>
<html>
<head>
    <title>To Do Items</title>
</head>
<body>
    <div>
        <h1>To Do Items</h1>
        <ul>
            <li>Total Items: @StatsService.GetCount()</li>
            <li>Completed: @StatsService.GetCompletedCount()</li>
            <li>Avg. Priority: @StatsService.GetAveragePriority()</li>
        </ul>
        <table>
            <tr>
                <th>Name</th>
                <th>Priority</th>
                <th>Is Done?</th>
            </tr>
            @foreach (var item in Model)
            {
                <tr>
                    <td>@item.Name</td>
                    <td>@item.Priority</td>
                    <td>@item.IsDone</td>
                </tr>
            }
        </table>
    </div>
</body>
</html>
```



Service injection

This view displays a list of `ToDoItem` instances, along with a summary showing overall statistics. The summary is populated from the injected `StatisticsService`. This service is registered for dependency injection in `ConfigureServices` in `Program.cs`:

C#

```
using ViewInjectSample.Helpers;
using ViewInjectSample.Infrastructure;
using ViewInjectSample.Interfaces;
using ViewInjectSample.Model.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

builder.Services.AddTransient<IToDoItemRepository, ToDoItemRepository>();
builder.Services.AddTransient<StatisticsService>();
builder.Services.AddTransient<ProfileOptionsService>();
builder.Services.AddTransient<MyHtmlHelper>();

var app = builder.Build();
```


Service injection

The StatisticsService performs some calculations on the set of ToDoItem instances, which it accesses via a repository:

```
using System.Linq;
using ViewInjectSample.Interfaces;

namespace ViewInjectSample.Model.Services
{
    public class StatisticsService
    {
        private readonly IToDoItemRepository _todoItemRepository;

        public StatisticsService(IToDoItemRepository todoItemRepository)
        {
            _todoItemRepository = todoItemRepository;
        }

        public int GetCount()
        {
            return _todoItemRepository.List().Count();
        }

        public int GetCompletedCount()
        {
            return _todoItemRepository.List().Count(x => x.IsDone);
        }


        public double GetAveragePriority()
        {
            if (_todoItemRepository.List().Count() == 0)
            {
                return 0.0;
            }

            return _todoItemRepository.List().Average(x => x.Priority);
        }
    }
}
```

Service injection

The sample repository uses an in-memory collection. An in-memory implementation shouldn't be used for large, remotely accessed data sets.

The sample displays data from the model bound to the view and the service injected into the view:



← → ↻ 🏠 localhost:10653/todo

To Do Items

- Total Items: 50
- Completed: 17
- Avg. Priority: 3

Name	Priority	Is Done?
Task 1	1	True
Task 2	2	False
Task 3	3	False
Task 4	4	True
Task 5	5	False

Populating Lookup Data

View injection can be useful to populate options in UI elements, such as dropdown lists. Consider a user profile form that includes options for specifying gender, state, and other preferences. Rendering such a form using a standard approach might require the controller or Razor Page to:

- Request data access services for each of the sets of options.
- Populate a model or ViewBag with each set of options to be bound.

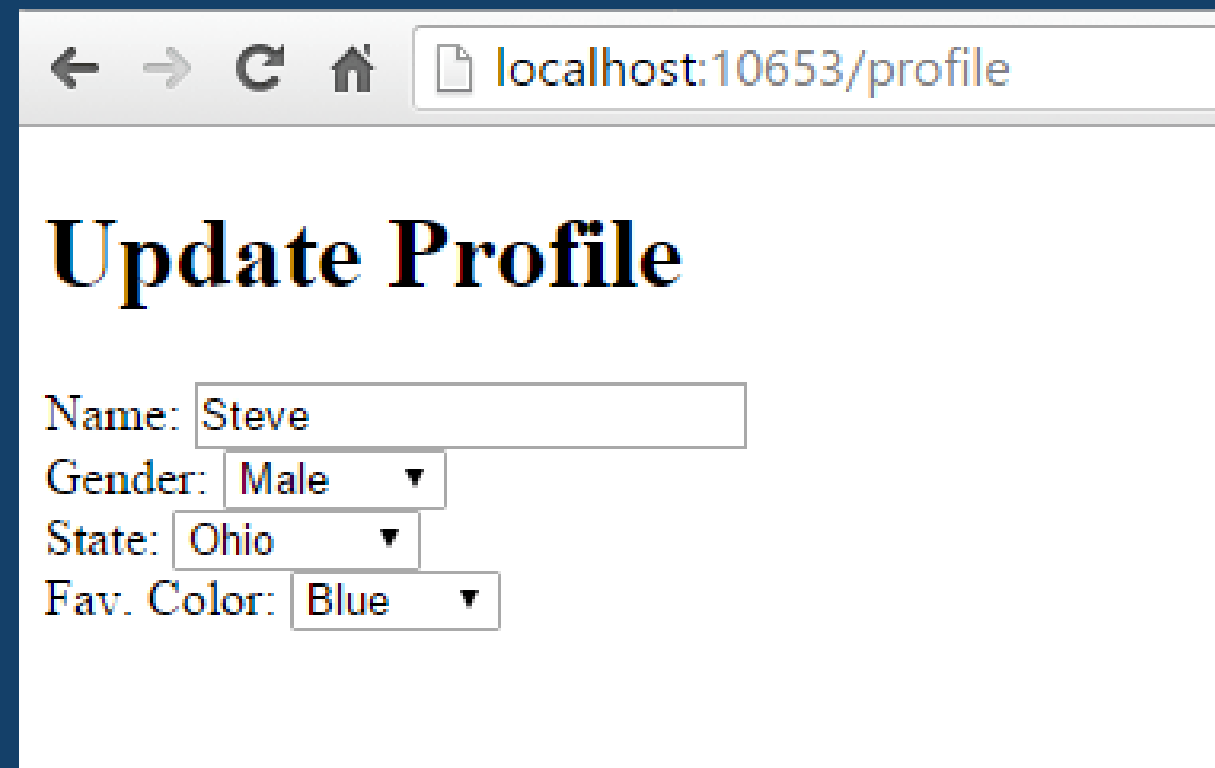
An alternative approach injects services directly into the view to obtain the options. This minimizes the amount of code required by the controller or razor Page, moving this view element construction logic into the view itself. The controller action or Razor Page to display a profile editing form only needs to pass the form the profile instance:

Populating Lookup Data

Code:

```
C#  
  
using Microsoft.AspNetCore.Mvc;  
using ViewInjectSample.Model;  
  
namespace ViewInjectSample.Controllers;  
  
public class ProfileController : Controller  
{  
    public IActionResult Index()  
    {  
        // A real app would up profile based on the user.  
        var profile = new Profile()  
        {  
            Name = "Rick",  
            FavColor = "Blue",  
            Gender = "Male",  
            State = new State("Ohio","OH")  
        };  
        return View(profile);  
    }  
}
```

The HTML form used to update the preferences includes dropdown lists for three of the properties:



← → ↻ 🏠 localhost:10653/profile

Update Profile

Name:

Gender:

State:

Fav. Color:

Populating Lookup Data

These lists are populated by a service that has been injected into the view:

C#HTML

```
@using System.Threading.Tasks
@using ViewInjectSample.Model.Services
@model ViewInjectSample.Model.Profile
@Inject ProfileOptionsService Options
<!DOCTYPE html>
<html>
<head>
    <title>Update Profile</title>
</head>
<body>
<div>
    <h1>Update Profile</h1>
    Name: @Html.TextBoxFor(m => m.Name)
    <br/>
    Gender: @Html.DropDownList("Gender",
        Options.ListGenders().Select(g =>
            new SelectListItem() { Text = g, Value = g }))
    <br/>
    State: @Html.DropDownListFor(m => m.State!.Code,
        Options.ListStates().Select(s =>
            new SelectListItem() { Text = s.Name, Value = s.Code}))
    <br />
    Fav. Color: @Html.DropDownList("FavColor",
        Options.ListColors().Select(c =>
            new SelectListItem() { Text = c, Value = c }))
    </div>
</body>
</html>
```

Populating Lookup Data

The ProfileOptionsService is a UI-level service designed to provide just the data needed for this form:

C#

```
namespace ViewInjectSample.Model.Services;

public class ProfileOptionsService
{
    public List<string> ListGenders()
    {
        // Basic sample
        return new List<string>() {"Female", "Male"};
    }

    public List<State> ListStates()
    {
        // Add a few states
        return new List<State>()
        {
            new State("Alabama", "AL"),
            new State("Alaska", "AK"),
            new State("Ohio", "OH")
        };
    }

    public List<string> ListColors()
    {
        return new List<string>() { "Blue", "Green", "Red", "Yellow" };
    }
}
```


Start your future at EIU

Thank You