**View components in ASP.NET Core**

* 12 minutes to read

[View or download sample code](https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/mvc/views/view-components/sample) ([how to download](https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0#how-to-download-a-sample))

**View components**

View components are similar to partial views, but they're much more powerful. View components don't use model binding, and only depend on the data provided when calling into it. This article was written using controllers and views, but view components also work with Razor Pages.

A view component:

* Renders a chunk rather than a whole response.
* Includes the same separation-of-concerns and testability benefits found between a controller and view.
* Can have parameters and business logic.
* Is typically invoked from a layout page.

View components are intended anywhere you have reusable rendering logic that's too complex for a partial view, such as:

* Dynamic navigation menus
* Tag cloud (where it queries the database)
* Login panel
* Shopping cart
* Recently published articles
* Sidebar content on a typical blog
* A login panel that would be rendered on every page and show either the links to log out or log in, depending on the log in state of the user

A view component consists of two parts: the class (typically derived from [ViewComponent](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.viewcomponent)) and the result it returns (typically a view). Like controllers, a view component can be a POCO, but most developers will want to take advantage of the methods and properties available by deriving from ViewComponent.

When considering if view components meet an app's specifications, consider using Razor components instead. Razor components also combine markup with C# code to produce reusable UI units. Razor components are designed for developer productivity when providing client-side UI logic and composition. For more information, see [ASP.NET Core Razor components](https://docs.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-5.0). For information on how to incorporate Razor components into an MVC or Razor Pages app, see [Prerender and integrate ASP.NET Core Razor components](https://docs.microsoft.com/en-us/aspnet/core/blazor/components/prerendering-and-integration?pivots=server&view=aspnetcore-5.0).

**Creating a view component**

This section contains the high-level requirements to create a view component. Later in the article, we'll examine each step in detail and create a view component.

**The view component class**

A view component class can be created by any of the following:

* Deriving from *ViewComponent*
* Decorating a class with the [ViewComponent] attribute, or deriving from a class with the [ViewComponent] attribute
* Creating a class where the name ends with the suffix *ViewComponent*

Like controllers, view components must be public, non-nested, and non-abstract classes. The view component name is the class name with the "ViewComponent" suffix removed. It can also be explicitly specified using the ViewComponentAttribute.Name property.

A view component class:

* Fully supports constructor [dependency injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0)
* Doesn't take part in the controller lifecycle, which means you can't use [filters](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-5.0) in a view component

**View component methods**

A view component defines its logic in an InvokeAsync method that returns a Task<IViewComponentResult> or in a synchronous Invoke method that returns an IViewComponentResult. Parameters come directly from invocation of the view component, not from model binding. A view component never directly handles a request. Typically, a view component initializes a model and passes it to a view by calling the View method. In summary, view component methods:

* Define an InvokeAsync method that returns a Task<IViewComponentResult> or a synchronous Invoke method that returns an IViewComponentResult.
* Typically initializes a model and passes it to a view by calling the ViewComponent View method.
* Parameters come from the calling method, not HTTP. There's no model binding.
* Are not reachable directly as an HTTP endpoint. They're invoked from your code (usually in a view). A view component never handles a request.
* Are overloaded on the signature rather than any details from the current HTTP request.

**View search path**

The runtime searches for the view in the following paths:

* /Views/{Controller Name}/Components/{View Component Name}/{View Name}
* /Views/Shared/Components/{View Component Name}/{View Name}
* /Pages/Shared/Components/{View Component Name}/{View Name}

The search path applies to projects using controllers + views and Razor Pages.

The default view name for a view component is *Default*, which means your view file will typically be named *Default.cshtml*. You can specify a different view name when creating the view component result or when calling the View method.

We recommend you name the view file *Default.cshtml* and use the *Views/Shared/Components/{View Component Name}/{View Name}* path. The PriorityList view component used in this sample uses *Views/Shared/Components/PriorityList/Default.cshtml* for the view component view.

**Customize the view search path**

To customize the view search path, modify Razor's [ViewLocationFormats](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.razor.razorviewengineoptions.viewlocationformats" \l "Microsoft_AspNetCore_Mvc_Razor_RazorViewEngineOptions_ViewLocationFormats) collection. For example, to search for views within the path "/Components/{View Component Name}/{View Name}", add a new item to the collection:

C#Copy

services.AddMvc()

.AddRazorOptions(options =>

{

options.ViewLocationFormats.Add("/{0}.cshtml");

})

.SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

In the preceding code, the placeholder "{0}" represents the path "Components/{View Component Name}/{View Name}".

**Invoking a view component**

To use the view component, call the following inside a view:

CSHTMLCopy

@await Component.InvokeAsync("Name of view component", {Anonymous Type Containing Parameters})

The parameters will be passed to the InvokeAsync method. The PriorityList view component developed in the article is invoked from the *Views/ToDo/Index.cshtml* view file. In the following, the InvokeAsync method is called with two parameters:

CSHTMLCopy

@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })

**Invoking a view component as a Tag Helper**

For ASP.NET Core 1.1 and higher, you can invoke a view component as a [Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-5.0):

CSHTMLCopy

<vc:priority-list max-priority="2" is-done="false">

</vc:priority-list>

Pascal-cased class and method parameters for Tag Helpers are translated into their [kebab case](https://stackoverflow.com/questions/11273282/whats-the-name-for-dash-separated-case/12273101). The Tag Helper to invoke a view component uses the <vc></vc> element. The view component is specified as follows:

CSHTMLCopy

<vc:[view-component-name]

parameter1="parameter1 value"

parameter2="parameter2 value">

</vc:[view-component-name]>

To use a view component as a Tag Helper, register the assembly containing the view component using the @addTagHelper directive. If your view component is in an assembly called MyWebApp, add the following directive to the *\_ViewImports.cshtml* file:

CSHTMLCopy

@addTagHelper \*, MyWebApp

You can register a view component as a Tag Helper to any file that references the view component. See [Managing Tag Helper Scope](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-5.0#managing-tag-helper-scope) for more information on how to register Tag Helpers.

The InvokeAsync method used in this tutorial:

CSHTMLCopy

@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })

In Tag Helper markup:

CSHTMLCopy

<vc:priority-list max-priority="2" is-done="false">

</vc:priority-list>

In the sample above, the PriorityList view component becomes priority-list. The parameters to the view component are passed as attributes in kebab case.

**Invoking a view component directly from a controller**

View components are typically invoked from a view, but you can invoke them directly from a controller method. While view components don't define endpoints like controllers, you can easily implement a controller action that returns the content of a ViewComponentResult.

In this example, the view component is called directly from the controller:

C#Copy

public IActionResult IndexVC()

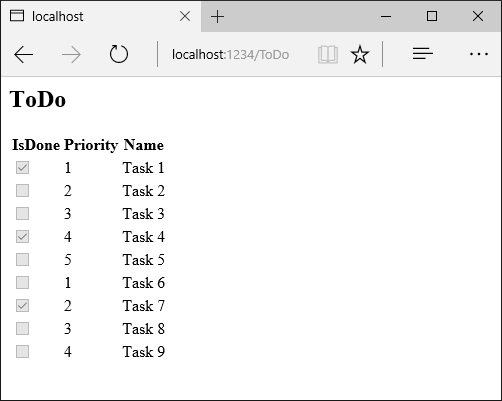
{

return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });

}

**Walkthrough: Creating a simple view component**

[Download](https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/mvc/views/view-components/sample), build and test the starter code. It's a simple project with a ToDo controller that displays a list of *ToDo* items.



**Add a ViewComponent class**

Create a *ViewComponents* folder and add the following PriorityListViewComponent class:

C#Copy

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents

{

public class PriorityListViewComponent : ViewComponent

{

private readonly ToDoContext db;

public PriorityListViewComponent(ToDoContext context)

{

db = context;

}

public async Task<IViewComponentResult> InvokeAsync(

int maxPriority, bool isDone)

{

var items = await GetItemsAsync(maxPriority, isDone);

return View(items);

}

private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)

{

return db.ToDo.Where(x => x.IsDone == isDone &&

x.Priority <= maxPriority).ToListAsync();

}

}

}

Notes on the code:

* View component classes can be contained in **any** folder in the project.
* Because the class name PriorityList**ViewComponent** ends with the suffix **ViewComponent**, the runtime will use the string "PriorityList" when referencing the class component from a view. I'll explain that in more detail later.
* The [ViewComponent] attribute can change the name used to reference a view component. For example, we could've named the class XYZ and applied the ViewComponent attribute:

C#Copy

[ViewComponent(Name = "PriorityList")]

public class XYZ : ViewComponent

* The [ViewComponent] attribute above tells the view component selector to use the name PriorityList when looking for the views associated with the component, and to use the string "PriorityList" when referencing the class component from a view. I'll explain that in more detail later.
* The component uses [dependency injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0) to make the data context available.
* InvokeAsync exposes a method which can be called from a view, and it can take an arbitrary number of arguments.
* The InvokeAsync method returns the set of ToDo items that satisfy the isDone and maxPriority parameters.

**Create the view component Razor view**

* Create the *Views/Shared/Components* folder. This folder **must** be named *Components*.
* Create the *Views/Shared/Components/PriorityList* folder. This folder name must match the name of the view component class, or the name of the class minus the suffix (if we followed convention and used the *ViewComponent* suffix in the class name). If you used the ViewComponent attribute, the class name would need to match the attribute designation.
* Create a *Views/Shared/Components/PriorityList/Default.cshtml* Razor view:

CSHTMLCopy

@model IEnumerable<ViewComponentSample.Models.TodoItem>

<h3>Priority Items</h3>

<ul>

@foreach (var todo in Model)

{

<li>@todo.Name</li>

}

</ul>

The Razor view takes a list of TodoItem and displays them. If the view component InvokeAsync method doesn't pass the name of the view (as in our sample), *Default* is used for the view name by convention. Later in the tutorial, I'll show you how to pass the name of the view. To override the default styling for a specific controller, add a view to the controller-specific view folder (for example *Views/ToDo/Components/PriorityList/Default.cshtml)*.

If the view component is controller-specific, you can add it to the controller-specific folder (*Views/ToDo/Components/PriorityList/Default.cshtml*).

* Add a div containing a call to the priority list component to the bottom of the *Views/ToDo/index.cshtml* file:

CSHTMLCopy

</table>

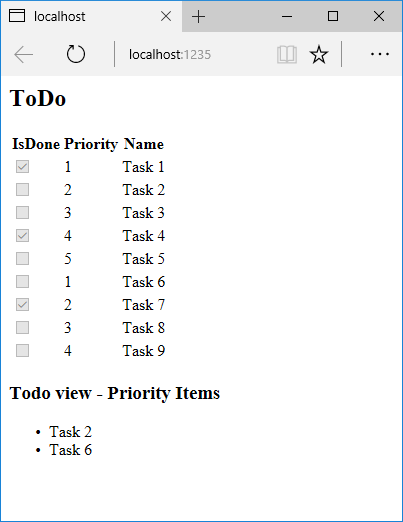
<div>

@await Component.InvokeAsync("PriorityList", new { maxPriority = 2, isDone = false })

</div>

The markup @await Component.InvokeAsync shows the syntax for calling view components. The first argument is the name of the component we want to invoke or call. Subsequent parameters are passed to the component. InvokeAsync can take an arbitrary number of arguments.

Test the app. The following image shows the ToDo list and the priority items:



You can also call the view component directly from the controller:

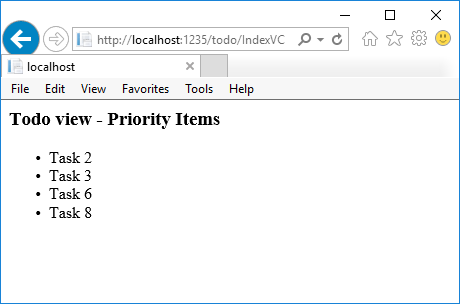
C#Copy

public IActionResult IndexVC()

{

return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });

}



**Specifying a view name**

A complex view component might need to specify a non-default view under some conditions. The following code shows how to specify the "PVC" view from the InvokeAsync method. Update the InvokeAsync method in the PriorityListViewComponent class.

C#Copy

public async Task<IViewComponentResult> InvokeAsync(

int maxPriority, bool isDone)

{

string MyView = "Default";

// If asking for all completed tasks, render with the "PVC" view.

if (maxPriority > 3 && isDone == true)

{

MyView = "PVC";

}

var items = await GetItemsAsync(maxPriority, isDone);

return View(MyView, items);

}

Copy the *Views/Shared/Components/PriorityList/Default.cshtml* file to a view named *Views/Shared/Components/PriorityList/PVC.cshtml*. Add a heading to indicate the PVC view is being used.

CSHTMLCopy

@model IEnumerable<ViewComponentSample.Models.TodoItem>

<h2> PVC Named Priority Component View</h2>

<h4>@ViewBag.PriorityMessage</h4>

<ul>

@foreach (var todo in Model)

{

<li>@todo.Name</li>

}

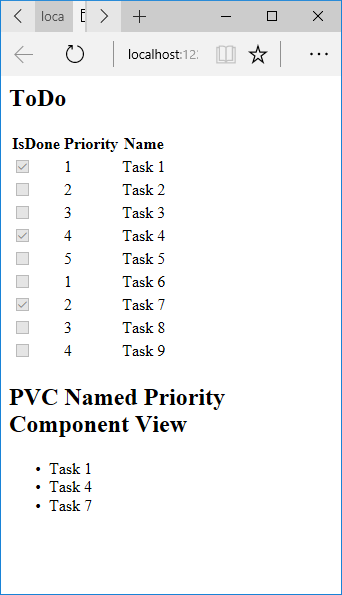
</ul>

Update *Views/ToDo/Index.cshtml*:

CSHTMLCopy

@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })

Run the app and verify PVC view.



If the PVC view isn't rendered, verify you are calling the view component with a priority of 4 or higher.

**Examine the view path**

* Change the priority parameter to three or less so the priority view isn't returned.
* Temporarily rename the *Views/ToDo/Components/PriorityList/Default.cshtml* to *1Default.cshtml*.
* Test the app, you'll get the following error:

Copy

An unhandled exception occurred while processing the request.

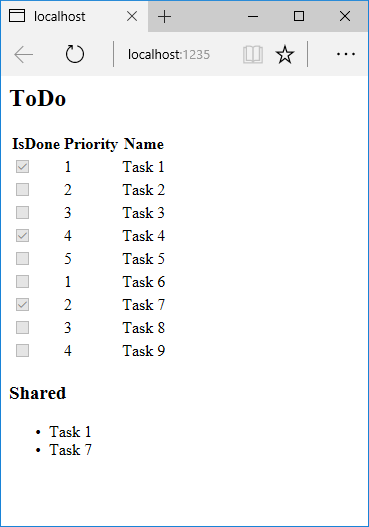
InvalidOperationException: The view 'Components/PriorityList/Default' wasn't found. The following locations were searched:

/Views/ToDo/Components/PriorityList/Default.cshtml

/Views/Shared/Components/PriorityList/Default.cshtml

EnsureSuccessful

* Copy *Views/ToDo/Components/PriorityList/1Default.cshtml* to *Views/Shared/Components/PriorityList/Default.cshtml*.
* Add some markup to the *Shared* ToDo view component view to indicate the view is from the *Shared* folder.
* Test the **Shared** component view.



**Avoiding hard-coded strings**

If you want compile time safety, you can replace the hard-coded view component name with the class name. Create the view component without the "ViewComponent" suffix:

C#Copy

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents

{

public class PriorityList : ViewComponent

{

private readonly ToDoContext db;

public PriorityList(ToDoContext context)

{

db = context;

}

public async Task<IViewComponentResult> InvokeAsync(

int maxPriority, bool isDone)

{

var items = await GetItemsAsync(maxPriority, isDone);

return View(items);

}

private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)

{

return db.ToDo.Where(x => x.IsDone == isDone &&

x.Priority <= maxPriority).ToListAsync();

}

}

}

Add a using statement to your Razor view file, and use the nameof operator:

CSHTMLCopy

@using ViewComponentSample.Models

@using ViewComponentSample.ViewComponents

@model IEnumerable<TodoItem>

<h2>ToDo nameof</h2>

<!-- Markup removed for brevity. -->

<div>

@\*

Note:

To use the below line, you need to #define no\_suffix in ViewComponents/PriorityList.cs or it won't compile.

By doing so it will cause a problem to index as there will be multiple viewcomponents

with the same name after the compiler removes the suffix "ViewComponent"

\*@

@\*@await Component.InvokeAsync(nameof(PriorityList), new { maxPriority = 4, isDone = true })\*@

</div>

**Perform synchronous work**

The framework handles invoking a synchronous Invoke method if you don't need to perform asynchronous work. The following method creates a synchronous Invoke view component:

C#Copy

public class PriorityList : ViewComponent

{

public IViewComponentResult Invoke(int maxPriority, bool isDone)

{

var items = new List<string> { $"maxPriority: {maxPriority}", $"isDone: {isDone}" };

return View(items);

}

}

The view component's Razor file lists the strings passed to the Invoke method (*Views/Home/Components/PriorityList/Default.cshtml*):

CSHTMLCopy

@model List<string>

<h3>Priority Items</h3>

<ul>

@foreach (var item in Model)

{

<li>@item</li>

}

</ul>

The view component is invoked in a Razor file (for example, *Views/Home/Index.cshtml*) using one of the following approaches:

* [IViewComponentHelper](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.iviewcomponenthelper)
* [Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-5.0)

To use the [IViewComponentHelper](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.iviewcomponenthelper) approach, call Component.InvokeAsync:

CSHTMLCopy

@await Component.InvokeAsync(nameof(PriorityList), new { maxPriority = 4, isDone = true })

To use the Tag Helper, register the assembly containing the View Component using the @addTagHelper directive (the view component is in an assembly called MyWebApp):

CSHTMLCopy

@addTagHelper \*, MyWebApp

Use the view component Tag Helper in the Razor markup file:

CSHTMLCopy

<vc:priority-list max-priority="999" is-done="false">

</vc:priority-list>

The method signature of PriorityList.Invoke is synchronous, but Razor finds and calls the method with Component.InvokeAsync in the markup file.

**All view component parameters are required**

Each parameter in a view component is a required attribute. See [this GitHub issue](https://github.com/dotnet/AspNetCore/issues/5011). If any parameter is omitted:

* The InvokeAsync method signature won't match, therefore the method won't execute.
* The ViewComponent won't render any markup.
* No errors will be thrown.

**Dependency injection into controllers in ASP.NET Core**

* 02/24/2019
* 4 minutes to read
  + [](https://github.com/ardalis)
  + [](https://github.com/Rick-Anderson)
  + [](https://github.com/scottaddie)
  + [](https://github.com/guardrex)
  + [](https://github.com/tdykstra)
  + +4

By [Shadi Namrouti](https://github.com/shadinamrouti), [Rick Anderson](https://twitter.com/RickAndMSFT), and [Steve Smith](https://github.com/ardalis)

ASP.NET Core MVC controllers request dependencies explicitly via constructors. ASP.NET Core has built-in support for [dependency injection (DI)](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0). DI makes apps easier to test and maintain.

[View or download sample code](https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/mvc/controllers/dependency-injection/sample) ([how to download](https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0#how-to-download-a-sample))

**Constructor Injection**

Services are added as a constructor parameter, and the runtime resolves the service from the service container. Services are typically defined using interfaces. For example, consider an app that requires the current time. The following interface exposes the IDateTime service:

C#Copy

public interface IDateTime

{

DateTime Now { get; }

}

The following code implements the IDateTime interface:

C#Copy

public class SystemDateTime : IDateTime

{

public DateTime Now

{

get { return DateTime.Now; }

}

}

Add the service to the service container:

C#Copy

public void ConfigureServices(IServiceCollection services)

{

services.AddSingleton<IDateTime, SystemDateTime>();

services.AddControllersWithViews();

}

For more information on [AddSingleton](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.servicecollectionserviceextensions.addsingleton), see [DI service lifetimes](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0#service-lifetimes).

The following code displays a greeting to the user based on the time of day:

C#Copy

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();

}

Run the app and a message is displayed based on the time.

**Action injection with FromServices**

The [FromServicesAttribute](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.fromservicesattribute) enables injecting a service directly into an action method without using constructor injection:

C#Copy

public IActionResult About([FromServices] IDateTime dateTime)

{

return Content( $"Current server time: {dateTime.Now}");

}

**Access settings from a controller**

Accessing app or configuration settings from within a controller is a common pattern. The *options pattern* described in [Options pattern in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-5.0) is the preferred approach to manage settings. Generally, don't directly inject [IConfiguration](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.configuration.iconfiguration) into a controller.

Create a class that represents the options. For example:

C#Copy

public class SampleWebSettings

{

public string Title { get; set; }

public int Updates { get; set; }

}

Add the configuration class to the services collection:

C#Copy

public void ConfigureServices(IServiceCollection services)

{

services.AddSingleton<IDateTime, SystemDateTime>();

services.Configure<SampleWebSettings>(Configuration);

services.AddControllersWithViews();

}

Configure the app to read the settings from a JSON-formatted file:

C#Copy

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>();

});

}

The following code requests the IOptions<SampleWebSettings> settings from the service container and uses them in the Index method:

C#Copy

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();

}

}

**Additional resources**

* See [Test controller logic in ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/testing?view=aspnetcore-5.0) to learn how to make code easier to test by explicitly requesting dependencies in controllers.
* [Replace the default dependency injection container with a third party implementation](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0#default-service-container-replacement).

**Dependency injection into views in ASP.NET Core**

* 10/14/2016
* 4 minutes to read
  + [](https://github.com/ardalis)
  + [](https://github.com/Rick-Anderson)
  + [](https://github.com/scottaddie)
  + [](https://github.com/serpent5)
  + [](https://github.com/guardrex)
  + +7

By [Steve Smith](https://ardalis.com/)

ASP.NET Core supports [dependency injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0) into views. This can be useful for view-specific services, such as localization or data required only for populating view elements. You should try to maintain [separation of concerns](https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles#separation-of-concerns) between your controllers and views. Most of the data your views display should be passed in from the controller.

[View or download sample code](https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/mvc/views/dependency-injection/sample) ([how to download](https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0#how-to-download-a-sample))

**Configuration injection**

*appsettings.json* values can be injected directly into a view.

Example of an *appsettings.json* file:

JSONCopy

{

"root": {

"parent": {

"child": "myvalue"

}

}

}

The syntax for @inject: @inject <type> <name>

An example using @inject:

C#Copy

@using Microsoft.Extensions.Configuration

@inject IConfiguration Configuration

@{

string myValue = Configuration["root:parent:child"];

...

}

**Service injection**

A service can be injected into a view using the @inject directive. You can think of @inject as adding a property to the view, and populating the property using DI.

CSHTMLCopy

@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>

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 *Startup.cs*:

C#Copy

// For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?LinkID=398940

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

services.AddTransient<IToDoItemRepository, ToDoItemRepository>();

services.AddTransient<StatisticsService>();

services.AddTransient<ProfileOptionsService>();

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

C#Copy

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);

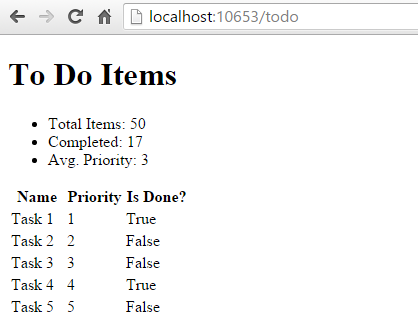
}

}

}

The sample repository uses an in-memory collection. The implementation shown above (which operates on all of the data in memory) isn't recommended for large, remotely accessed data sets.

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



**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 MVC approach would require the controller to request data access services for each of these sets of options, and then 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, moving this view element construction logic into the view itself. The controller action to display a profile editing form only needs to pass the form the profile instance:

C#Copy

using Microsoft.AspNetCore.Mvc;

using ViewInjectSample.Model;

namespace ViewInjectSample.Controllers

{

public class ProfileController : Controller

{

[Route("Profile")]

public IActionResult Index()

{

// TODO: look up profile based on logged-in user

var profile = new Profile()

{

Name = "Steve",

FavColor = "Blue",

Gender = "Male",

State = new State("Ohio","OH")

};

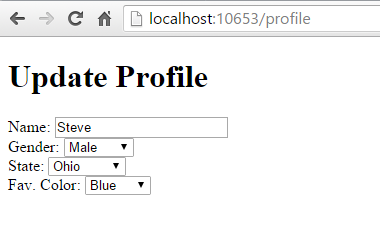
return View(profile);

}

}

}

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



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

CSHTMLCopy

@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>

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

C#Copy

using System.Collections.Generic;

namespace ViewInjectSample.Model.Services

{

public class ProfileOptionsService

{

public List<string> ListGenders()

{

// keeping this simple

return new List<string>() {"Female", "Male"};

}

public List<State> ListStates()

{

// a few states from USA

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" };

}

}

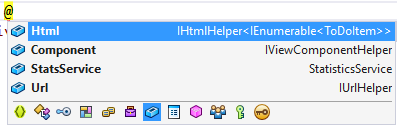
}

**Important**

Don't forget to register types you request through dependency injection in Startup.ConfigureServices. An unregistered type throws an exception at runtime because the service provider is internally queried via **[GetRequiredService](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.serviceproviderserviceextensions.getrequiredservice)**.

**Overriding Services**

In addition to injecting new services, this technique can also be used to override previously injected services on a page. The figure below shows all of the fields available on the page used in the first example:



As you can see, the default fields include Html, Component, and Url (as well as the StatsService that we injected). If for instance you wanted to replace the default HTML Helpers with your own, you could easily do so using @inject:

CSHTMLCopy

@using System.Threading.Tasks

@using ViewInjectSample.Helpers

@inject MyHtmlHelper Html

<!DOCTYPE html>

<html>

<head>

<title>My Helper</title>

</head>

<body>

<div>

Test: @Html.Value

</div>

</body>

</html>

If you want to extend existing services, you can simply use this technique while inheriting from or wrapping the existing implementation with your own.