View components in ASP.NET Core

12/18/2019 • 11 minutes to read • +16

In this article

By Rick Anderson

View or download sample code (how to download)

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) 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 Create and use ASP.NET Core Razor components.

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

- Doesn't take part in the controller lifecycle, which means you can't use filters 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 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

```csharp
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:

CSHTML

```
@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:

CSHTML

```
@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:

CSHTML

```
<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. The Tag Helper to invoke a view component uses the `<vc></vc>` element. The view component is specified as follows:

CSHTML

```
<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:

CSHTML

```
@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 for more information on how to register Tag Helpers.

The `InvokeAsync` method used in this tutorial:

CSHTML

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

In Tag Helper markup:

CSHTML

```
<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`.

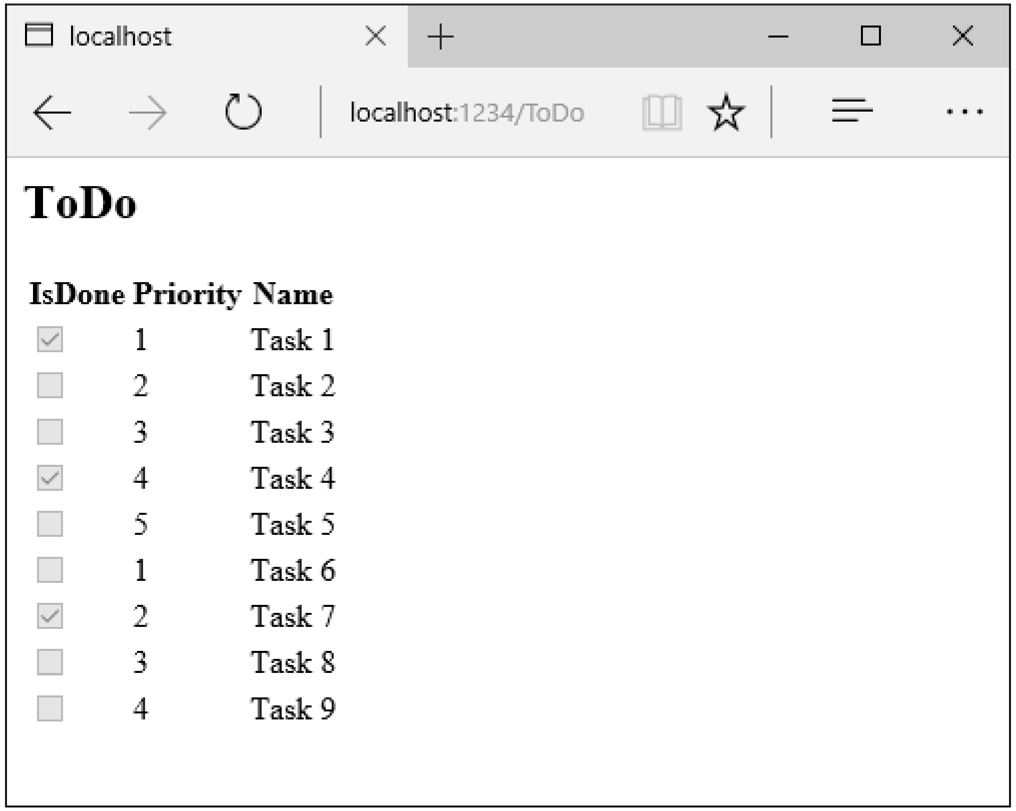
In this example, the view component is called directly from the controller:

C# Copy

```csharp
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```

Walkthrough: Creating a simple view component

Download, build and test the starter code. It's a simple project with a `ToDo` controller that displays a list of *ToDo* items.



ToDo

IsDone	Priority	Name
✓	1	Task 1
☐	2	Task 2
☐	3	Task 3
✓	4	Task 4
☐	5	Task 5
☐	1	Task 6
✓	2	Task 7
☐	3	Task 8
☐	4	Task 9

Add a ViewComponent class

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

C#

```csharp
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 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:

CSHTML Copy

```
@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*).

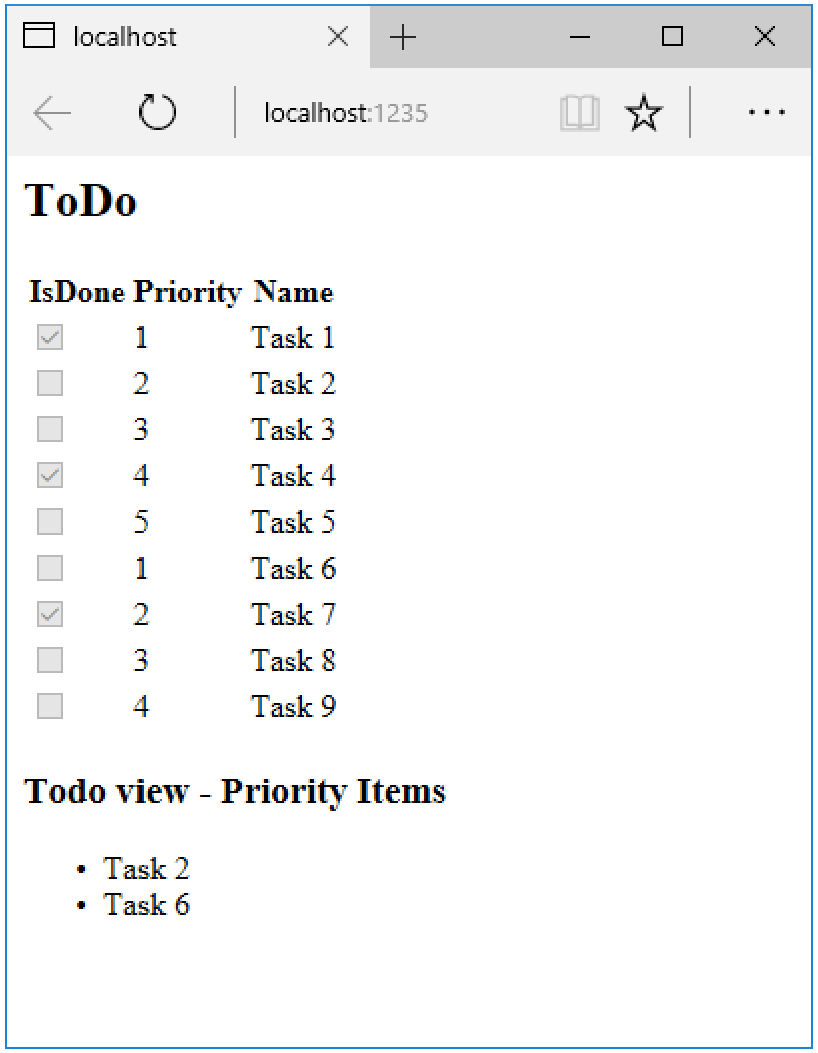
- Add a `div` containing a call to the priority list component to the bottom of the *Views/ToDo/index.cshtml* file:

CSHTML

```
</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:



ToDo

IsDone	Priority	Name
✓	1	Task 1
	2	Task 2
	3	Task 3
✓	4	Task 4
	5	Task 5
	1	Task 6
✓	2	Task 7
	3	Task 8
	4	Task 9

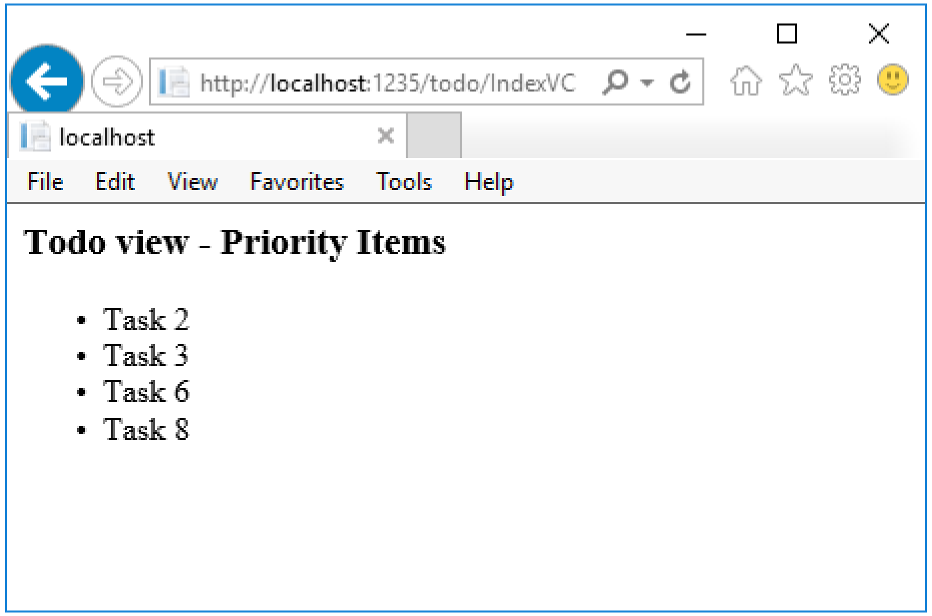
Todo view - Priority Items

- Task 2
- Task 6

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

C# Copy

```csharp
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```


http://localhost:1235/todo/IndexVC
localhost
File Edit View Favorites Tools Help

Todo view - Priority Items

- Task 2
- Task 3
- Task 6
- Task 8

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#

```csharp
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.

CSHTML

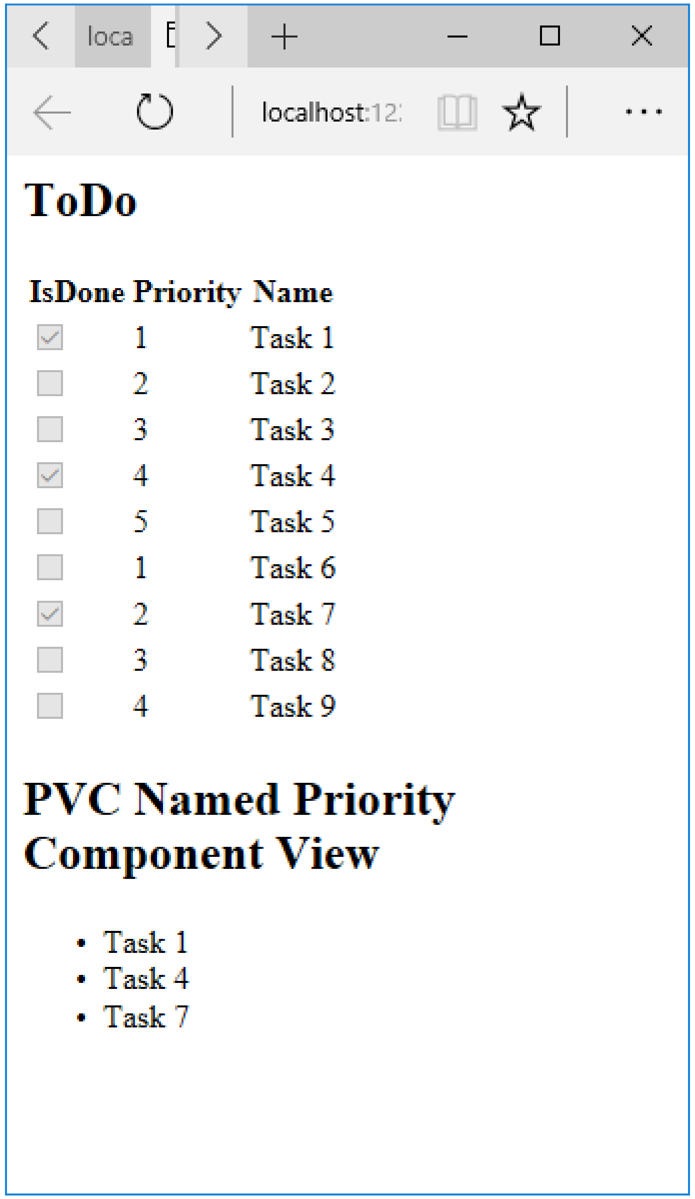
```
@model IEnumerable<ViewComponentSample.Models.TodoItem>

<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*:

CSHTML

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

Run the app and verify PVC view.



ToDo

IsDone	Priority	Name
✓	1	Task 1
☐	2	Task 2
☐	3	Task 3
✓	4	Task 4
☐	5	Task 5
☐	1	Task 6
✓	2	Task 7
☐	3	Task 8
☐	4	Task 9

PVC Named Priority Component View

- Task 1
- Task 4
- Task 7

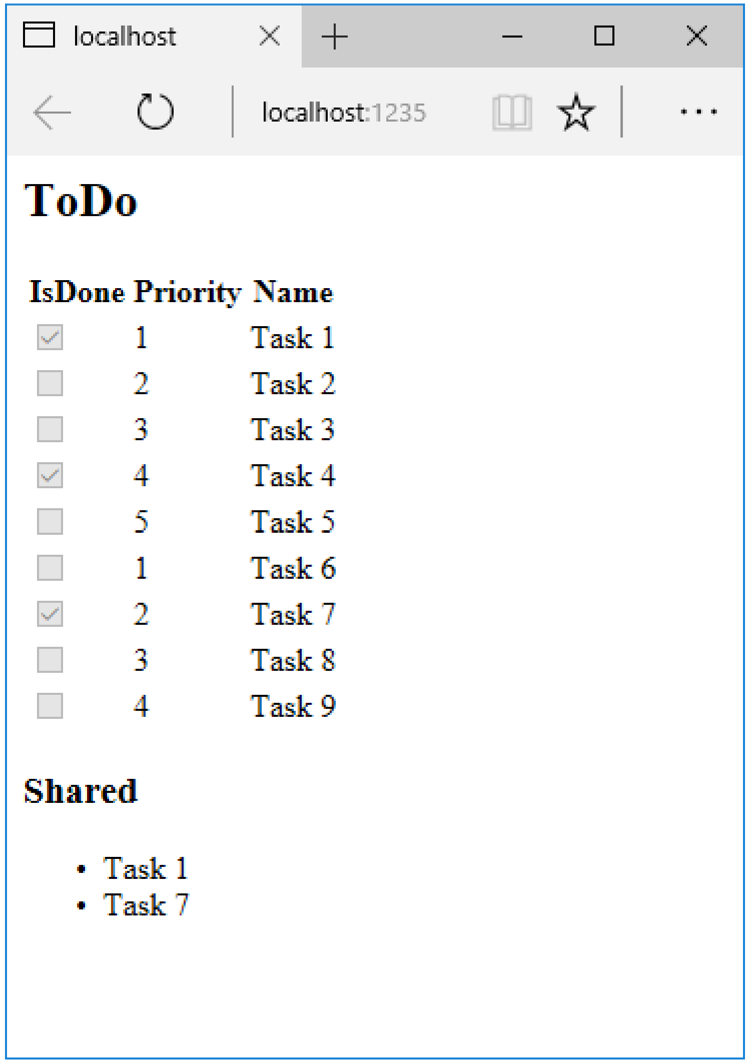
If the PVC view isn't rendered, verify you are calling the view component with a priority of 4 or higher.

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:

```
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.



ToDo

IsDone	Priority	Name
✓	1	Task 1
☐	2	Task 2
☐	3	Task 3
✓	4	Task 4
☐	5	Task 5
☐	1	Task 6
✓	2	Task 7
☐	3	Task 8
☐	4	Task 9

Shared

- Task 1
- Task 7

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:

CSHTML Copy

```
@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

```csharp
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*):

CSHTML Copy

```
@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
- Tag Helper

To use the IViewComponentHelper approach, call `Component.InvokeAsync`:

CSHTML Copy

```
@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`):

CSHTML Copy

```
@addTagHelper *, MyWebApp
```

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

CSHTML Copy

```cshtml
<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. 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.

Additional resources

- Dependency injection into views

Is this page helpful?

Yes No