View Components in ASP.NET Core MVC

Posted May 9, 2019 by Wolfgang Ofner

View components are a new feature in ASP.NET Core MVC which replaces the child action feature from the previous version. View components are classes which provide action-style logic to support partial views. This means that complex content can be embedded in views with C# code which can be easily maintained and unit tested. You can find the source code for the following demo on GitHub.

Understanding View Components

Applications commonly need to embed content in views that aren't related to the main purpose of the application. Common examples include site navigation tools and authentication panels that let the user login without visiting a separate page. The common thread that all these examples have is that the data required to display the embedded content isn't part of the model data passed from the action to the view.

Partial views are used to create reusable markup that is required in views, avoiding the need to duplicate the same content in multiple places in the application. Partial views are a useful feature, but they just contain fragments of HTML and Razor, and the data they operate on is received from the parent view. If you need to display different data, then you run into a problem. You could access the data you need directly from the partial view, but this breaks the separation of concerns that underpins the MVC pattern and results in data retrieval and processing logic being placed in a view file.

Alternatively, you could extend the view models used by the application so that it includes the data you require, but this means you have to change every action method and it is hard to isolate the functionality of action methods for effective testing.

This is where view components come in. A view component is a C# class that provides a partial view with the data that it needs, independently from the parent view and the action that renders it. In this regard, a view component can be thought of as a specialized action, but one that is used only to provide a partial view with data. It can't receive HTTP requests, and the content that it provides will always be included in the parent view.

Creating a View Component

View components can be created in three different ways which are:

- defining a Poco view component
- deriving from the ViewComponent base class
- using the ViewComponent attribute

Creating Poco View Components

A Poco view component is a class that provides view component functionality without relying on any of the MVC APIs. As with Poco controllers, the kind of view component is awkward to work with but can be helpful in understanding how they work. A Poco view component is any class whose name ends with ViewComponent and that defines an Invoke method. VIew component classes can be defined anywhere in an application, but the convention is to group them together in a folder called Components at the root level of the project.

```
public class PocoViewComponent
{
    private readonly ICountryRepository _countryRepository;

    0 references | 0 exceptions
    public PocoViewComponent(ICountryRepository countryRepository)
    {
        _countryRepository = countryRepository;
    }

    0 references | 0 exceptions
    public string Invoke()
    {
        return $"{_countryRepository.Countries().Count()} counties with a " +
               $"population of {_countryRepository.Countries().Sum(x => x.Population)}";
    }
}
```

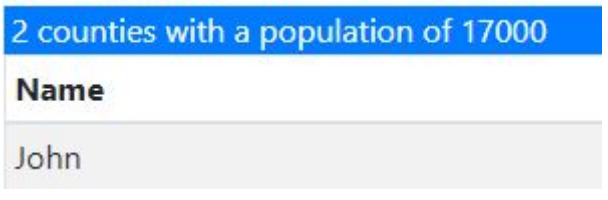
The Poco view component

You can add your view component to a view by calling the invoke method and passing the view component name:

```
<div class="col-12 bg-primary">
    <div class="row text-white pl-1">
        @await Component.InvokeAsync("Poco")
    </div>
</div>
```

Calling the VIewComponent from the view

This code adds the number of countries and its total population on top of the customer table.



Testing the view component

I know that this is a very simple example but I think it demonstrates how view components work very well.

First, the PocoViewComponent class was able to get access to the data it required without depending on the action handling the HTTP request or its parent view. Second, defining the logic required to obtain and process the country summary in a C# class which is easily readable and can also be unit tested. Third, the application hasn't been twisted out of shape by trying to include country objects in view models that are focused on Customer objects. In short, a view component is a self-contained chunk of reusable functionality that can be applied throughout the application and can be developed and tested in isolation.

Note that you have to include the await keyword. Otherwise, you won't see an error but only a string representation of a Task will be displayed.

Deriving from the ViewComponent Base Class

Poco view components are limited in functionality unless they take advantage of the MVC API, which is possible but requires a lot more effort than the more common approach, deriving from the ViewComponent class. Deriving from the base class gives you access to context data and makes it easier to generate results. When you create a derived view component, you don't have to put ViewComponent in the name.

```
<div class="col-12 bg-success">
    <div class="row text-white pl-1">
        @await Component.InvokeAsync("Derived")
    </div>
```

Calling the derived view component from the view

Understanding View Component Results

The ability to insert simple values into a parent view isn't especially useful, but fortunately, view components are capable of much more. More complex effects can be achieved by having the Invoke method return an object that implements the IViewComponentResult interface. There are three built-in classes that implement the IViewComponentResult interface:

Name	Description
ViewViewComponentResult	This class is used to specify a Razor view, with optional view model data. Instances of this class are created using the View method.
ContentViewComponentResult	This class is used to specify a text result that will be safely encoded for inclusion in an HTML document. Instances of this class are created using the Content method.
HtmlContentViewComponentResult	This class is used to specify a fragment of HTML that will be included in the HTML document without further encoding. There is no ViewComponent method to create this type of result

There is special handling for two result types. If a view component returns a string, then it is used to create a ContentViewComponentResult object. If a view component returns an IHtmlContent object, then it is used to create an HtmlContentViewComponentResult object.

Return a Partial View

The most useful response is the awkwardly named ViewViewComponentResult object, which tells Razor to render a partial view and includes the result in the parent view. The ViewComponent base class provides the View method for creating ViewViewComponentResult objects, and there are the following four versions of the method available:

Name	Description
View()	Using this method selects the default view for the view component and does not provide a view model
View(model)	Using the method selects the default view and uses the specified object as the view model.
View(viewName)	Using this method selects the specified view and does not provide a view model.
View(viewName, model)	Using this method selects the specified view and uses the specified object as the view model.

These methods correspond to those provided by the Controller base class and are used in much the same way.

```
public class DerivedViewComponent : ViewComponent
{
    private readonly ICountryRepository _countryRepository;

    0 references | 0 exceptions
    public DerivedViewComponent(ICountryRepository countryRepository)
    {
        _countryRepository = countryRepository;
    }

    0 references | 0 exceptions
    public IViewComponentResult Invoke()
    {
        return View(new CountryViewModel
        {
            Countries = _countryRepository.Countries().Count(),
            Population = _countryRepository.Countries().Sum(x => x.Population)
        });
    }
}
```

The derived view component returning an IViewComponentResult object

Selecting a partial view in a view component is similar to selecting a view in a controller but without two important differences: Razor looks for views in different locations and uses a different default view name if none is specified. Razor is looking for the partial view in the following locations:

- /Views/Home/Components/Derived/Default.cshtml

- /Views/Shared/Components/Derived/Default.cshtml
- /Pages/Shared/Components/Derived/Default.cshtml

If no name is specified, then Razor looks for a file called Default.cshtml. Razor looks in two locations for the partial view. The first location takes into account the name of the controller handling the HTTP request, which allows each controller to have its own custom view. The second location is shared between all controllers.

```
@model CountryViewModel

<table class="table table-sm table-bordered">
    <tr>
        <td>Countries:</td>
        <td>
            @Model.Countries
        </td>
    </tr>
    <tr>
        <td>Population:</td>
        <td>
            @Model.Population.ToString("#,###")
        </td>
    </tr>
</table>
```

View for the view component

Start the application and you will see the table, produced by the view component in green on top of the customer table.

2 counties with a population of 17000	
Countries:	2
Population:	17,000

Name	Age
John	30

The rendered view from the view component

Returning HTML Fragments

The ContentViewComponentResult class is used to include fragments of HTML in the parent view without using a view. Instances of the ContentViewComponentResult class are created using the Content method inherited from the ViewComponent base class, which accepts a string value. In addition to the Content method, the Invoke method can return a string, and MVC will automatically convert to a ContentViewComponent.

```
public class ReturningHtml : ViewComponent
{
    0 references | 0 exceptions
    public IViewComponentResult Invoke()
    {
        return new HtmlContentViewComponentResult(
            new HtmlString("This code was created in the ReturningHtml view component"));
    }
}
```

View component which returns HTML

Getting Context Data

Details about the current request and the parent view are provided to a view component through properties of the ViewComponentContext class. The following table shows the available properties:

Name	Description
Arguments	This property returns a dictionary of the arguments provided by the view, which can also be received via the Invoke method.

HtmlEncoder	This property returns an HtmlEncoder object that can be used to safely encode HTML fragments.
ViewComponentDescriptor	This property returns a ViewComponentDescriptor, which provides a description of the view component.
ViewContext	This property returns the ViewContext object from the parent view.
ViewData	This property returns a ViewDataDictionary, which provides access to the view data provided for the view component.

ViewComponent base class Properties

The ViewComponent base class provides a set of convenience properties that make it easier to access specific context information:

Name	Description
ViewComponentContext	This property returns the ViewComponentContext object.
HttpContext	This property returns an HttpContext object that describes the current request and the response that is being prepared.
Request	This property returns an HttpRequest object that describes the current HTTP request.
User	This property returns an IPrincipal object that describes the current user.
RouteData	This property returns a RouteData object that describes the routing data for the current request.
ViewBag	This property returns the dynamic view bag object, which can be used to pass data between the view component and the view.
ModelState	This property returns a ModelStateDictionary, which provides details of the model binding process.
ViewContext	This property returns the ViewContext object that was provided to the parent view.
ViewData	This property returns a ViewDataDictionary, which provides access to the view data provided for the view component.
Url	This property returns an IUrlHelper object that can be used to generate URLs.

The context data can be used in whatever way it helps the view component to its work, including varying the way that data is selected or rendering different content or views. To demonstrate this feature, I changed my view component to read the route data and if there is a value for id, I return a message.

```
public IViewComponentResult Invoke()
{
    var id = RouteData.Values["id"] as string;

    if (!string.IsNullOrWhiteSpace(id))
    {
        return Content($"The entered id is {id}");
    }

    return View(new CountryViewModel
    {
        Countries = _countryRepository.Countries().Count(),
        Population = _countryRepository.Countries().Sum(x => x.Population)
    });
}
```

Reading RouteData in the view component

If I call my action with a value for the id parameter, the string is returned instead of the countries and population.

Changing the output of the view component, depending on the route data

Providing Context from the Parent View using Arguments

Parent views can provide additional context data as arguments in the Component.Invoke expression. This feature can be used to provide data from the parent view model or to give guidance about the type of content that the view component should produce. I extended my view component to take a bool parameter. If this parameter is true, I return a message, indicating that the parameter was received and processed.

```
public IViewComponentResult Invoke(bool useParameter)
{
    if (useParameter)
    {
        return Content("Doing something with the value provided from the parent view");
    }
```

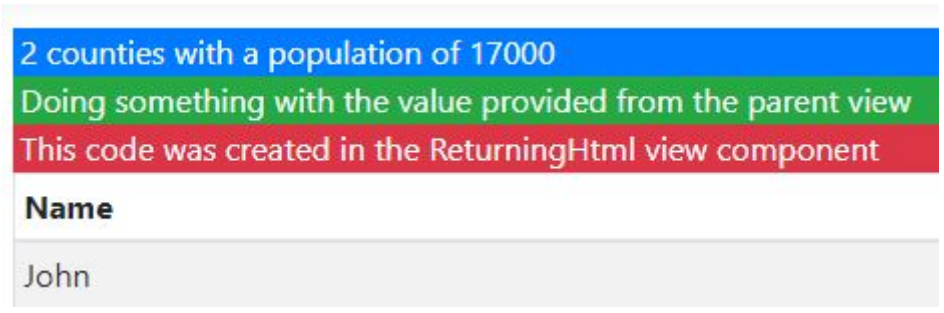
Working with the parameter in the view component

Next, I change the main view to provide a parameter for my view component.

```
<div class="col-12 bg-success">
    <div class="row text-white pl-1">
        @await Component.InvokeAsync("Derived", new {useParameter = true})
    </div>
</div>
```

Providing an argument to the view component

Start the application and you will see that the previously defined string will be returned to the view.



Changing the output of the view component, depending on the provided parameter

Unit Testing View Components

View components can be unit tested like every other C# class. I created a new test project and implemented a simple test to demonstrate how to test the Invoke method.

```
public class DerivedViewComponentTests
{
    0 references | 0 exceptions
    public DerivedViewComponentTests()
    {
        _testee = new DerivedViewComponent(A.Fake<ICountryRepository>());
    }

    private readonly DerivedViewComponent _testee;

    [Fact]
    0 references | 0 exceptions
    public void Invoke_WhenParameterIsProvided_ShouldReturnContentString()
    {
        var result = _testee.Invoke(true);

        ((ContentViewComponentResult)result).Content.Should().Be
            ("Doing something with the value provided from the parent view");
    }
}
```

Unit testing the view component

Creating Asynchronous View Components

All view components so far were executed synchronously. You can create an asynchronous view component by defining an InvokeAsync method that returns a Task. When Razor receives the Task from the InvokeAsync method, it will wait for it to complete and then insert the result into the main view.

```
public class Async : ViewComponent
{
    0 references | 0 exceptions
    public async Task<IViewComponentResult> InvokeAsync()
    {
        const string website = "https://www.google.com";
        var client = new HttpClient();
        var response = await client.GetAsync(website);

        return Content($"The size of {website} is " +
                       $"{response.Content.Headers.ContentLength.ToString()} bytes");
    }
}
```

Creating an async view component

Hybrid Controller / View Component Classes

View components often provide basic functionality to enhance the current view with additional data. If you have to do more complex operations, you can create a class that is a controller and a view component. This allows for related functionality to be grouped together and reduces code duplication.

```
[ViewComponent(Name = "HybridComponent")]
1 reference | 0 changes | 0 authors, 0 changes | 0 requests
public class CountryController : Controller
{
    private readonly ICountryRepository _countryRepository;

    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public CountryController(ICountryRepository countryRepository)
    {
        _countryRepository = countryRepository;
    }

    0 references | 0 changes | 0 authors, 0 changes | 0 requests | 0 exceptions
    public IActionResult Create()
    {
        return View();
    }

    [HttpPost]
    0 references | 0 changes | 0 authors, 0 changes | 0 requests | 0 exceptions
    public IActionResult Create(Country country)
    {
        _countryRepository.AddCountry(country);

        return View("Index", "Home");
    }

    0 references | 0 changes | 0 authors, 0 changes | 0 requests | 0 exceptions
    public IViewComponentResult Invoke()
    {
        return new ViewViewComponentResult()
        {
            ViewData = new ViewDataDictionary<IEnumerable<Country>>(ViewData,
            _countryRepository.Countries())
        };
    }
}
```

The hybrid controller – view component

The ViewComponent attribute is applied to classes that don't inherit from the ViewCompnent base class and whose name doesn't end with ViewComponent, meaning that the normal discovery process wouldn't normally categorize the class as view component. The Name property sets the name by which the class can be referred to when applying the class using the @Component.Invoke expression in the parent view.

Since hybrid classes don't inherit from the ViewComponent base class, they don't have access to the convenience methods for creating IViewComponentResult object, which means that I have to create the ViewViewComponentResult object directly.

Creating Hybrid View

A hybrid class requires two sets of views: those that are rendered when the class is used as a controller and those that are rendered when the class is used as a view component. First, I add the view for the controller under Views/Country/Create.cshtml

```
@model Country

@{
    ViewData["Title"] = "Create a Country";
    Layout = "_Layout";
}

<form method="post" asp-action="Create">
    <div class="form-group">
        <label asp-for="Name">Name:</label>
        <input class="form-control" asp-for="Name" />
    </div>
    <div class="form-group">
        <label asp-for="Population">Population:</label>
        <input class="form-control" asp-for="Population" />
    </div>
    <button type="submit" class="btn btn-primary">Create</button>
    <a class="btn btn-secondary" asp-controller="Home"
      asp-action="Index">
       Cancel
    </a>
</form>
```

The view for the controller

Next, I add a the partial view under Views/Shared/Components/HybridComponent/Default.cshml.

```
@model IEnumerable<Country>

<table class="table table-sm table-bordered">
    <tr>
        <td>Biggest Country:</td>
        <td>
            @Model.OrderByDescending(c => c.Population).First().Name
        </td>
    </tr>
</table>
<a class="btn btn-sm btn-info" asp-controller="Country" asp-action="Create">
    Create Country
</a>
```

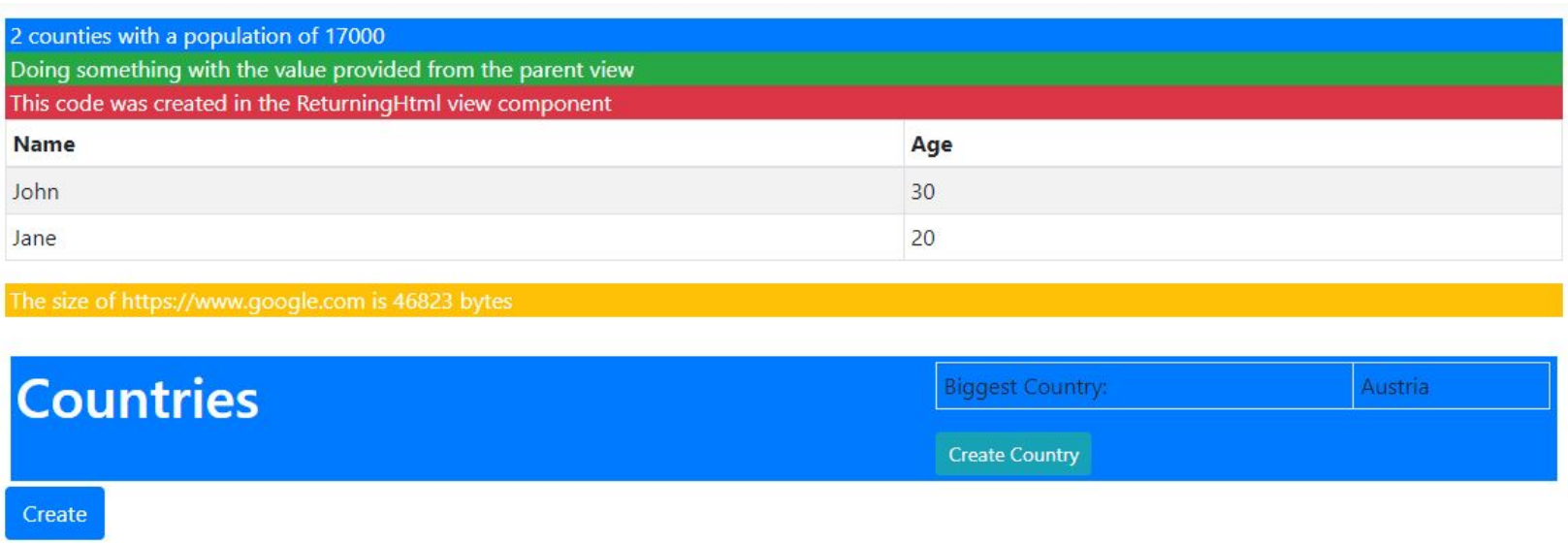
Partial view for the hybrid controller

Lastly, I invoke my hybrid controller from the view with the previously applied name.

```
<div class="bg-primary m-1 p-1">
    <div class="row text-white">
        <div class="col-7"><h1>Countries</h1></div>
        <div class="col-5">
            @await Component.InvokeAsync("HybridComponent")
        </div>
    </div>
</div>
```

Invoking the hybrid controller – view component

This renders the (super ugly) form.



The rendered view of the hybrid controller

Conclusion

Today, I talked about view components and how they can help you extending your views and introducing more functionality and more information for the users. You can call them synchronously or asynchronously and also easily unit test them. Lastly, I showed how hybrid controller / view components can be used to group more complex code together.

For more details about the configuring ASP.NET Core, I highly recommend the book “Pro ASP.NET Core MVC 2“. You can find the source code for this demo on GitHub.

ASP.NET

NET Core ASP.NET Core MVC C# View Component

Share:

Further Reading

Apr 29, 2019

Getting to know the Startup Class of ASP.Net Core MVC

Every .NET Core web application has a Program class with a static Main method.…

May 1, 2019

Middleware in ASP.NET Core MVC

Middleware is the term used for the components that are combined to form t…

May 3, 2019

Configure ASP.NET Core MVC

Some configurations like the connection string usually change on every…

OLDER
Dependency Injection in ASP.NET Core MVC

NEWER
Understanding Tag Helpers in ASP.NET Core MVC