The Learning House
427 S 4th Street #300
Louisville KY 40202

View Organization

.NET Cohort

Shared Layouts

Oftentimes, we want to maintain a consistent look and feel across all of the pages in our websites . Good examples of this are common headers and footers.

Let’s see how to wire this up from scratch.

Add View Dialogue

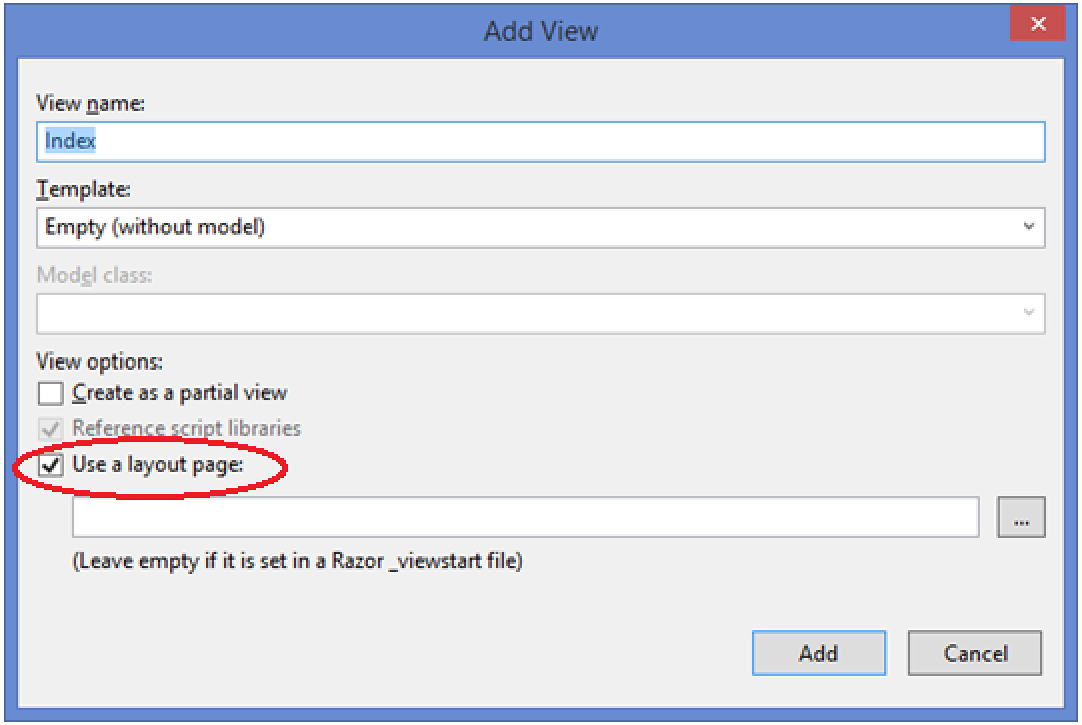
In the Add View Dialogue, we have an option to use a layout page.

If we check the box, it will generate a portion of HTML and expect the layout page to wrap the view with common layout.

If we uncheck the box, it will create a full HTML page with head and body tags.

Optionally, you can hit the ellipsis button and select a specific layout page.

If you leave it empty, it will look at a file in Views called _viewstart where the default layout page is set (typically it is _Layout).



Add View

View name: Index

Template: Empty (without model)

Model class:

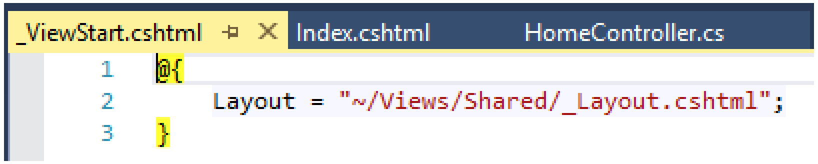
View options:

□ Create as a partial view

☑ Reference script libraries

☑ Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)



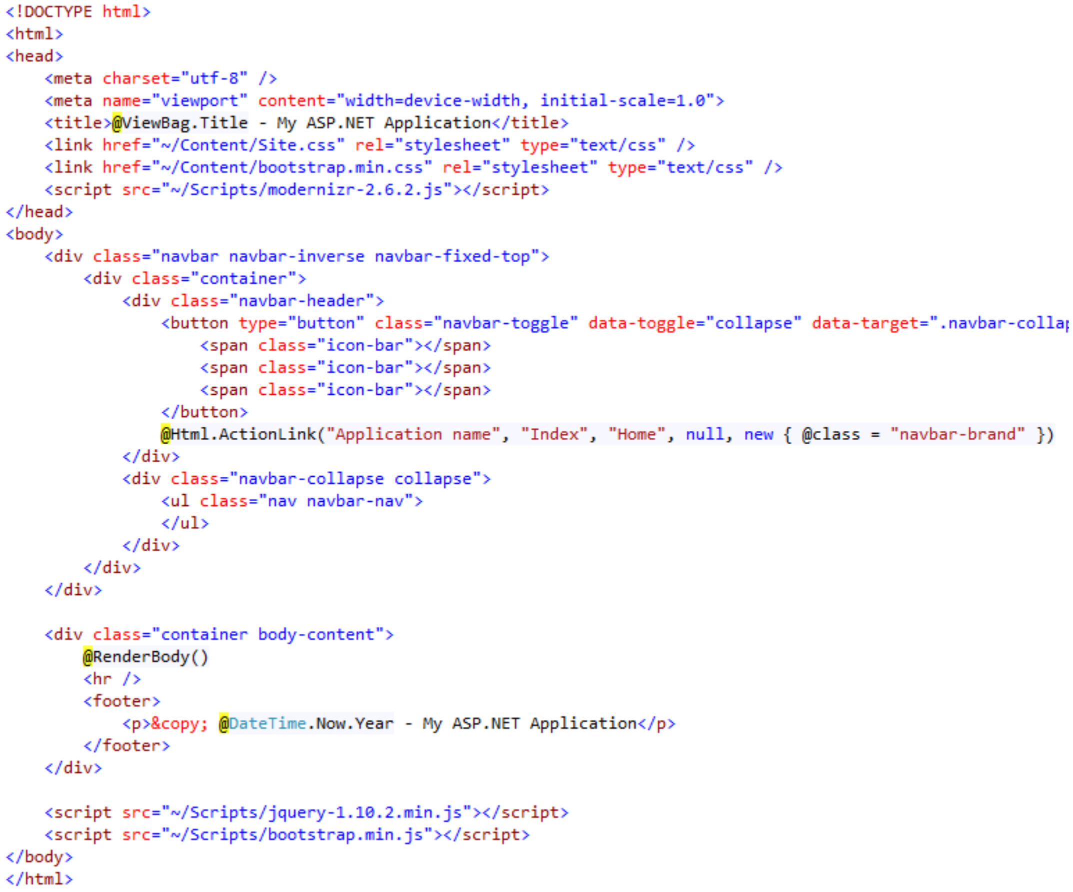
```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

_Layout.cshtml

Any file can be a shared layout page, but _Layout.cshtml is the default that is set in the _viewstart.cshtml file.

The convention in MVC is that any shared views should start with an underscore. It is not required, just encouraged.

Notice that Visual Studio set us up with a bootstrap-enabled layout form with a menu and footer. The CSS is linked into the head and jquery/bootsrap JavaScript is linked in at the bottom, following best practices.

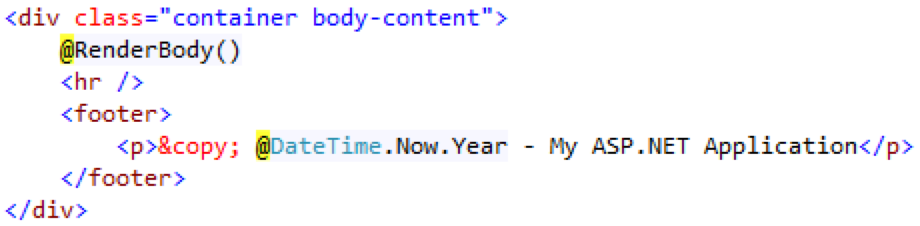


```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    <link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" type="text/css" />
    <script src="~/Scripts/modernizr-2.6.2.js"></script>
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-colla
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", null, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                </ul>
            </div>
        </div>
    </div>

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

    <script src="~/Scripts/jquery-1.10.2.min.js"></script>
    <script src="~/Scripts/bootstrap.min.js"></script>
</body>
</html>
```

@RenderBody()

The @RenderBody() method in a layout file shows the Razor engine where to inject the specific view we are rendering.

Here, our Index.cshtml view will be put inside the <div class=“container body=content”> tag. A horizontal rule and footer will be placed below it.

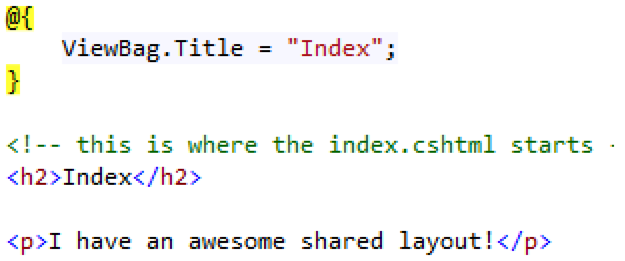


```
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
    </footer>
</div>
```

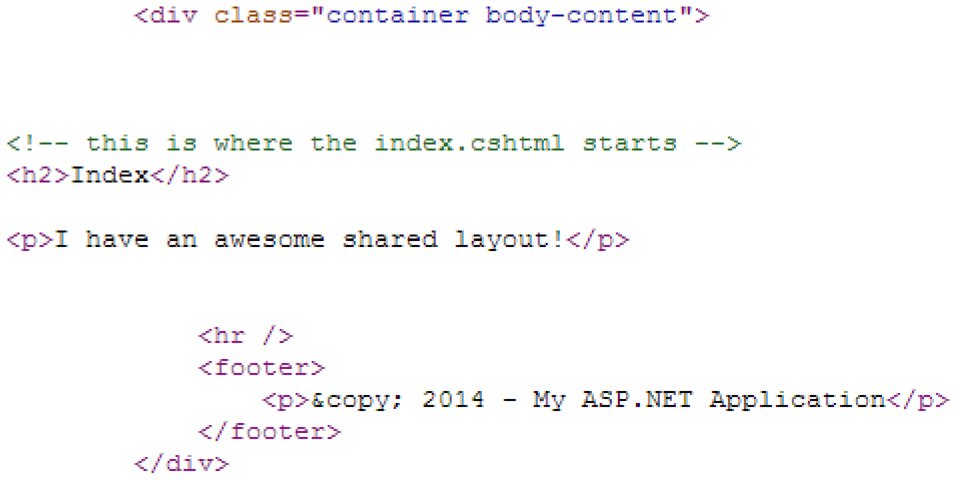
@RenderBody() in action

Notice that our Index.cshtml (view) up top gets injected into the _Layout page where the @RenderBody() command existed.

Also note that it takes all vertical spacing and tabs as literal. Since we didn't tab over the HTML in the Index view, it is not tabbed over in the rendered output.



```
@{
    ViewBag.Title = "Index";
}

<!-- this is where the index.cshtml starts -
<h2>Index</h2>

<p>I have an awesome shared layout!</p>
```



```
        <div class="container body-content">



<!-- this is where the index.cshtml starts -->
<h2>Index</h2>

<p>I have an awesome shared layout!</p>

            <hr />
            <footer>
                <p>&copy; 2014 - My ASP.NET Application</p>
            </footer>
        </div>
```

@RenderSection()

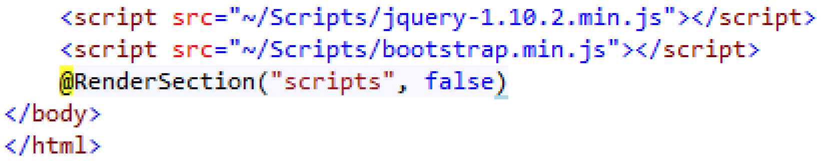
Often, we want to set aside parts of the layout page to be used by the view that is being rendered in the RenderBody().

One of the most frequent cases is when we want to put in JavaScript that is specific to the rendered view at the bottom.

Sections can be required or optional.

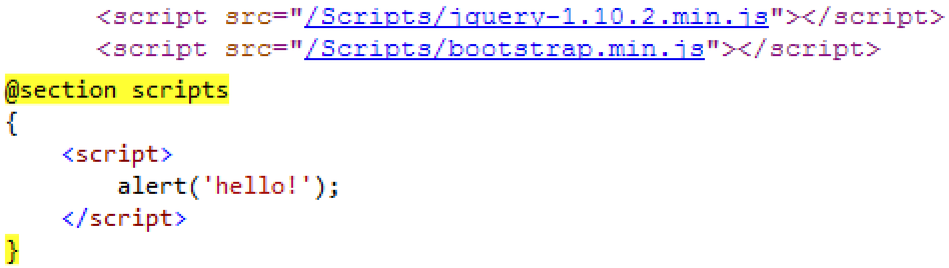
Declaring a Section

Let’s add a section for custom JavaScript to our _Layout view after the other scripts. It is optional — not every view will have its own custom JavaScript.



```
    <script src="~/Scripts/jquery-1.10.2.min.js"></script>
    <script src="~/Scripts/bootstrap.min.js"></script>
    @RenderSection("scripts", false)
</body>
</html>
```

We declare the section in our Index.cshtml view, calling it by name. The @section call can be anywhere in the file. Anything in the code block { } will be placed at the @RenderSection command in the layout.



```
    <script src="/Scripts/jquery-1.10.2.min.js"></script>
    <script src="/Scripts/bootstrap.min.js"></script>
@section scripts
{
    <script>
        alert('hello!');
    </script>
}
```

Again, it is common to declare sections for custom CSS, custom JavaScript, and custom meta tags in the header.

Partial Views

Partial view is designed for reusing snippets of HTML in our code. It behaves like a regular view and you can pass it model data if you like.

Typically, it is used when you have a common bit of code reuse (special headers, footers, detail views of models, etc.).

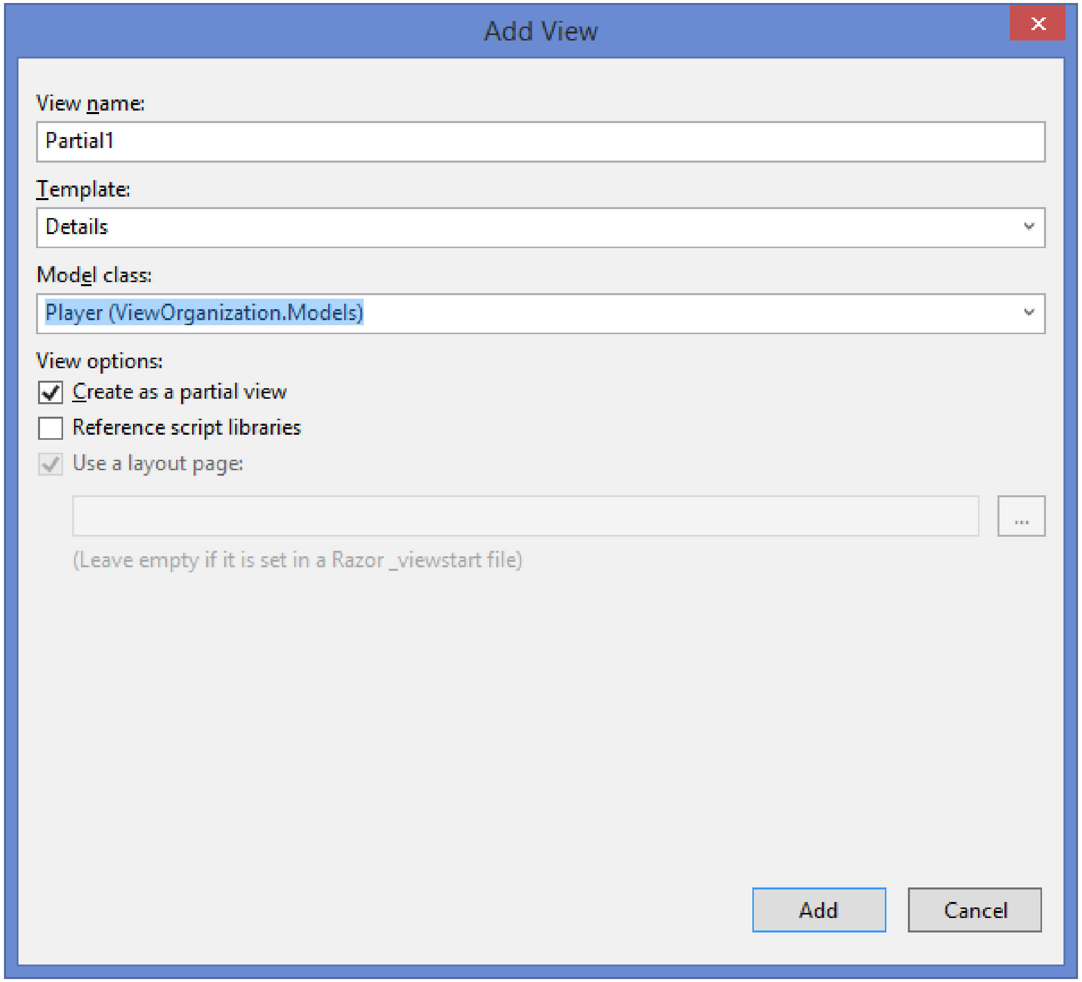
Creating a partial view

Let's pretend that we have a block of HTML that represents an NFL player.

This detail block will be used on the roster page and the box score page, but we only want to write the HTML once and then pass a player into it.

If we have a Player model that has the name, position, and number, we can check the box to Create as a partial view.

Add a partial to the Views/Shared folder and name it _Player.



Add View

View name: Partial1

Template: Details

Model class: Player (ViewOrganization.Models)

View options:

☑ Create as a partial view

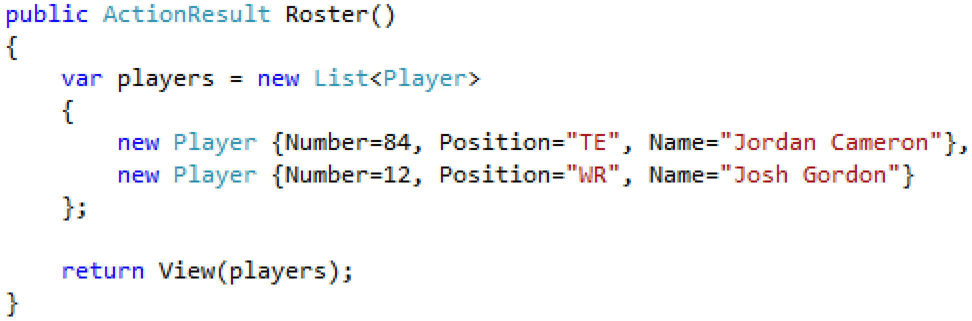
□ Reference script libraries

☑ Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

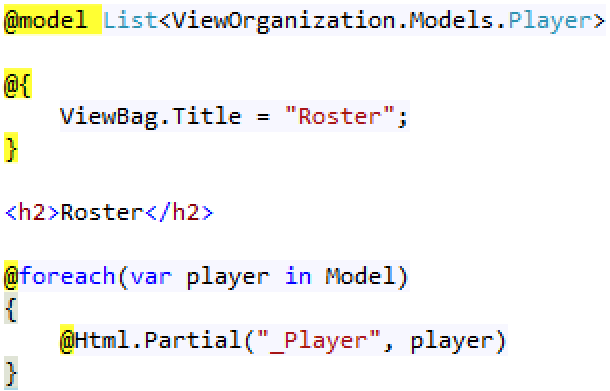
Using a partial view

Let’s create a Roster action on our home controller and pass the view a list of players. We can then loop the list and render the partial with data, like so.



```
public ActionResult Roster()
{
    var players = new List<Player>
    {
        new Player {Number=84, Position="TE", Name="Jordan Cameron"},
        new Player {Number=12, Position="WR", Name="Josh Gordon"}
    };

    return View(players);
}
```

@Html.Partial looks in the shared folder by default, so if you want to store your partials elsewhere, you have to put the full path.



```
@model List<ViewOrganization.Models.Player>

@{
    ViewBag.Title = "Roster";
}

<h2>Roster</h2>

@foreach(var player in Model)
{
    @Html.Partial("_Player", player)
}
```

ex:
~/Views/Folder/Partial.cshtml

The tilde ~ always means “root.”

Gut Check

- In what file will our “site wrap” content normally live, by default?
  - Why? Where is that controlled?
- In a layout view, how do we tell the view where to render the specific page’s content?
- How do we divide up a large view into separate, manageable chunks?