Microsoft

Programming ASP.NET Core

Professional

Dino Esposito

Microsoft

Programming ASP.NET Core

Dino Esposito

The Razor View Engine

In ASP.NET Core, a view engine is merely a class that implements a fixed interface—the *IViewEngine* interface. Each application can have one or more view engines and use all of them in different cases. In ASP.NET Core, however, each application is armed by just one default view engine—the *RazorViewEngine* class. The aspect of the view engine that most impacts the development is the syntax it supports for defining the template of the view.

The Razor syntax is quite clean and friendly. A view template is essentially an HTML page with a few code placeholders. Each placeholder contains an executable expression—much like a code snippet. The code in the snippets is evaluated when the view gets rendered, and the resulting markup is integrated into the HTML template. Code snippets can be written in C# or other .NET languages supported by the .NET Core platform.

Note It is possible to implement your own view engine based on your custom syntax in addition to the *RazorViewEngine* class provided by ASP.NET Core.

Generalities of the Razor View Engine

The Razor view engine reads templates from a physical location on disk. Any ASP.NET Core project has a root folder named *Views* where the templates are stored in a specific structure of subdirectories. The *Views* folder usually has some subfolders—each named after an existing controller. Each controller-specific subdirectory contains physical files whose name is expected to match the name of an action. The extension has to be *.cshtml* for the Razor view engine. (If you're writing your ASP.NET Core application in, say, Visual Basic, then the extension must be *.vbhtml*.)

ASP.NET MVC requires that you place each view template under the directory named from the controller that uses it. In case multiple controllers are expected to invoke the same view, then you move the view template file under the *Shared* folder.

It is important to note that the same hierarchy of directories that exists at the project level under the *Views* folder must be replicated on the production server when you deploy the site.

View Location Formats

The Razor view engine defines a few properties through which you can control how view templates are located. For the internal working of the Razor view engine, it is necessary to provide a default location for master, regular, and partial views both in a default project configuration and when areas are used.

Table 5-1 shows the location properties supported by the Razor view engine with the predefined value. The *AreaViewLocationFormats* property is a list of strings, each of which points to a placeholder string defining a virtual path. Also, the *ViewLocationFormats* property is a list of strings, and each of its contained strings refers to a valid virtual path for the view template.

TABLE 5-1 The default location formats of the Razor view engine

Property	Default location format
AreaViewLocationFormats	~/Areas/{2}/Views/{1}/{0}.cshtml
	~/Areas/{2}/Views/Shared/{0}.cshtml
ViewLocationFormats	~/Views/{1}/{0}.cshtml
	~/Views/Shared/{0}.cshtml

As you can see, locations are not fully qualified paths but contain up to three placeholders.

- The placeholder {0} refers to the name of the view, as it is being invoked from the controller method.
- The placeholder {1} refers to the controller name as it is used in the URL.
- Finally, the controller {2}, if specified, refers to the area name.

Note If you're familiar with classic ASP.NET MVC development, you might be surprised to see that in ASP.NET Core, there's nothing like view location formats for partial views and layouts. In general, as we'll see in Chapter 6, views, partial views, and layouts are similar and are treated and discovered in the same way by the system. This is probably the rationale behind such a decision. Therefore, to add a custom view location for partial views or layout views, you simply add it to the *ViewLocationFormats* list.

Areas in ASP.NET MVC

Areas are a feature of the MVC application model used to group related functionalities within the context of a single application. Using areas is comparable to using multiple sub-applications, and it is a way to partition a large application into smaller segments.

The partition that areas offer is analogous to namespaces, and in an MVC project, adding an area (which you can do from the Visual Studio menu) results in adding a project folder where you have a distinct list of controllers, model types, and views. This allows you to have two or more HomeController classes for different areas of the application. Area partitioning is up to you and is not necessarily functional. You can also consider using areas one-to-one with roles.

In the end, areas are nothing technical or functional; instead, they're mostly related to the design and organization of the project and the code. When used, areas have an impact on routing. The name of the area is another parameter to be considered in the conventional routing. For more information refer to http://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/areas.

Customizing Location Formats

If I look back at almost a decade of ASP.NET MVC programming, I realize that in nearly any production application of medium complexity, I have ended up having a custom view engine or, more often, a customized version of the default Razor view engine.

The primary reason for using a configuration different from the default is always the need of organizing views and partial views in specific folders to make it simpler and faster to retrieve files when the number of views and partial views exceeds a couple of dozens. Razor views can be given any name following any sort of naming convention. Although neither a naming convention nor a custom organization of folders is strictly required, in the end, both are useful to manage and maintain your code.

My favorite naming convention is based on the use of a prefix in the name of views. For example, all my partial views begin with *pv_* whereas layout files begin with *layout_*. This guarantees that even when quite a few files are found in the same folder, they are grouped by name and can be spotted easily. Also, I still like to have a few additional subfolders at least for partial views and layouts. The code below shows how you can customize the view locations in ASP.NET Core.

```
public void ConfigureServices(IServiceCollection services)
{
    services
        .AddMvc()
        .AddRazorOptions(options =>
        {
            // Clear the current list of view location formats. At this time,
            // the list contains default view location formats.
            options.ViewLocationFormats.Clear();

            // {0} - Action Name
            // {1} - Controller Name
            // {2} - Area Name
            options.ViewLocationFormats.Add("/Views/{1}/{0}.cshtml");
            options.ViewLocationFormats.Add("/Views/Shared/{0}.cshtml");
            options.ViewLocationFormats.Add("/Views/Shared/Layouts/{0}.cshtml");
            options.ViewLocationFormats.Add("/Views/Shared/PartialViews/{0}.cshtml");
        });
}
```

The call to *Clear* empties the default list of view location strings so that the system will only work according to custom location rules. Figure 5-2 presents the resulting folder structure as it appears in a sample project. Note that now partial views will only be discovered if located under *Views/Shared* or *Views/Shared/PartialViews,* and layout files will only be discovered if located under *Views/Shared* or *Views/Shared/Layouts*.

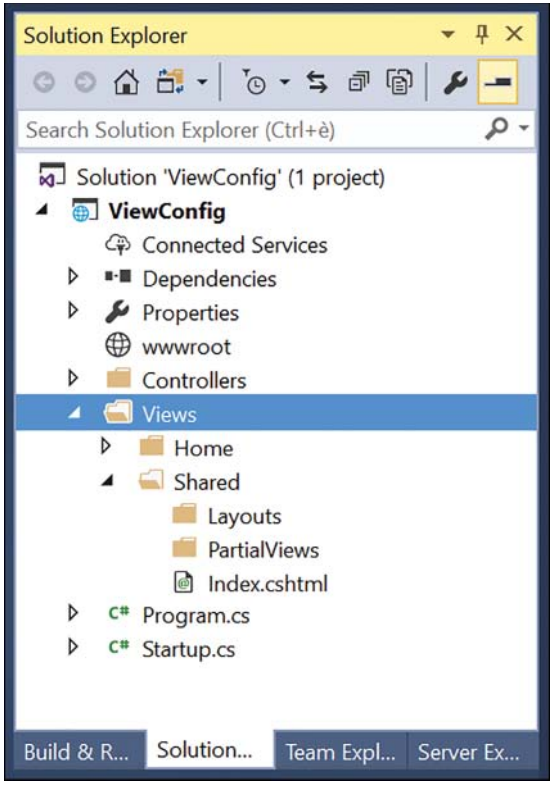

FIGURE 5-2 Customized view locations

Note If you are a bit unfamiliar with the concept of partial views and layout files, don't worry. In the next chapter, they will be fully explained with examples.

View Location Expanders

View location formats are a static setting for the view engine. You define view location formats at the application startup, and they remain active for the entire lifetime. Each time a view must be rendered, the view engine goes through the list of registered locations until it finds a location that contains the desired template. If no template is found, an exception is thrown. So far so good.

What if, instead, you need to determine the path to the view dynamically on a per-request basis? If it sounds like a weird use-case, think about multi-tenant applications. Imagine you have an application that is consumed as a service by multiple customers concurrently. It's always the same codebase, and it's always the same set of logical views, but each user can be served a specific version of the view, maybe styled differently or with a different layout.

A common approach for this type of application is defining the collection of default views and then allowing customers to add customized views. For example, let's say customer Contoso navigates to the view *index.cshtml* and expects to see *Views/Contoso/Home/index.cshtml* instead of the default view at *Views/Home/index.cshtml*. How would you code this?

In classic ASP.NET MVC, you had to create a custom view engine and override the logic to find views. It was not a huge amount of work—just a few lines of code—but yet you had to roll your own view engine and learn a lot about its internals. In ASP.NET Core, view location expanders are a new type of component made to resolve views dynamically. A view location expander is a class that implements the *IViewLocationExpander* interface.

```
public class MultiTenantViewLocationExpander : IViewLocationExpander
{
    public void PopulateValues(ViewLocationExpanderContext context)
    {
        var tenant = context.ActionContext.HttpContext.ExtractTenantCode();
        context.Values["tenant"] = tenant;
    }

    public IEnumerable<string> ExpandViewLocations(
                ViewLocationExpanderContext context,
                IEnumerable<string> viewLocations)
    {
        if (!context.Values.ContainsKey("tenant") ||
             string.IsNullOrWhiteSpace(context.Values["tenant"]))
                return viewLocations;

        var tenant = context.Values["tenant"];
        var views = viewLocations
                .Select(f => f.Replace("/Views/", "/Views/" + tenant + "/"))
                .Concat(viewLocations)
                .ToList();
        return views;
    }
}
```

In *PopulateValues,* you access the HTTP context and determine the key value that will determine the view path to use. This could easily be the code of the tenant you extract in some way from the requesting URL. The key value to be used to determine the path is stored in the view location expander context. In *ExpandViewLocations,* you receive the current list of view location formats, edit as appropriate based on the current context, and return it. Editing the list typically means inserting additional and context-specific view location formats.

According to the code above, if you get a request from *http://contoso.yourapp.com/home/index* and the tenant code is "contoso," then the returned list of view location formats can be as shown in Figure 5-3.

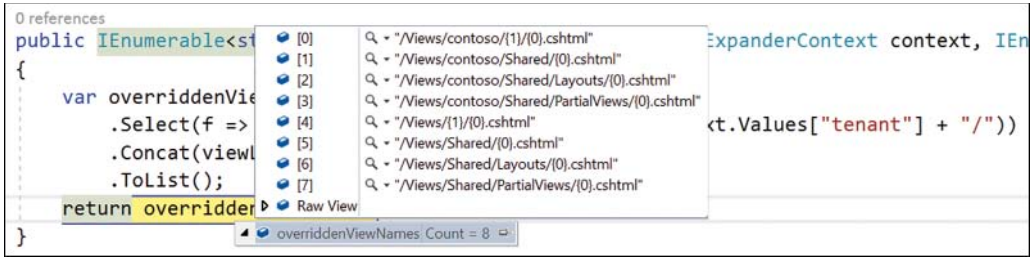

FIGURE 5-3 Using a custom location expander for a multi-tenant application

Tenant-specific location formats have been added at the top of the list, meaning that any overridden view will take precedence over any default view.

Your custom expander must be registered in the startup phase. Here's how to do it.

```
public void ConfigureServices(IServiceCollection services)
{
    services
        .AddMvc()
        .AddRazorOptions(options =>
        {
            options.ViewLocationExpanders.Add(new MultiTenantViewLocationExpander());
        });
}
```

Note that by default the no view location expander is registered in the system.

Adding a Custom View Engine

In ASP.NET Core the availability of view location expander components drastically reduces the need of having a custom view engine, at least for the need of customizing the way that views are retrieved and processed. A custom view engine is based on the *IViewEngine* interface, as shown below.

```
public interface IViewEngine
{
    ViewEngineResult FindView(ActionContext context, string viewName, bool isMainPage);
    ViewEngineResult GetView(string executingFilePath, string viewPath, bool isMainPage);
}
```

The method *FindView* is responsible for locating the specified view, and in ASP.NET Core, its behavior is largely customizable through location expanders. Instead, the method *GetView* is responsible for creating the view object, namely the component that will then be rendered to the output stream to capture the final markup. Typically, there's no need to override the behavior of *GetView* unless you need to something unusual, such as changing the template language.

These days, the Razor language and the Razor view are largely sufficient for most needs, and examples of alternate view engines are rare. However, some developers started projects to create and evolve alternate view engines that use the Markdown (MD) language to express HTML content. In my opinion, that is one of the few cases for really having (or using) a custom view engine.