CHAPTER 20

Views

In Chapter 17, you saw how action methods can return `ActionResult` objects. As you learned, the most commonly used action result is `ViewResult`, which causes a view to be rendered and returned to the client. You have seen views being used in many examples already, so you know roughly what they do. In this chapter, I focus and clarify that knowledge. I begin by showing you how the MVC Framework handles `ViewResults` using *view engines*, including demonstrating how to create a custom view engine. Next, I will describe techniques for working effectively with the built-in Razor View Engine. Then I will cover how to create and use partial views, child actions, and Razor sections, which are all essential topics for effective MVC development. Table 20-1 provides the summary for this chapter.

Table 20-1. *Chapter Summary*

Problem	Solution	Listing
Create a custom view engine	Implement the `IViewEngine` and `IView` interfaces	1–8
Customize the Razor view engine	Derive from the `RazorViewEngine` class	9–15
Define regions of content for use in a layout	Use Razor sections	16
Apply sections in a layout	Use the `RenderSection` and `RenderBody` helpers	17–22
Define reusable fragments of markup	Use partial views	23–26
Define reusable business logic	Use child actions	27–29

Creating a Custom View Engine

I am going to dive in at the deep end and create a custom view engine. You do not need to do this for most projects because the MVC Framework includes the Razor view engine, whose syntax I described in Chapter 5 and which I have been using for all of the examples so far in this book.

Tip Older versions of the MVC Framework supported views created using the same markup and view engine as ASP. NET Web Forms, which is why you will sometimes see references to `.aspx` files in debugging and error messages.

The value in creating a custom view engine is to demonstrate how the request processing pipeline works and complete your knowledge of how the MVC Framework operates. This includes understanding just how much freedom view engines have in translating a `ViewResult` into a response to the client. View engines implement the `IViewEngine` interface, which is shown in Listing 20-1.

Listing 20-1. The IViewEngine Interface from the MVC Framework

```
namespace System.Web.Mvc {
    public interface IViewEngine {

        ViewEngineResult FindPartialView(ControllerContext controllerContext,
            string partialViewName, bool useCache);

        ViewEngineResult FindView(ControllerContext controllerContext,
            string viewName, string masterName, bool useCache);

        void ReleaseView(ControllerContext controllerContext, IView view);
    }
}
```

The role of a view engine is to translate requests for views into `ViewEngineResult` objects. The first two methods in the interface, `FindView` and `FindPartialView`, are passed parameters that describe the request and the controller that processed it (a `ControllerContext` object), the name of the view and its layout, and whether the view engine is allowed to reuse a previous result from its cache. These methods are called when a `ViewResult` is being processed. The final method, `ReleaseView`, is called when a view is no longer needed.

■ **Note** The MVC Framework support for view engines is implemented by the `ControllerActionInvoker` class, which is the built-in implementation of the `IActionInvoker` interface, as described in Chapter 17. You will not have automatic access to the view engines feature if you have implemented your own action invoker or controller factory directly from the `IActionInvoker` or `IControllerFactory` interfaces.

The `ViewEngineResult` class allows a view engine to respond to the MVC Framework when a view is requested. Listing 20-2 shows the `ViewEngineResult` class.

Listing 20-2. The ViewEngineResult Class from the MVC Framework

```
using System.Collections.Generic;

namespace System.Web.Mvc {
    public class ViewEngineResult {

        public ViewEngineResult(IEnumerable<string> searchedLocations) {
            if (searchedLocations == null) {
                throw new ArgumentNullException("searchedLocations");
            }
            SearchedLocations = searchedLocations;
        }

        public ViewEngineResult(IView view, IViewEngine viewEngine) {
            if (view == null) { throw new ArgumentNullException("view");}
            if (viewEngine == null) { throw new ArgumentNullException("viewEngine");}
            View = view;
            ViewEngine = viewEngine;
        }
        public IEnumerable<string> SearchedLocations { get; private set; }
        public IView View { get; private set; }
        public IViewEngine ViewEngine { get; private set; }
    }
}
```

You express a result by choosing one of the two constructors. If your view engine is able to provide a view for a request, then you create a ViewEngineResult using this constructor:

```
...
public ViewEngineResult(IView view, IViewEngine viewEngine)
...
```

The parameters to this constructor are an implementation of the IView interface and a view engine (so that the ReleaseView method can be called later). If your view engine cannot provide a view for a request, then you use this constructor:

```
...
public ViewEngineResult(IEnumerable<string> searchedLocations)
...
```

The parameter for this version is an enumeration of the places you searched to find a view. This information is displayed to the user if no view can be found, as I will demonstrate later.

■ **Note** You are not alone if you think that the ViewEngineResult class is a little awkward. Expressing outcomes using different versions of a class constructor is an odd approach and does not really fit with the rest of the MVC Framework design.

The last building block of the view engine system is the IView interface, which is shown in Listing 20-3.

Listing 20-3. The IView Interface from the MVC Framework

```
using System.IO;

namespace System.Web.Mvc {
    public interface IView {
        void Render(ViewContext viewContext, TextWriter writer);
    }
}
```

An IView implementation is passed to the constructor of a ViewEngineResult object, which is then returned from the view engine methods. The MVC Framework then calls the Render method. The ViewContext parameter provides information about the request from the client and the output from the action method. The TextWriter parameter is for writing output to the client.

The ViewContext object defines properties that give you access to information about the request and details of how the MVC Framework has processed it so far. I have described the most useful of these properties in Table 20-2.

Table 20-2. *Useful ViewContext Properties*

Name	Description
`Controller`	Returns the `IController` implementation that processed the current request
`RequestContext`	Returns details of the current request
`RouteData`	Returns the routing data for the current request
`TempData`	Returns the temp data associated with the request
`View`	Returns the implementation of the `IView` interface that will process the request. Obviously, this will be the current class if you are creating a custom view implementation.
`ViewBag`	Returns an `object` that represents the view bag
`ViewData`	Returns a dictionary of the view model data, which also contains the view bag and meta data for the model. See Table 20-3 for details.

The most interesting of these properties is `ViewData`, which returns a `ViewDataDictionary` object. The `ViewDataDictionary` class defines a number of useful properties that give access to the view model, the view bag and the view model metadata. I have described the most useful of these properties in Table 20-3.

Table 20-3. *Useful ViewDataDictionary Properties*

Name	Description
`Keys`	Returns a collection of key values for the data in the dictionary, which can be used to access view bag properties
`Model`	Returns the view model object for the request
`ModelMetadata`	Returns a `ModelMetadata` object that can be used to reflect on the model type
`ModelState`	Returns information about the state of the model, which I describe in detail in Chapter 25

As I said earlier, the simplest way to see how this works—how `IViewEngine`, `IView`, and `ViewEngineResult` fit together—is to create a view engine. I am going to create a simple view engine that returns one kind of view. This view will render a result that contains information about the request and the view data produced by the action method. This approach lets me demonstrate the way that view engines operate without getting bogged down in parsing view templates.

Preparing the Example Project

The example project for this part of the chapter is called `Views` and I created it using the `Empty` template, checking the option to add the core MVC folders and references. I created a `Home` controller, which you can see in Listing 20-4.

Listing 20-4. The Contents of the HomeController.cs File

```
using System;
using System.Web.Mvc;

namespace Views.Controllers {
    public class HomeController : Controller {
        public ActionResult Index() {
            ViewBag.Message = "Hello, World";
            ViewBag.Time = DateTime.Now.ToShortTimeString();
            return View("DebugData");
        }

        public ActionResult List() {
            return View();
        }
    }
}
```

I have not created any views for this project because I am going to implement a custom view engine rather than relying on Razor.

Creating a Custom IView

I am going to start by creating an implementation of the `IView` interface. I added an `Infrastructure` folder to the example project and created a new class file within it called `DebugDataView.cs`, which is shown in Listing 20-5.

Listing 20-5. The Contents of the DebugDataView.cs

```
using System.IO;
using System.Web.Mvc;

namespace Views.Infrastructure {
    public class DebugDataView : IView {

        public void Render(ViewContext viewContext, TextWriter writer) {

            Write(writer, "---Routing Data---");
            foreach (string key in viewContext.RouteData.Values.Keys) {
                Write(writer, "Key: {0}, Value: {1}",
                    key, viewContext.RouteData.Values[key]);
            }

            Write(writer, "---View Data---");
            foreach (string key in viewContext.ViewData.Keys) {
                Write(writer, "Key: {0}, Value: {1}", key,
                    viewContext.ViewData[key]);
            }
        }

        private void Write(TextWriter writer, string template, params object[] values) {
            writer.Write(string.Format(template, values) + "<p/>");
        }
    }
}
```

This view demonstrates the use of the two parameters to the `Render` method: I take values from the `ViewContext` and write a response to the client using the `TextWriter`. First I write out the routing data information and then the view bag data.

■ **Tip** The view data feature is a holdover from earlier versions of the MVC Framework that were released before C# had support for dynamic objects (which I described in Chapter 4). View data was a less flexible precursor to the view bag and isn't used directly any more, except when writing custom `IView` implementations when it provides easy access to the properties defined on the view bag object.

Creating an IViewEngine Implementation

Remember that the purpose of the view engine is to produce a `ViewEngineResult` object that contains either an `IView` or a list of the places that searched for a suitable view. Now that I have an `IView` implementation to work with, I can create the view engine. I added a class file called `DebugDataViewEngine.cs` in the `Infrastructure` folder, the contents of which are shown in Listing 20-6.

Listing 20-6. The Contents of the DebugDataViewEngine.cs File

```
using System.Web.Mvc;

namespace Views.Infrastructure {
    public class DebugDataViewEngine : IViewEngine {

        public ViewEngineResult FindView(ControllerContext controllerContext,
                string viewName, string masterName, bool useCache) {

            if (viewName == "DebugData") {
                return new ViewEngineResult(new DebugDataView(), this);
            } else {
                return new ViewEngineResult(new string[]
                    { "No view (Debug Data View Engine)" });
            }
        }

        public ViewEngineResult FindPartialView(ControllerContext controllerContext,
                string partialViewName, bool useCache) {

            return new ViewEngineResult(new string[]
                { "No view (Debug Data View Engine)" });
        }

        public void ReleaseView(ControllerContext controllerContext, IView view) {
            // do nothing
        }
    }
}
```

I am going to support only a single view, which is called DebugData. When I see a request for that view, I will return a new instance of the custom IView implementation, like this:

```
...
return new ViewEngineResult(new DebugDataView(), this);
...
```

If I were implementing a more serious view engine, I would use this opportunity to search for templates, taking into account the layout and provided caching settings. As it is, this simple example only requires a new instance of the DebugDataView class. If I receive a request for a view other than DebugData, I return a ViewEngineResult, like this:

```
...
return new ViewEngineResult(new string[] { "No view (Debug Data View Engine)" });
...
```

The IViewEngine interface presumes that the view engine has places it needs to look to find views. This is a reasonable assumption, because views are typically template files that are stored as files in the project. In this case, I do not have anywhere to look, so I just return a dummy location which will indicate that I was asked for a view that cannot be delivered.

The custom view engine doesn't support partial views, so I return a result from the FindPartialView method that indicates I do not have a view to offer. I return to the topic of partial views and how they are handled in the Razor engine later in the chapter. I have not implemented the ReleaseView method, because there are no resources that I need to release in the custom IView implementation, which is the usual purpose of this method.

Registering a Custom View Engine

I register view engines in the Application_Start method of Global.asax, as shown in Listing 20-7.

Listing 20-7. Registering a Custom View Engine Using Global.asax

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using Views.Infrastructure;

namespace Views {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ViewEngines.Engines.Add(new DebugDataViewEngine());
        }
    }
}
```

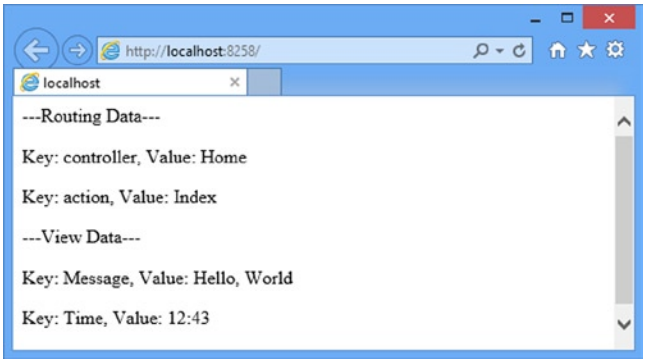
The static ViewEngine.Engines collection contains the set of view engines that are installed in the application. The MVC Framework supports the idea of there being several engines installed in a single application. When a ViewResult is being processed, the action invoker obtains the set of installed view engines and calls their FindView methods in turn.

The action invoker stops calling FindView methods as soon as it receives a ViewEngineResult object that contains an IView. This means that the order in which engines are added to the ViewEngines.Engines collection is significant if two or more engines are able to service a request for the same view name. If you want your view to take precedence, then you can insert it at the start of the collection, like this:

```
...
ViewEngines.Engines.Insert(0, new DebugDataViewEngine());
...
```

Testing the View Engine

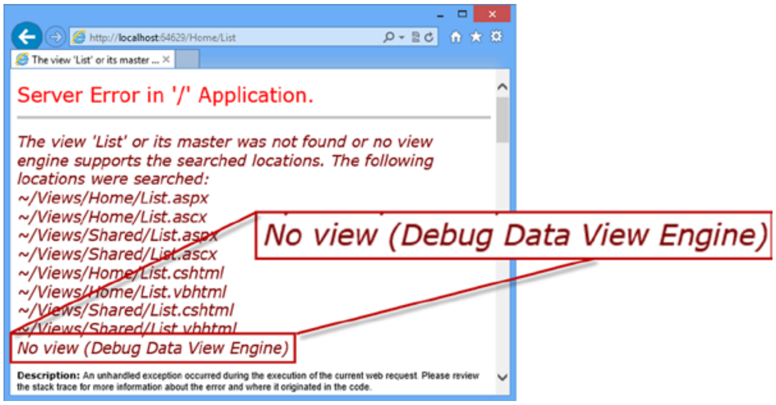
I am now in a position to test the custom view engine. When the application is started, the browser will automatically navigate to the root URL for the project, which will be mapped to the Index action in the Home controller. The action method uses the View method to return a ViewResult that specifies the DebugData view. You can see the result of this in Figure 20-1.



```
---Routing Data---

Key: controller, Value: Home

Key: action, Value: Index

---View Data---

Key: Message, Value: Hello, World

Key: Time, Value: 12:43
```

Figure 20-1. *Using a custom view engine*

This is the result of the FindView method being called for a view that I am able to process. If you navigate to the /Home/List URL, the MVC Framework will invoke the List action method, which calls the View method to request its default view, which is not one that is supported. You can see the result in Figure 20-2.


http://localhost:64629/Home/List

The view 'List' or its master ... ×

Server Error in '/' Application.

The view 'List' or its master was not found or no view engine supports the searched locations. The following locations were searched:
~/Views/Home/List.aspx
~/Views/Home/List.ascx
~/Views/Shared/List.aspx
~/Views/Shared/List.ascx
~/Views/Home/List.cshtml
~/Views/Home/List.vbhtml
~/Views/Shared/List.cshtml
~/Views/Shared/List.vbhtml
No view (Debug Data View Engine)

No view (Debug Data View Engine)

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

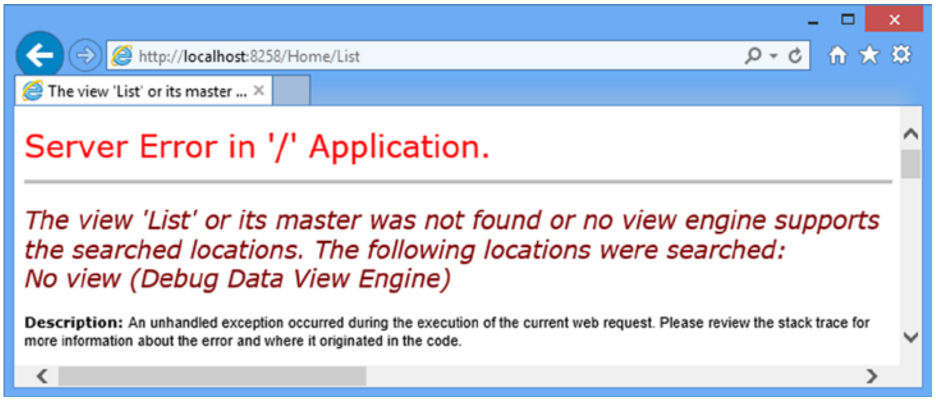
Figure 20-2. *Requesting an unsupported view*

You can see that my message is reported as one of the locations that have been searched for a view. Notice that Razor and ASPX views appear on the list as well. This is because those view engines are still being used. If I want to ensure that only my custom view engine is used, then I have to call the `Clear` method before I register my engine in the `Global.asax` file, as shown in Listing 20-8.

Listing 20-8. Removing the Other View Engines in the Global.asax File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using Views.Infrastructure;

namespace Views {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ViewEngines.Engines.Clear();
            ViewEngines.Engines.Add(new DebugDataViewEngine());
        }
    }
}
```

If you restart the application and navigate to `/Home/List` again, only the custom view engine will be used, as shown in Figure 20-3.



http://localhost:8258/Home/List

Server Error in '/' Application.

The view 'List' or its master was not found or no view engine supports the searched locations. The following locations were searched:
No view (Debug Data View Engine)

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Figure 20-3. *Using only the custom view engine in the example application*

Working with the Razor Engine

In the previous section, I was able to create a custom view engine by implementing just two interfaces. Admittedly, I ended up with something simple that generated ugly views, but you saw how the concept of MVC extensibility continues throughout the request processing pipeline.

The complexity in a view engine comes from the system of view templates that includes code fragments and support layouts, and is compiled to optimize performance. I did not do any of these things in the simple custom view engine—and there isn't much need to—because the built-in Razor engine takes care of all of that for me. The functionality that almost all MVC applications require is available in Razor. Only a vanishingly small number of projects need to go to the trouble of creating a custom view engine. I gave you a primer on the Razor syntax in Chapter 5. In this chapter, I am going to show you how to use other features to create and render Razor views. You will also learn how to customize the Razor engine.

Preparing the Example Project

For this part of the chapter, I have created a new MVC project using the `Empty` template option, checking the option to add the core MVC folders and references. I called the project `WorkingWithRazor` and I added a `Home` controller, which is shown in Listing 20-9.

Listing 20-9. The Contents of the HomeController.cs File

```
using System.Web.Mvc;

namespace WorkingWithRazor.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            string[] names = { "Apple", "Orange", "Pear" };
            return View(names);
        }
    }
}
```

I also created a view called `Index.cshtml` in the `Views/Home` folder. You can see the contents of the view file in Listing 20-10.

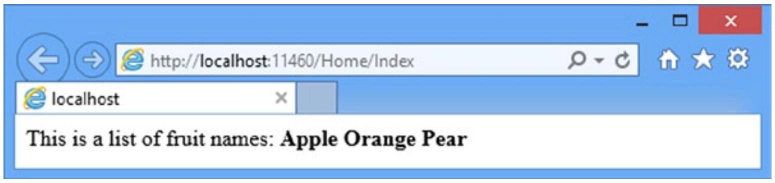
Listing 20-10. The Contents of the Index.cshtml File

```
@model string[]

@{
    ViewBag.Title = "Index";
}

This is a list of fruit names:

@foreach (string name in Model) {
    <span><b>@name</b></span>
}
```

Understanding Razor View Rendering

The Razor View Engine compiles the views in your applications to improve performance. The views are translated into C# classes, and then compiled, which is why you are able to include C# code fragments so easily. It is instructive to look at the source code that Razor views generate, because it helps to put many of the Razor features in context.

The views in an MVC application are not compiled until the application is started, so to see the classes that are created by Razor, you need to start the application and navigate to the `/Home/Index` action. The initial request to MVC application triggers the compilation process for all views. You can see the output from the request in Figure 20-4.



This is a list of fruit names: **Apple Orange Pear**

Figure 20-4. *The output from the Index action method on the Home controller*

Conveniently, classes generated from the view files are written to the disk as C# code files and then compiled, which means that you can see the C# statements that represent a view. You can find the generated files in `c:\Users\<`*`yourLoginName`*`>\AppData\Local\Temp\Temporary ASP.NET Files` on Windows 7 and Windows 8.

Finding the code file generated for a particular view requires a bit of poking around. There are usually a number of folders with cryptic names, and the names of the `.cs` files do not correspond to the names of the classes they contain. As an example, I found the generated class for the view in Listing 20-10 in a file called `App_Web_ihpp0d0l.0.cs` in the `root\7bbfc2bc\bd7485cd` folder. I have tidied up the class from my system to make it easier to read, as shown in Listing 20-11.

Listing 20-11. The Generated C# Class for a Razor View

```
namespace ASP {
    using System;
    using System.Collections.Generic;
    using System.IO;
    using System.Linq;
    using System.Net;
    using System.Web;
    using System.Web.Helpers;
    using System.Web.Security;
    using System.Web.UI;
    using System.Web.WebPages;
    using System.Web.Mvc;
    using System.Web.Mvc.Ajax;
    using System.Web.Mvc.Html;
    using System.Web.Optimization;
    using System.Web.Routing;

    public class _Page_Views_Home_Index_cshtml : System.Web.Mvc.WebViewPage<string[]> {

        public _Page_Views_Home_Index_cshtml() {
        }

        public override void Execute() {
            ViewBag.Title = "Index";
            WriteLiteral("\r\n\r\nThis is a list of fruit names:\r\n\r\n");

            foreach (string name in Model) {
                WriteLiteral("    <span><b>");
                Write(name);
                WriteLiteral("</b></span>\r\n");
            }
        }
    }
}
```

First, note that the class is derived from `WebViewPage<T>`, where `T` is the model type: `WebViewPage<string[]>` for this example. This is how strongly typed views are handled. Also notice the name of the class that has been generated: `_Page_Views_Home_Index_cshtml`. You can see how the path of the view file has been encoded in the class name. This is how Razor maps requests for views into instances of compiled classes.

In the `Execute` method, you can see how the statements and elements in the view have been handled. The code fragments that I prefixed with the `@` symbol are expressed directly as C# statements. The HTML elements are handled with the `WriteLiteral` method, which writes the contents of the parameter to the result as they are given. This is opposed to the `Write` method, which is used for C# variables and encodes the string values to make them safe for use in an HTML page.

Both the `Write` and `WriteLiteral` methods write content to a `TextWriter` object. This is the same object that is passed to the `IView.Render` method, which you saw at the start of the chapter. The goal of a compiled Razor view is to generate the static and dynamic content and send it to the client via the `TextWriter.` This is useful to keep in mind when I turn to HTML helper methods later in later chapters.

Configuring the View Search Locations

The Razor View Engine follows a standard convention when looking for a view. For example, if you request the `Index` view associated with the `Home` controller, Razor looks through this list of views:

- `~/Views/Home/Index.cshtml`
- `~/Views/Home/Index.vbhtml`

- `~/Views/Shared/Index.cshtml`
- `~/Views/Shared/Index.vbhtml`

As you now know, Razor is not really looking for the view files on disk, because they have already been compiled into C# classes. Razor looks for the compiled class that represents these views. The `.cshtml` files are templates containing C# statements (the kind I am using), and the `.vbhtml` files contain Visual Basic statements.

You can change the view files that Razor searches for by creating a subclass of `RazorViewEngine`. This class is the Razor `IViewEngine` implementation. It builds on a series of base classes that define a set of properties that determine which view files are searched for. These properties are described in Table 20-4.

Table 20-4. *Razor View Engine Search Properties*

Property	Description	Default Value

`ViewLocationFormats`
`MasterLocationFormats`
`PartialViewLocationFormats`
	The locations to look for views, partial views, and layouts	`~/Views/{1}/{0}.cshtml,`
		`~/Views/{1}/{0}.vbhtml,`
		`~/Views/Shared/{0}.cshtml,`
		`~/Views/Shared/{0}.vbhtml`

`AreaViewLocationFormats`
`AreaMasterLocationFormats`
`AreaPartialViewLocationFormats`
	The locations to look for views, partial views, and layouts for an area	`~/Areas/{2}/Views/{1}/{0}.cshtml,`
		`~/Areas/{2}/Views/{1}/{0}.vbhtml,`
		`~/Areas/{2}/Views/Shared/{0}.cshtml,`
		`~/Areas/{2}/Views/Shared/{0}.vbhtml`

These properties predate the introduction of Razor, which why each set of three properties has the same values. Each property is an array of strings, which are expressed using the composite string formatting notation. The following are the parameter values that correspond to the placeholders:

- `{0}` represents the name of the view.
- `{1}` represents the name of the controller.
- `{2}` represents the name of the area.

To change the search locations, you create a new class that is derived from `RazorViewEngine` and change the values for one or more of the properties described in Table 20-4.

To demonstrate how to change the locations that are searched, I added an `Infrastructure` folder to the project and created a class file called `CustomLocationViewEngine.cs`, which is shown in Listing 20-12.

Listing 20-12. The Contents of the CustomLocationViewEngine.cs File

```
using System.Web.Mvc;

namespace WorkingWithRazor.Infrastructure {
    public class CustomLocationViewEngine : RazorViewEngine {

        public CustomLocationViewEngine() {
            ViewLocationFormats = new string[]
                {"~/Views/{1}/{0}.cshtml", "~/Views/Common/{0}.cshtml"};
        }
    }
}
```

I have set a new value for the ViewLocationFormats. The new array contains entries only for .cshtml files. In addition, I have changed the location I look for shared views to be Views/Common, rather than Views/Shared. I register the derived view engine using the ViewEngines.Engines collection in the Application_Start method of Global.asax, as shown in Listing 20-13.

Listing 20-13. Registering the Custom View Engine in the Global.asax File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using WorkingWithRazor.Infrastructure;

namespace WorkingWithRazor {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ViewEngines.Engines.Clear();
            ViewEngines.Engines.Add(new CustomLocationViewEngine());
        }
    }
}
```

Remember that the action invoker goes to each view engine in turn to see if a view can be found. By the time that I am able to add the view to the collection, it will already contain the standard Razor View Engine. To avoid competing with that implementation, I call the Clear method to remove any other view engines that may have been registered, and then call the Add method to register the custom implementation.

To demonstrate the changed locations, I created the /Views/Common folder and added a view file called List.cshtml. You can see the contents of this file in Listing 20-14.

Listing 20-14. The Contents of the /Views/Common/List.cshtml File

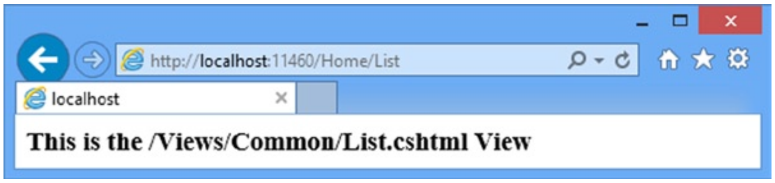
```
@{
    ViewBag.Title = "List";
}

<h3>This is the /Views/Common/List.cshtml View</h3>
```

To display this view, I added a new action method to the Home controller, as shown in Listing 20-15.

Listing 20-15. Adding a New Action Method in the HomeController.cs File

```
using System.Web.Mvc;

namespace WorkingWithRazor.Controllers {
    public class HomeController : Controller {
        public ActionResult Index() {
            string[] names = { "Apple", "Orange", "Pear" };
            return View(names);
        }

        public ActionResult List() {
            return View();
        }
    }
}
```

When I start the application and navigate to the /Home/List URL, the custom locations will be used to locate the List.cshtml view file in the /Views/Common folder, as shown in Figure 20-5.



This is the /Views/Common/List.cshtml View

Figure 20-5. *The effect of custom locations in the view engine*

Adding Dynamic Content to a Razor View

The whole purpose of views is to allow you to render parts of your domain model as a user interface. To do that, you need to be able to add *dynamic content* to views. Dynamic content is generated at runtime, and can be different for each and every request. This is opposed to *static content*, such as HTML, which you create when you are writing the application and is the same for each and every request. You can add dynamic content to views in the different ways described in Table 20-5.

Table 20-5. *Adding Dynamic Content to a View*

Technique	When to Use
Inline code	Use for small, self-contained pieces of view logic, such as if and foreach statements. This is the fundamental tool for creating dynamic content in views, and some of the other approaches are built on it. I introduced this technique in Chapter 5 and you have seen countless examples in the chapters since.
HTML helper methods	Use to generate single HTML elements or small collections of them, typically based on view model or view data values. The MVC Framework includes a number of useful HTML helper methods, and it is easy to create your own. HTML helper methods are the topic of Chapter 21.
Sections	Use for creating sections of content that will be inserted into layout at specific locations.
Partial views	Use for sharing subsections of view markup between views. Partial views can contain inline code, HTML helper methods, and references to other partial views. Partial views do not invoke an action method, so they cannot be used to perform business logic.
Child actions	Use for creating reusable UI controls or widgets that need to contain business logic. When you use a child action, it invokes an action method, renders a view, and injects the result into the response stream.

Two of these options, inline code and HTML helper methods, are covered elsewhere in this book and I describe the others in the sections that follow.

Using Layout Sections

The Razor engine supports the concept of *sections*, which allow you to provide regions of content within a layout. Razor sections give greater control over which parts of the view are inserted into the layout and where they are placed. To demonstrate the sections feature, I have edited the `/Views/Home/Index.cshtml` file, as shown in Listing 20-16.

Listing 20-16. Defining a Section in the Index.cshtml File

```
@model string[]

@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Header {
    <div class="view">
        @foreach (string str in new [] {"Home", "List", "Edit"}) {
            @Html.ActionLink(str, str, null, new { style = "margin: 5px" })
        }
    </div>
}

<div class="view">
    This is a list of fruit names:

    @foreach (string name in Model) {
        <span><b>@name</b></span>
    }
</div>

@section Footer {
    <div class="view">
        This is the footer
    </div>
}
```

Sections are defined using the Razor `@section` tag followed by a name for the section. In this listing, I created sections called `Header` and `Footer`. The content of a section contains the usual mix of HTML markup and Razor tags. Sections are defined in the view, but applied in a layout with the `@RenderSection` helper method. To demonstrate how this works, I created the `/Views/Shared/_Layout.cshtml` file, the contents of which you can see in Listing 20-17.

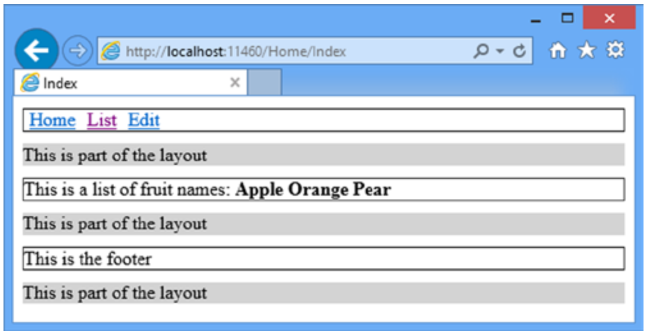
Listing 20-17. Using Sections in the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout { background-color: lightgray;}
        div.view { border: thin solid black; margin: 10px 0;}
    </style>
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")

    <div class="layout">
        This is part of the layout
    </div>

    @RenderBody()

    <div class="layout">
        This is part of the layout
    </div>

    @RenderSection("Footer")

    <div class="layout">
        This is part of the layout
    </div>
</body>
</html>
```

■ **Tip** My custom view engine locations are still in use, but I have specified the view explicitly in the `Index.cshtml` file, which means that my layout will be obtained from the `/Views/Shared` folder, even though shared views are located in the `/Views/Common` folder.

When Razor parses the layout, the `RenderSection` helper method is replaced with the contents of the section in the view with the specified name. The parts of the view that are not contained with a section are inserted into the layout using the `RenderBody` helper.

You can see the effect of the sections by starting the application, as shown in Figure 20-6. I added some basic CSS styles to help make it clear which sections of the output are from the view and which are from the layout. This result is not pretty, but it neatly demonstrates how you can put regions of content from the view into specific locations in the layout.



Home List Edit

This is part of the layout

This is a list of fruit names: **Apple Orange Pear**

This is part of the layout

This is the footer

This is part of the layout

Figure 20-6. *Using sections in a view to locate content in a layout*

■ **Note** A view can define only the sections that are referred to in the layout. The MVC Framework will throw an exception if you attempt to define sections in the view for which there is no corresponding `@RenderSection` helper call in the layout.

Mixing the sections in with the rest of the view is unusual. The convention is to define the sections at either the start or the end of the view, to make it easier to see which regions of content will be treated as sections and which will be captured by the `RenderBody` helper. Another approach, which I tend to use, is to define the view solely in terms of sections, including one for the body, as shown in Listing 20-18.

Listing 20-18. Defining a View in Terms of Razor Sections in the Index.cshtml File

```
@model string[]

@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@section Header {
    <div class="view">
        @foreach (string str in new [] {"Home", "List", "Edit"}) {
            @Html.ActionLink(str, str, null, new { style = "margin: 5px" })
        }
    </div>
}

@section Body {
    <div class="view">
        This is a list of fruit names:

        @foreach (string name in Model) {
            <span><b>@name</b></span>
        }
    </div>
}
```

```
@section Footer {
    <div class="view">
        This is the footer
    </div>
}
```

I find this makes for clearer views and reduces the chances of extraneous content being captured by RenderBody. To use this approach, I have to replace the call to the RenderBody helper with RenderSection("Body"), as shown in Listing 20-19.

Listing 20-19. Using RenderSection("Body") in the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout { background-color: lightgray;}
        div.view { border: thin solid black; margin: 10px 0;}
    </style>
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")

    <div class="layout">
        This is part of the layout
    </div>

    @RenderSection("Body")

    <div class="layout">
        This is part of the layout
    </div>

    @RenderSection("Footer")

    <div class="layout">
        This is part of the layout
    </div>
</body>
</html>
```

Testing For Sections

You can check to see if a view has defined a specific section from the layout. This is a useful way to provide default content for a section if a view does not need or want to provide specific content. I have modified the _Layout.cshtml file to check to see if a Footer section is defined, as shown in Listing 20-20.

Listing 20-20. Checking Whether a Section Is Defined in the _Layout.cshtml File

```
...
@if (IsSectionDefined("Footer")) {
    @RenderSection("Footer")
} else {
    <h4>This is the default footer</h4>
}
...
```

The `IsSectionDefined` helper takes the name of the section you want to check and returns `true` if the view you are rendering defines that section. In the example, I used this helper to determine if I should render some default content when the view does not define the `Footer` section.

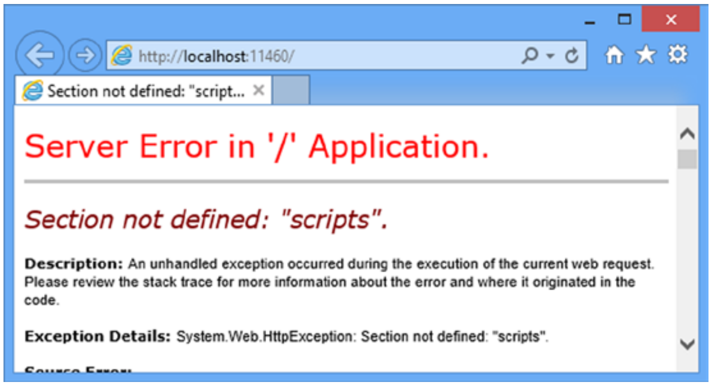
Rendering Optional Sections

By default a view has to contain all of the sections for which there are `RenderSection` calls in the layout. If sections are missing, then the MVC Framework will report an exception to the user. To demonstrate this, I have added a new `RenderSection` call to the `_Layout.cshtml` file for a section called `scripts`, as shown in Listing 20-21. This is a section that Visual Studio adds to the layout by default when you create an MVC project using the `MVC` template.

Listing 20-21. A RenderSection Call for Which There Is No Corresponding Section in the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout { background-color: lightgray;}
        div.view { border: thin solid black; margin: 10px 0;}
    </style>
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")

    <div class="layout">
        This is part of the layout
    </div>

    @RenderSection("Body")

    <div class="layout">
        This is part of the layout
    </div>

    @if (IsSectionDefined("Footer")) {
        @RenderSection("Footer")
    } else {
        <h4>This is the default footer</h4>
    }
    @RenderSection("scripts")

    <div class="layout">
        This is part of the layout
    </div>
</body>
</html>
```

When you start the application and the Razor engine attempts to render the layout and the view, you will see the error shown in Figure 20-7.


http://localhost:11460/

Section not defined: "script...

Server Error in '/' Application.

Section not defined: "scripts".

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Web.HttpException: Section not defined: "scripts".

Figure 20-7. *The error shown when there is a missing section*

You can use the `IsSectionDefined` method to avoid making `RenderSection` calls for sections that the view does not define, but a more elegant approach is to use optional sections, which you do by passing an additional `false` value to the `RenderSection` method, as shown in Listing 20-22.

Listing 20-22. Making a Section Optional

```
...
@RenderSection("scripts", false)
...
```

This creates an optional section, the contents of which will be inserted into the result if the view defines it and which will not throw an exception otherwise.

Using Partial Views

You will often need to use the same fragments of Razor tags and HTML markup in several different places in the application. Rather than duplicate the content, you can use *partial views*, which are separate view files that contain fragments of tags and markup that can be included in other views. In this section, I show you how to create and use partial views, explain how they work, and demonstrate the techniques available for passing view data to a partial view.

Creating a Partial View

I am going to start by creating a partial view called `MyPartial`. Right-click on the `Views/Shared` folder, select `Add` ➤ `View` from the popup menu. Visual Studio will display the `Add View` dialog window, which you have seen in previous chapters. Set `View Name` to `MyPartial`, `Template` to `Empty (without model)` and check the `Create as partial view` option, as shown in Figure 20-8.

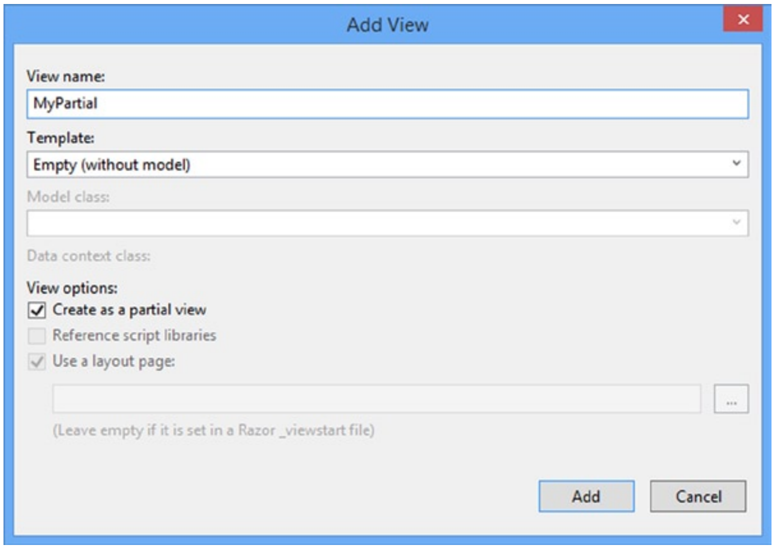


Figure 20-8. *Creating a partial view*

Click the `Add` button and Visual Studio will create the partial view, which is initially empty. I added the content shown in Listing 20-23.

Listing 20-23. The Content of the MyPartial.cshtml File

```
<div>
    This is the message from the partial view.
    @Html.ActionLink("This is a link to the Index action", "Index")
</div>
```

■ **Tip** The scaffolding feature only sets the initial content for a file. What makes a view a partial is its content (it only contains a fragment of HTML, rather than a complete HTML document, and doesn't reference layouts) and the way that it is used (which I describe shortly). Once you are familiar with the different kinds of view, you can just use `Add` ➤ `MVC 5 View Page (Razor)` and set the contents you require directly.

I want to demonstrate that you can mix HTML markup and Razor tags in a partial view, so I have defined a simple message and a call to the `ActionLink` helper method. A partial view is consumed by calling the `Html.Partial` helper method from within another view. To demonstrate this, I have made the changes to the `~/Views/Common/List.cshtml` view file shown in Listing 20-24.

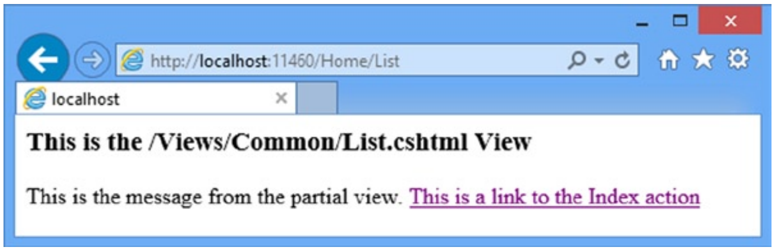
Listing 20-24. Consuming a Partial View in the List.cshtml File

```
@{
    ViewBag.Title = "List";
    Layout = null;
}

<h3>This is the /Views/Common/List.cshtml View</h3>

@Html.Partial("MyPartial")
```

I specify the name of the partial view file without the file extension. The view engine will look for the partial view that I have specified in the usual locations, which means the `/Views/Home` and `/Views/Shared` folders for this example, since I called the `Html.Partial` method in a view that is being rendered for the `Home` controller. (I set the `Layout` variable to `null` so that I do not have to specify the sections I defined in the `_Layout.cshtml` file used earlier in the chapter.)

■ **Tip** The Razor View Engine looks for partial views in the same way that it looks for regular views (in the `~/Views/<controller>` and `~/Views/Shared` folders). This means that you can create specialized versions of partial views that are controller-specific and override partial views of the same name in the `Shared` folder. This may seem like an odd thing to do, but one of the most common uses of partial views is to render content in layouts, and this feature can be handy.

You can see the effect of consuming the partial view by starting the application and navigating to the `/Home/List` URL, as shown in Figure 20-9.



http://localhost:11460/Home/List

This is the /Views/Common/List.cshtml View

This is the message from the partial view. This is a link to the Index action

Figure 20-9. *The effect of consuming a partial view*

■ **Tip** The call I made to the `ActionLink` helper method in the partial view takes its controller information from the request that is being processed. That means that when I specified the `Index` method, the `a` element will refer to the `Home` controller, since that was the controller that led to the partial view being rendered. If I consume the partial view in a view being rendered for another controller, then the `ActionLink` would generate a reference to that controller instead. I come back to the topic of HTML helper methods in Chapter 21.

Using Strongly Typed Partial Views

You can create strongly typed partial views, and then pass view model objects to be used when the partial view is rendered. To demonstrate this feature, I created a new strongly typed partial view called `MyStronglyTypedPartial.cshtml` in the `/Views/Shared` folder. This time, rather than use the scaffold option, I selected `Add` ➤ `MVC 5 View Page (Razor)`, set the name to `MyStronglyTypedPartial` and clicked the `OK` button to create the view. As I explained in the previous section, there is nothing about the file itself that denotes a partial view, just the content and the way it is used in the application. I removed the default content that Visual Studio adds to new view files and replaced it with the markup shown in Listing 20-25.

Listing 20-25. The Contents of the MyStronglyTypedPartial.cshtml File

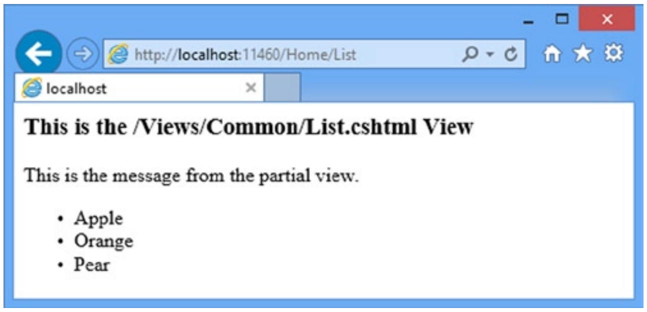
```
@model IEnumerable<string>

<div>
    This is the message from the partial view.
    <ul>
        @foreach (string str in Model) {
            <li>@str</li>
        }
    </ul>
</div>
```

I use a Razor `@foreach` loop to display the contents of the view model object as items in an HTML list. To demonstrate the use of this partial view, I updated the `/Views/Common/List.cshtml` file, as shown in Listing 20-26.

Listing 20-26. Consuming a Strongly Typed Partial View in the List.cshtml File

```
@{
    ViewBag.Title = "List";
    Layout = null;
}

<h3>This is the /Views/Common/List.cshtml View</h3>

@Html.Partial("MyStronglyTypedPartial", new [] {"Apple", "Orange", "Pear"})
```

The difference from the previous example is that I pass an additional argument to the `Partial` helper method which defines the view model object. You can see the strongly typed partial view in use by starting the application and navigating to the `/Home/List` URL, as shown in Figure 20-10.



This is the /Views/Common/List.cshtml View

This is the message from the partial view.

- Apple
- Orange
- Pear

Figure 20-10. *Consuming a strongly typed partial view*

Using Child Actions

Child actions are action methods invoked from within a view. This lets you avoid repeating controller logic that you want to use in several places in the application. Child actions are to actions as partial views are to views. You can use a child action whenever you want to display some data-driven widget that appears on multiple pages and contains data unrelated to the main action that is running. I used this technique in the SportsStore example to include a data-driven navigation menu on every page, without needing to supply the navigation data directly from every action method. The navigation data was supplied independently by the child action.

Creating a Child Action

Any action can be used as a child action. To demonstrate the child action feature, I have added a new action method to the Home controller, as shown in Listing 20-27.

Listing 20-27. Adding a Child Action in the HomeController.cs File

```
using System;
using System.Web.Mvc;

namespace WorkingWithRazor.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            string[] names = { "Apple", "Orange", "Pear" };
            return View(names);
        }

        public ActionResult List() {
            return View();
        }

        [ChildActionOnly]
        public ActionResult Time() {
            return PartialView(DateTime.Now);
        }
    }
}
```

The action method is called Time and it renders a partial view by calling the PartialView method (which I described in Chapter 17). The ChildActionOnly attribute ensures that an action method can be called *only* as a child method from within a view. An action method doesn't need to have this attribute to be used as a child action, but I tend to use it to prevent the action methods from being invoked as a result of a user request.

Having defined an action method, I need to create the partial view that will be rendered when the action is invoked. Child actions are typically associated with partial views, although this is not compulsory. Listing 20-28 shows the /Views/Home/Time.cshtml view that I created for this demonstration. This is a strongly typed partial view whose view model is a DateTime object.

Listing 20-28. The Contents of the Time.cshtml File

```
@model DateTime

<p>The time is: @Model.ToShortTimeString()</p>
```

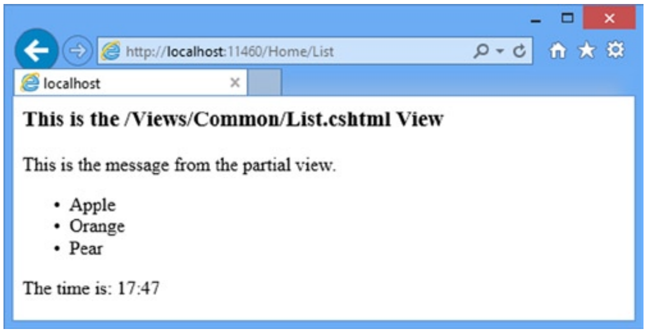
Rendering a Child Action

Child actions are invoked using the `Html.Action` helper. With this helper, the action method is executed, the `ViewResult` is processed, and the output is injected into the response to the client. Listing 20-29 shows the changes I have made to the `/Views/Common/List.cshtml` file to render the child action.

Listing 20-29. Calling a Child Action in the List.cshtml File

```
@{
    ViewBag.Title = "List";
    Layout = null;
}

<h3>This is the /Views/Common/List.cshtml View</h3>

@Html.Partial("MyStronglyTypedPartial", new [] {"Apple", "Orange", "Pear"})

@Html.Action("Time")
```

You can see the effect of the child action by starting the application and navigating to the `/Home/List` URL again, as shown in Figure 20-11.



This is the /Views/Common/List.cshtml View

This is the message from the partial view.

- Apple
- Orange
- Pear

The time is: 17:47

Figure 20-11. *Using a child action*

When I called the `Action` helper in Listing 20-29, I provided a single parameter that specified the name of the action method to invoke. This causes the MVC Framework to look for an action method in the controller that is handling the current request. To call action methods in other controllers, provide the controller name, like this:

```
...
@Html.Action("Time", "MyController")
...
```

You can pass parameters to action methods by providing an anonymously typed object whose properties correspond to the names of the child action method parameters. So, for example, if I have this child action:

```
...
[ChildActionOnly]
public ActionResult Time(DateTime time) {
    return PartialView(time);
}
...
```

then I can invoke it from a view as follows:

```
...
@Html.Action("Time", new { time = DateTime.Now })
...
```

Summary

In this chapter, I explored the details of the MVC view system and the Razor View Engine. You have seen how to create a custom view engine, how to customize the behavior of the default Razor engine and the different techniques available for inserting dynamic content into a view. In the next chapter, I focus on helper methods, which assist in generating content that you can insert into your views.