CHAPTER 19

Controller Extensibility

In this chapter, I am going to show you some of the advanced MVC features for working with controllers. I start this chapter by exploring the parts of the request handling process that lead to the execution of an action method and demonstrating the different ways to take control of it. Figure 19-1 shows the basic flow of control between components.

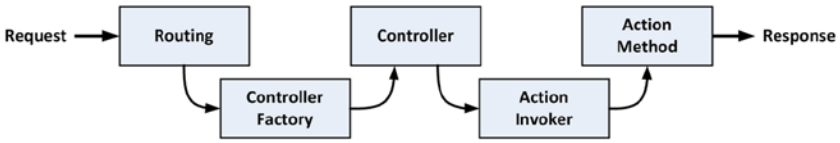

Figure 19-1. *Invoking an action method*

My focus for the first part of this chapter is the *controller factory* and the *action invoker*. The names of these components suggest their purpose. The controller factory is responsible for creating instances of controllers to service a request and the action invoker is responsible for finding and invoking the action method in the controller class. The MVC Framework includes default implementations of both of these components, and I will show you how to configure and control their behavior. I will also show you how to replace these components entirely and use custom logic. Table 19-1 provides the summary for this chapter.

Table 19-1. *Chapter Summary*

Problem	Solution	Listing
Create a custom controller factory	Implement the `IControllerFactory` interface	1–7
Prioritize namespaces in the default controller factory	Use the `DefaultNamespaces` collection	8
Create a custom controller activator	Implement the `IControllerActivator` interface	9–11
Create a custom action invoker	Implement the `IActionInvoker` interface	12–14
Specify an action name that is different from the action method name	Use the `ActionName` attribute	15
Control the selection of action methods	Apply action method selectors	16
Prevent a method from being used as an action	Use the `NoAction` attribute	17

(*continued*)

Table 19-1. (*continued*)

Problem	Solution	Listing
Create a custom action method selector	Derive from the `ActionMethodSelectorAttribute` class	18–21
Respond to requests for non-existent action methods	Override the `HandleUnknownAction` method in the controller	22
Control how controllers use the session feature	Return a value from the `SessionStateBehavior` enumeration in the `IControllerFactory` implementation or apply the `SessionState` attribute to the controller class	23, 24
Prevent controllers from blocking worker threads when waiting for input	Create an asynchronous controller	25–30

Preparing the Example Project

For this chapter, I created a project called `ControllerExtensibility` using the `Empty` template option and enabled the option to add the core `MVC` references and folders. I need some simple controllers to work with in this chapter, so that I can demonstrate the different kinds of extensibility features that are available. To get set up, I created the `Result.cs` file in the `Models` folder and used it to define the `Result` class shown in Listing 19-1.

Listing 19-1. The Contents of the Result.cs File

```
namespace ControllerExtensibility.Models {
    public class Result {
        public string ControllerName { get; set; }
        public string ActionName { get; set; }
    }
}
```

The next step is to create the `/Views/Shared` folder and add a new view called `Result.cshtml`. This is the view that all of the action methods in the controllers will render, and you can see the contents of this file in Listing 19-2.

Listing 19-2. The Contents of the Result.cshtml File

```
@model ControllerExtensibility.Models.Result

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Result</title>
```

```
</head>
<body>
    <div>Controller: @Model.ControllerName</div>
    <div>Action: @Model.ActionName</div>
</body>
</html>
```

This view uses the `Result` class that I defined in Listing 19-2 as its model and displays the values of the `ControllerName` and `ActionName` properties. Finally, I need to create some basic controllers. Listing 19-3 shows the `Product` controller.

Listing 19-3. The Contents of the Product Controller.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class ProductController : Controller {

        public ViewResult Index() {
            return View("Result", new Result {
                ControllerName = "Product",
                ActionName = "Index"
            });
        }

        public ViewResult List() {
            return View("Result", new Result {
                ControllerName = "Product",
                ActionName = "List"
            });
        }
    }
}
```

Listing 19-4 shows the `Customer` controller.

Listing 19-4. The Contents of the CustomerController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class CustomerController : Controller {

        public ViewResult Index() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "Index"
            });
        }
```

```
        public ViewResult List() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "List"
            });
        }
    }
}
```

These controllers do not perform any useful actions other than to report that they have been called via the `Result.cshtml` view.

Setting the Start URL

I want Visual Studio to start with the root URL for the application rather than guess the URL based on the file that is being edited. Select `ControllerExtensibility Properties` from the Visual Studio `Project` menu, switch to the `Web` tab and check the `Specific Page` option in the `Start Action` section. You don't have to provide a value. Just checking the option is enough.

Creating a Custom Controller Factory

As with much of the MVC Framework, the best way to understand how controller factories work is to create a custom implementation. I do not recommend that you do this in a real project, as there are easier ways to create custom behavior by extending the built-in factory. But this is a nice way to demonstrate how the MVC framework creates instances of controllers. Controller factories are defined by the `IControllerFactory` interface, which is shown in Listing 19-5.

Listing 19-5. The IControllerFactory Interface

```
using System.Web.Routing;
using System.Web.SessionState;

namespace System.Web.Mvc {
    public interface IControllerFactory {

        IController CreateController(RequestContext requestContext,
            string controllerName);

        SessionStateBehavior GetControllerSessionBehavior(RequestContext requestContext,
            string controllerName);

        void ReleaseController(IController controller);
    }
}
```

In the sections that follow, I create a simple custom controller factory and walk you through implementations for each of the methods in the `IControllerFactory` interface. To begin, I created an `Infrastructure` folder and added a new class file called `CustomControllerFactory.cs`, which I used to create the custom controller factory shown in Listing 19-6.

Listing 19-6. The Contents of the CustomControllerFactory.cs File

```
using System;
using System.Web.Mvc;
using System.Web.Routing;
using System.Web.SessionState;
using ControllerExtensibility.Controllers;

namespace ControllerExtensibility.Infrastructure {

    public class CustomControllerFactory: IControllerFactory {

        public IController CreateController(RequestContext requestContext,
            string controllerName) {

            Type targetType = null;
            switch (controllerName) {
                case "Product":
                    targetType = typeof(ProductController);
                    break;
                case "Customer":
                    targetType = typeof(CustomerController);
                    break;
                default:
                    requestContext.RouteData.Values["controller"] = "Product";
                    targetType = typeof(ProductController);
                    break;
            }

            return targetType == null ? null :
                (IController)DependencyResolver.Current.GetService(targetType);
        }

        public SessionStateBehavior GetControllerSessionBehavior(RequestContext
            requestContext, string controllerName) {

            return SessionStateBehavior.Default;
        }

        public void ReleaseController(IController controller) {
            IDisposable disposable = controller as IDisposable;
            if (disposable != null) {
                disposable.Dispose();
            }
        }
    }
}
```

The most important method in the interface is `CreateController`, which the MVC Framework calls when it needs a controller to service a request. The parameters to this method are a `RequestContext` object, which allows the factory to inspect details of the request, and a string, which contains the `controller` value from the routed URL. The `RequestContext` class defines the properties described in Table 19-2.

Table 19-2. *RequestContext Properties*

Name	Type	Description
`HttpContext`	`HttpContextBase`	Provides information about the HTTP request
`RouteData`	`RouteData`	Provides information about the route that matches the request

One of the reasons that I do not recommend creating a custom controller this way is that finding controller classes in the web application and instantiating them is complicated. You need to be able to locate controllers dynamically and consistently and deal with all sorts of potential problems, such as disambiguating between classes with the same name in different namespaces, constructor exceptions and a whole lot more.

There are only two controllers in the example project and I am going to instantiate them directly, which means hard-wiring the class names into the controller factory, something which is obviously not a good idea for a real project, but which lets me side-step enormous amounts of complexity.

The purpose of the `CreateController` method is to create instances of controller classes that can handle the current request. There are no restrictions on how you do this. The only rule is that you *must* return an object that implements the `IController` interface as the method result.

The conventions that you have seen so far in this book exist because that's how the default controller factory has been written. As an example, I have implemented one of these conventions in my code: that when I receive a request for a controller, I append `Controller` to the class name, so that a request for `Product` leads to the `ProductController` class being instantiated.

You are free to follow the MVC Framework conventions when you write a controller factory or to discard them and create your own to suit your project needs. I do not think it is sensible to create your own conventions just for the sake of it, but it is useful to understand just how flexible the MVC Framework can be.

Dealing with the Fallback Controller

Custom controller factories must return an implementation of the `IController` interface as the result from the `CreateController` method, otherwise, an error will be displayed to the user. This means that you need to have a fallback position for when the request you are processing does not target any of the controllers in your project. You can create any policy you like for dealing with this situation: you could define a special controller that renders an error message, for example, or do as I have and map the request to a controller class that is known to always exist.

When I get a request that does not map to either of the controllers in the project, I target the `ProductController` class. This may not be the most useful thing to do in a real project, but it demonstrates that the controller factory has complete flexibility in how requests are interpreted. However, you do need to be aware of how the other points in the MVC Framework operate.

By default, the MVC Framework selects a view based on the `controller` value in the routing data, not the name of the controller class. So, in my example, if I want the fallback position to work with views that follow the convention of being organized by controller name, I need to change the value of the `controller` routing property, like this:

```
...
requestContext.RouteData.Values["controller"] = "Product";
...
```

This change will cause the MVC Framework to search for views associated with the fallback controller and not the controller that the routing system has identified based on the URL that the user requested.

There are two important points here: the first is that not only does the controller factory have sole responsibility for matching requests to controllers, but it can *change* the request to alter the behavior of subsequent steps in the request processing pipeline. This is pretty potent stuff and a critical characteristic of the MVC Framework.

The second point is that while you are free to follow whatever conventions you want in your controller factory, you still need to know what the conventions are for other parts of the MVC Framework. And, because those other components can be replaced with custom code as well (as I demonstrate for views in Chapter 20), it makes sense to follow as many of the conventions as possible to allow components to be developed and used independently of one another.

Instantiating Controller Classes

There are no rules about how you instantiate your controller classes, but it is good practice to use the dependency resolver that I introduced in Chapter 6. This allows you to keep your custom controller factory focused on mapping requests to controller classes, and leaves issues like dependency inject to be handled separately and for the entire application. You can see how I used the `DependencyResolver` class to create controllers instances:

```
...
return targetType == null ? null :
    (IController)DependencyResolver.Current.GetService(targetType);
...
```

The static `DependencyResolver.Current` property returns an implementation of the `IDependencyResolver` interface, which defines the `GetService` method. You pass a `System.Type` object to this method and get an instance of it in return. There is a strongly typed version of the `GetService` method, but because I do not know what type I am dealing with in advance, I have to use the version that returns an `Object` and then perform an explicit case to `IController`.

■ **Note** Notice that I am not using the dependency resolver to address tight-coupling issues between classes. Instead, I am asking it to create instances of types that I specify so that it can examine the dependencies that the controller classes have declared and resolve them. I have not set up Ninject in this chapter, which means that the default resolver will be used and that simply creates instances by looking for parameterless constructors and invoking them. However, by building my controller factory to use the `DependencyResolver` class, I ensure that I can seamlessly take advantage of more advanced dependency resolvers like Ninject if one is added to the project.

Implementing the Other Interface Methods

Two other methods are in the `IControllerFactory` interface:

- The `GetControllerSessionBehavior` method is used by the MVC Framework to determine if session data should be maintained for a controller. I will come back to this in the "Using Sessionless Controllers" section later in this chapter.
- The `ReleaseController` method is called when a controller object created by the `CreateController` method is no longer needed. In my example implementation, I check to see if the class implements the `IDisposable` interface. If it does, I call the `Dispose` method to release any resources that can be freed.

My implementations of the `GetControllerSessionBehavior` and `ReleaseController` methods are suitable for most projects and can be used verbatim (although you should read the section on sessionless controllers later in this chapter to make sure you understand the options available).

Registering a Custom Controller Factory

I tell the MVC Framework to use the custom controller factory through the `ControllerBuilder` class. You need to register custom factory controllers when the application is started, which means using the `Application_Start` method in the `Global.asax.cs` file, as shown in Listing 19-7.

Listing 19-7. Registering a Custom Controller Factory 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 ControllerExtensibility.Infrastructure;

namespace ControllerExtensibility {

    public class MvcApplication : System.Web.HttpApplication {

        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ControllerBuilder.Current.SetControllerFactory(new
                CustomControllerFactory());
        }
    }
}
```

Once the controller factory has been registered, it will be responsible for handling all of the requests that the application receives. You can see the effect of the custom factory by starting the application. The browser will request the root URL, which will be mapped to the `Home` controller by the routing system. The custom factory will handle the request for the `Home` controller by creating an instance of the `ProductController` class, which will produce the result shown in Figure 19-2.

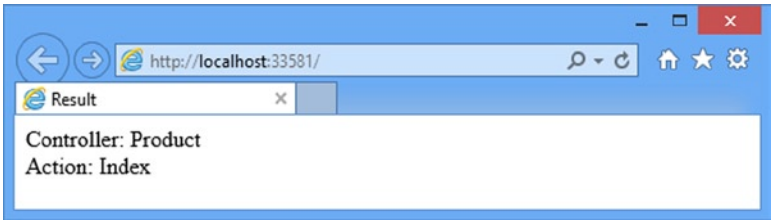

Figure 19-2. *Using the custom controller factory*

Working with the Built-in Controller Factory

I showed you how to create a custom controller factory because it is the most effective way of demonstrating what a controller factory does and how it functions. For most applications, however, the built-in controller factory class, called `DefaultControllerFactory`, is perfectly adequate. When it receives a request from the routing system, this

factory looks at the routing data to find the value of the `controller` property and tries to find a class in the Web application that meets the following criteria:

- The class must be `public`.
- The class must be concrete (not `abstract`).
- The class must *not* take generic parameters.
- The name of the class must end with `Controller`.
- The class must implement the `IController` interface.

The `DefaultControllerFactory` class maintains a list of such classes in the application so that it does not need to perform a search every time a request arrives. If a suitable class is found, then an instance is created using the controller activator (I will come back to this in the upcoming "Customizing DefaultControllerFactory Controller Creation" section), and the job of the controller is complete. If there is no matching controller, then the request cannot be processed any further.

Notice how the `DefaultControllerFactory` class follows the convention-over-configuration pattern. You do not need to register your controllers in a configuration file, because the factory will find them for you. All you need to do is create classes that meet the criteria that the factory is seeking.

If you want to create custom controller factory behavior, you can configure the settings of the default factory or override some of the methods. This way, you are able to build on the useful convention-over-configuration behavior without having to re-create it, a task which, I noted earlier, is complicated and painful. In the sections that follow, I show you different ways to tailor controller creation.

Prioritizing Namespaces

In Chapter 16, I showed you how to prioritize one or more namespaces when creating a route. This was to address the ambiguous controller problem, where controller classes have the same name but reside in different namespaces. It is the `DefaultControllerFactory` that processes the list of namespaces and prioritizes them.

■ **Tip** Global prioritization is overridden by route-specific prioritization. This means you can define a global policy, and then tailor individual routes as required. See Chapter 16 for details on specifying namespaces for individual routes.

If you have an application that has a lot of routes, it can be more convenient to specify priority namespaces globally, so that they are applied to all of your routes. Listing 19-8 shows how to do this in the `Application_Start` method of the `Global.asax` file. (This is where I put these statements, but you can also use the `RouteConfig.cs` file in the `App_Start` folder if you prefer.)

Listing 19-8. Global Namespace Prioritization 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 ControllerExtensibility.Infrastructure;
```

```
namespace ControllerExtensibility {

    public class MvcApplication : System.Web.HttpApplication {

        protected void Application_Start() {

            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ControllerBuilder.Current.DefaultNamespaces.Add("MyControllerNamespace");
            ControllerBuilder.Current.DefaultNamespaces.Add("MyProject.*");
        }
    }
}
```

I use the static `ControllerBuilder.Current.DefaultNamespaces.Add` method to add namespaces that should be given priority. The order in which I add the namespaces does not imply any kind of search order or relative priority. All of the namespaces defined by the `Add` method are treated equally and the priority is relative to those namespaces which have not been specified by the `Add` method. This means that the controller factory will search the entire application if it can't find a suitable controller class in the namespaces defined by the `Add` method.

■ **Tip** Notice that I used an asterisk character (*) in the second statement shown in bold in Listing 19-8. This allows me to specify that the controller factory should look in the `MyProject` namespace and any child namespaces that `MyProject` contains. Although this looks like regular expression syntax, it isn't; you can end your namespaces with `.*`, but you cannot use any other regular expression syntax with the `Add` method.

Customizing DefaultControllerFactory Controller Instantiation

There are a number of ways to customize how the `DefaultControllerFactory` class instantiates controller objects. By far, the most common reason for customizing the controller factory is to add support for DI. There are several different ways of doing this. The most suitable technique depends on how you are using DI elsewhere in your application.

Using the Dependency Resolver

The `DefaultControllerFactory` class will use a dependency resolver to create controllers if one is available. I covered dependency resolvers in Chapter 6 and showed you the `NinjectDependencyResolver` class, which implements the `IDependencyResolver` interface to provide Ninject DI support. I also demonstrated how to use the `DependencyResolver` class earlier in this chapter when I created my own custom controller factory. The `DefaultControllerFactory` will call the `IDependencyResolver.GetService` method to request a controller instance, which gives you the opportunity to resolve and inject any dependencies.

Using a Controller Activator

You can also introduce DI into controllers by creating a *controller activator*. You create this activator by implementing the `IControllerActivator` interface, as shown in Listing 19-9.

Listing 19-9. The IControllerActivator Interface

```
namespace System.Web.Mvc {
    using System.Web.Routing;

    public interface IControllerActivator {
        IController Create(RequestContext requestContext, Type controllerType);
    }
}
```

The interface contains one method, called `Create`, which is passed a `RequestContext` object describing the request and a `Type` that specifies which controller class should be instantiated.

To demonstrate an implementation of this interface, I added a new class file called `CustomControllerActivator.cs` in the `Infrastructure` folder and used it to define the class shown in Listing 19-10.

Listing 19-10. The Contents of the CustomControllerActivator.cs File

```
using System;
using System.Web.Mvc;
using System.Web.Routing;
using ControllerExtensibility.Controllers;

namespace ControllerExtensibility.Infrastructure {
    public class CustomControllerActivator : IControllerActivator {

        public IController Create(RequestContext requestContext,
            Type controllerType) {

            if (controllerType == typeof(ProductController)) {
                controllerType = typeof(CustomerController);
            }
            return (IController)DependencyResolver.Current.GetService(controllerType);
        }
    }
}
```

This `IControllerActivator` implementation is simple. If the `ProductController` class is requested, it responds with an instance of the `CustomerController` class. This is not something you would want to do in a real project, but it demonstrates how you can use the `IControllerActivator` interface to intercept requests between the controller factory and the dependency resolver.

To use a custom activator, I need to pass an instance of the implementation class to the `DefaultControllerFactory` constructor and register the result in the `Application_Start` method of the `Global.asax` file, as shown in Listing 19-11.

Listing 19-11. Registering a Custom Activator 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 ControllerExtensibility.Infrastructure;
```

```
namespace ControllerExtensibility {

    public class MvcApplication : System.Web.HttpApplication {

        protected void Application_Start() {

            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ControllerBuilder.Current.SetControllerFactory(new
                DefaultControllerFactory(new CustomControllerActivator()));
        }
    }
}
```

You can see the effect of the custom activator if you start the application and navigate to the `/Product` URL. The route will target the `Product` controller and the `DefaultControllerFactory` will ask the activator to instantiate the `ProductFactory` class, but my activator intercepts this request and creates an instance of the `CustomerController` class instead, as shown in Figure 19-3.

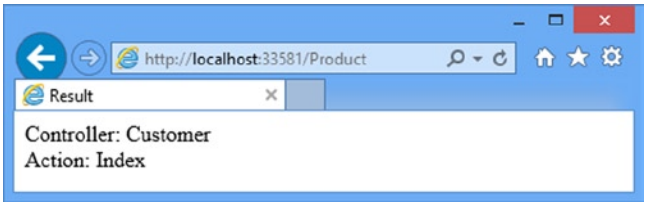

Figure 19-3. *Intercepting instantiation requests using a custom controller activator*

Overriding DefaultControllerFactory Methods

You can override methods in the `DefaultControllerFactory` class to customize the creation of controllers. Table 19-3 describes the three methods you can override, each of which performs a different role.

Table 19-3. *Overridable DefaultContollerFactory Methods*

Method	Result	Description
`CreateController`	`IController`	The implementation of the `CreateController` method from the `IControllerFactory` interface. By default, this method calls `GetControllerType` to determine which type should be instantiated, and then gets a controller object by passing the result to the `GetControllerInstance` method.
`GetControllerType`	`Type`	Maps requests to controller types. This is where most of the conventions listed earlier in the chapter are enforced.
`GetControllerInstance`	`IController`	Creates an instance of a specified type.

Creating a Custom Action Invoker

Once the controller factory has created an instance of a class, the framework needs a way of invoking an action on that instance. If you derived your controller from the `Controller` class, then this is the responsibility of an *action invoker*, which is the subject of this section.

■ **Tip** If you create a controller directly from the `IController` interface, then you are responsible for executing the action yourself. (See details of creating controllers in this way.) Action invokers are part of the functionality included in the `Controller` class.

An action invoker implements the `IActionInvoker` interface, which is shown in Listing 19-12.

Listing 19-12. The IActionInvoker Interface

```
namespace System.Web.Mvc {

    public interface IActionInvoker {

        bool InvokeAction(ControllerContext controllerContext, string actionName);
    }
}
```

The interface has only a single member: `InvokeAction`. The parameters are a `ControllerContext` object (which I described in Chapter 17) and a string that contains the name of the action to be invoked. The result type is a `bool`: a value of `true` indicates that the action was found and invoked and `false` indicates that the controller has no matching action.

Notice that I have not used the word *method* in this description. The association between actions and methods is strictly optional. Although this is the approach that the built-in action invoker takes, you are free to handle actions any way that you choose. Listing 19-13 shows an implementation of the `IActionInvoker` interface that takes a different approach, which I defined in a class file called `CustomActionInvoker.cs` in the `Infrastructure` folder.

Listing 19-13. The Contents of the CustomActionInvoker.cs File

```
using System.Web.Mvc;

namespace ControllerExtensibility.Infrastructure {

    public class CustomActionInvoker : IActionInvoker {

        public bool InvokeAction(ControllerContext controllerContext,
                string actionName) {

            if (actionName == "Index") {
                controllerContext.HttpContext.
                    Response.Write("This is output from the Index action");
                return true;
            } else {
                return false;
            }
        }
    }
}
```

This action invoker doesn't care about the methods in the controller class. In fact, it deals with actions itself. If the request is for the `Index` action, then the invoker writes a message directly to the `Response`. If the request is for any other action, then it returns `false`, which causes a `404–Not found` error to be displayed to the user.

The action invoker associated with a controller is obtained through the `Controller.ActionInvoker` property. This means that different controllers in the same application can use different action invokers. To demonstrate this, I have added a new controller to the example project called `ActionInvoker`, the definition of which you can see in Listing 19-14.

Listing 19-14. The Contents of the ActionInvokerController.cs File

```
using ControllerExtensibility.Infrastructure;
using System.Web.Mvc;

namespace ControllerExtensibility.Controllers {

    public class ActionInvokerController : Controller {
        public ActionInvokerController() {
            this.ActionInvoker = new CustomActionInvoker();
        }
    }
}
```

There are no action methods in this controller. It depends on the action invoker to process requests. You can see how this works by starting the application and navigating to the `/ActionInvoker/Index` URL. The custom action invoker will generate the response shown in Figure 19-4. If you navigate to a URL that targets any other action on the same controller, you will see the 404 error page.

Figure 19-4. *The effect of a custom action invoker*

I am not suggesting that you implement your own action invoker. And, if you do, I do not suggest you follow this approach. Why? First, the built-in support has some useful features, as you will see shortly. Second, the example has some problems: a lack of extensibility, poor separation of responsibilities, and a lack of support for views of any kind. But the example shows how the MVC Framework fits together and demonstrates, once again, that almost every aspect of the request processing pipeline can be customized or replaced entirely.

Using the Built-in Action Invoker

The built-in action invoker, which is the `ControllerActionInvoker` class, has some sophisticated techniques for matching requests to actions. And, unlike my implementation in the previous section, the default action invoker operates on methods. To qualify as an action, a method must meet the following criteria:

- The method must be `public`.
- The method must *not* be `static`.

- The method must *not* be present in `System.Web.Mvc.Controller` or any of its base classes.
- The method must *not* have a special name.

The first two criteria are simple enough. For the next, excluding any method that is present in the `Controller` class or its bases means that methods such as `ToString` and `GetHashCode` are excluded, as are the methods that implement the `IController` interface. This is sensible, because the inner workings of controllers should not be exposed to the outside world. The last criterion means that constructors, property and event accessors are excluded. In fact, no class member that has the `IsSpecialName` flag from `System.Reflection.MethodBase` will be used to process an action.

■ **Note** Methods that have generic parameters (such as `MyMethod<T>()`) meet all of the criteria, but the MVC Framework will throw an exception if you try to invoke such a method to process a request.

By default, the `ControllerActionInvoker` finds a method that has the same name as the requested action. So, for example, if the `action` value that the routing system produces is `Index`, then the `ControllerActionInvoker` will look for a method called `Index` that fits the action criteria. If it finds such a method, it will be invoked to handle the request. This behavior is exactly what you want almost all of the time, but as you might expect, the MVC Framework provides some opportunities to fine-tune the process.

Using a Custom Action Name

Usually, the name of an action method determines the action that it represents. The `Index` action method services requests for the `Index` action. You can override this behavior using the `ActionName` attribute, which I have applied to the `Customer` controller as shown in Listing 19-15.

Listing 19-15. Using a Custom Action Name in the CustomerController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class CustomerController : Controller {

        public ViewResult Index() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "Index"
            });
        }

        [ActionName("Enumerate")]
        public ViewResult List() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "List"
            });
        }
    }
}
```

In this listing, I have applied the attribute to the `List` method, passing in a parameter value of `Enumerate`. When the action invoker receives a request for the `Enumerate` action, it will now use the `List` method to service it. You can see the effect of the `ActionName` attribute by starting the application and navigating to the `/Customer/Enumerate` URL. You can see that the results shown by the browser in Figure 19-5 are those from the `List` method.

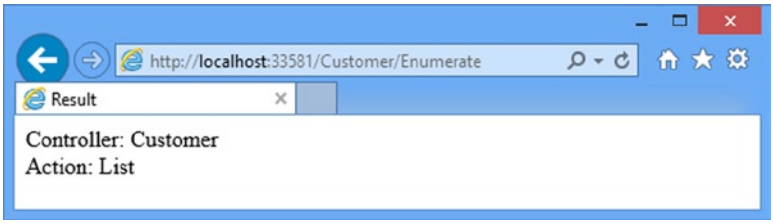

Figure 19-5. *The effect of the ActionName attribute*

Applying the attribute overrides the name of the action. This means that URLs which directly target the `List` method will no longer work, as shown in Figure 19-6.

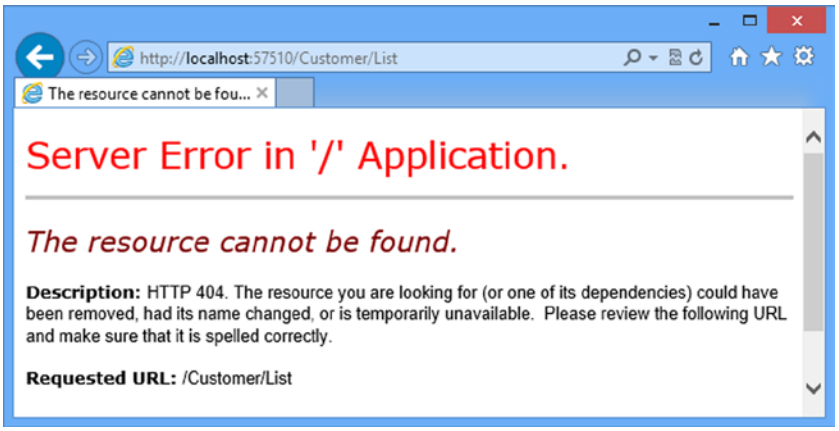

Figure 19-6. *Using the method name as the action when the ActionName attribute has been applied*

There are two main reasons why you might want to override a method name in this way:

- You can then accept an action name that wouldn't be legal as a C# method name (for example, `[ActionName("User-Registration")]`).
- If you want to have two different C# methods that accept the same set of parameters and should handle the same action name, but in response to different HTTP request types (for example, one with `[HttpGet]` and the other with `[HttpPost]`), you can give the methods different C# names to satisfy the compiler, but then use `[ActionName]` to map them both to the same action name.

Using Action Method Selection

It is often the case that a controller will contain several actions with the same name. This can be because there are multiple methods, each with different parameters, or because you used the `ActionName` attribute so that multiple methods represent the same action.

In these situations, the MVC Framework needs some help selecting the appropriate action with which to process a request. The mechanism for doing this is called *action method selection*. It allows you to define kinds of requests that an action is willing to process. You have already seen an example of action method selection when I restricted an action using the `HttpPost` attribute when I built the SportsStore application. I had two methods called `Checkout` in the `Cart` controller and I used the `HttpPost` attribute to indicate that one of them was to be used only for HTTP `POST` requests, as shown in Listing 19-16.

Listing 19-16. Using the HttpPost Attribute

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;
        private IOrderProcessor orderProcessor;

        public CartController(IProductRepository repo, IOrderProcessor proc) {
            repository = repo;
            orderProcessor = proc;
        }

        // ...other action methods omitted for brevity...

        public ViewResult Checkout() {
            return View(new ShippingDetails());
        }

        [HttpPost]
        public ViewResult Checkout(Cart cart, ShippingDetails shippingDetails) {
            if (cart.Lines.Count() == 0) {
                ModelState.AddModelError("", "Sorry, your cart is empty!");
            }

            if (ModelState.IsValid) {
                orderProcessor.ProcessOrder(cart, shippingDetails);
                cart.Clear();
                return View("Completed");
            } else {
                return View(shippingDetails);
            }
        }
    }
}
```

The action invoker uses action method selectors to resolve ambiguity when selecting an action. In Listing 19-16, there are two candidates for the Checkout action. The invoker gives preference to the actions that have selectors. In this case, the HttpPost selector is evaluated to see if the request can be processed. If it can, then this is the method that will be used. If not, then the *other* method, the one without the attribute, will be used.

There are built-in attributes that work as selectors for the different kinds of HTTP requests: HttpPost for POST requests, HttpGet for GET requests, HttpPut for PUT requests, and so on. Another built-in attribute is NonAction, which indicates to the action invoker that a method that would otherwise be considered a valid action method should not be used. You can see how I have applied the NonAction attribute in Listing 19-17, where I have defined a new action method in the Customer controller.

Listing 19-17. Using the NonAction Selector in the CustomerController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class CustomerController : Controller {

        public ViewResult Index() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "Index"
            });
        }

        [ActionName("Enumerate")]
        public ViewResult List() {
            return View("Result", new Result {
                ControllerName = "Customer",
                ActionName = "List"
            });
        }

        [NonAction]
        public ActionResult MyAction() {
            return View();
        }
    }
}
```

The MyAction method in the listing will not be considered as an action method, even though it meets all of the criteria that the invoker looks for. This is useful for ensuring that you do not expose the workings of your controller classes as actions. Of course, normally such methods should simply be marked private, which will prevent them from being invoked as actions; however, [NonAction] is useful if for some reason you must mark such a method as public. Requests for URLs that target NonAction methods will generate 404–Not Found errors, as shown in Figure 19-7.

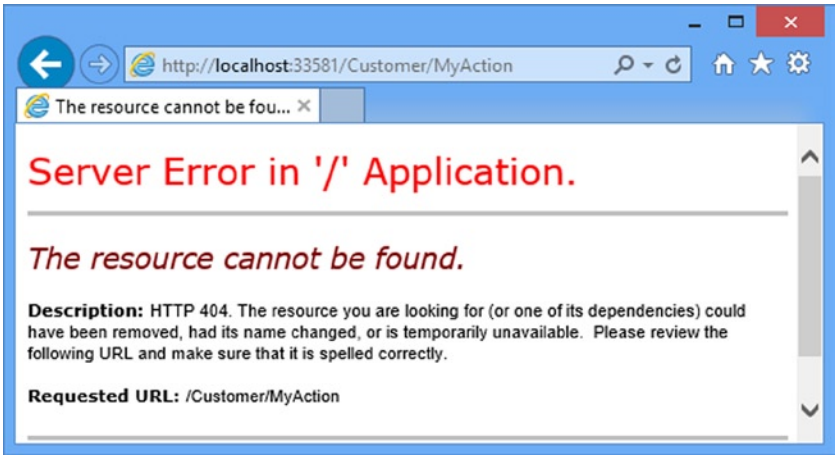

Figure 19-7. *The effect of requesting a URL that targets a NonAction method*

Creating a Custom Action Method Selector

Action method selectors are derived from the `ActionMethodSelectorAttribute` class, which is shown in Listing 19-18.

Listing 19-18. The ActionMethodSelectorAttribute Class

```
using System.Reflection;

namespace System.Web.Mvc {
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = false, Inherited = true)]
    public abstract class ActionMethodSelectorAttribute : Attribute {

        public abstract bool IsValidForRequest(ControllerContext controllerContext,
            MethodInfo methodInfo);
    }
}
```

The `ActionMethodSelectorAttribute` is abstract and defines one abstract method: `IsValidForRequest`. The parameters for this method are a `ControllerContext` object, which allows you to inspect the request, and a `MethodInfo` object, which you can use to get information about the method to which your selector has been applied. You return `true` from `IsValidForRequest` if the method is able to process a request, and `false` otherwise. I created a simple custom action method selector in a class file, `LocalAttribute.cs`, that I added to the `Infrastructure` folder of the example project, as shown in Listing 19-19.

Listing 19-19. The Contents of the LocalAttribute.cs File

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

namespace ControllerExtensibility.Infrastructure {
    public class LocalAttribute : ActionMethodSelectorAttribute {
```

```
        public override bool IsValidForRequest(ControllerContext controllerContext,
                MethodInfo methodInfo) {
            return controllerContext.HttpContext.Request.IsLocal;
        }
    }
}
```

I have overridden the `IsValidForRequest` method so that it returns `true` when the request originates from the local machine. To demonstrate the custom action method selector, I created a `Home` controller in the example project, as shown in Listing 19-20.

Listing 19-20. The Contents of the HomeController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Infrastructure;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            return View("Result", new Result {
                ControllerName = "Home", ActionName = "Index"
            });
        }

        [ActionName("Index")]
        public ActionResult LocalIndex() {
            return View("Result", new Result {
                ControllerName = "Home", ActionName = "LocalIndex"
            });
        }
    }
}
```

I have used the `ActionName` attribute to create a situation in which there are two `Index` action methods. At this point, the action invoker doesn't have any way to figure out which one should be used when a request for the `/Home/Index` URL arrives and will generate the error shown in Figure 19-8 when such a request is received.

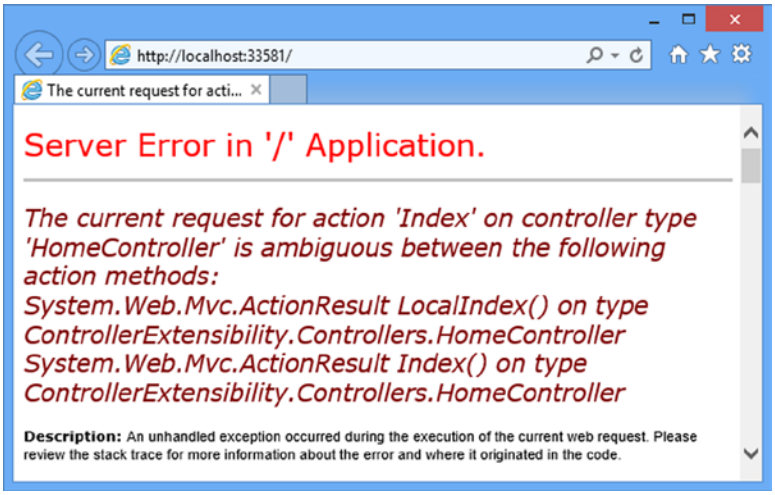

Figure 19-8. *The error shown when there are ambiguous action method names*

To resolve this situation, I can apply the custom method selection attribute to one of the ambiguous methods, as shown in Listing 19-21.

Listing 19-21. Applying the Method Selection Attribute to the HomeController.cs File

```
...
[Local]
[ActionName("Index")]
public ActionResult LocalIndex() {
    return View("Result", new Result {
        ControllerName = "Home", ActionName = "LocalIndex"
    });
}
...
```

If you restart the application and navigate to the root URL from a browser running on the local machine, you will see that the MVC Framework takes the method selection attribute into account to resolve the ambiguity between the methods in the controller class, as shown in Figure 19-9.

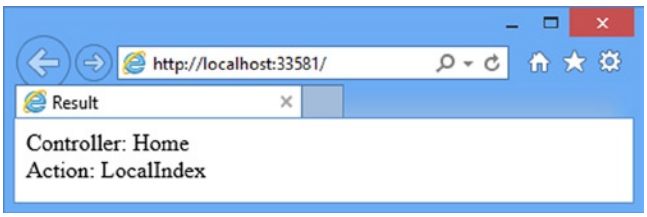

Figure 19-9. *Using a method selection attribute to resolve action method ambiguity*

THE ACTION METHOD DISAMBIGUATION PROCESS

Now that you have seen inside the action method selector base class, you can understand how the action invoker selects an action method. The invoker starts the process with a list of possible candidates, which are the controller methods that meet the action method criteria. Then it goes through the following process:

- The invoker discards any method based on name. Only methods that have the same name as the target action or have a suitable `ActionName` attribute are kept on the list.
- The invoker discards any method that has an action method selector attribute that returns `false` for the current request.
- If there is exactly one action method with a selector left, then this is the method that is used. If there is more than one method with a selector, then an exception is thrown, because the action invoker cannot disambiguate between the available methods.
- If there are no action methods with selectors, then the invoker looks at those without selectors. If there is exactly one such method, then this is the one that is invoked. If there is more than one method without a selector, an exception is thrown, because the invoker can't choose between them.

Handling Unknown Actions

If the action invoker is unable to find an action method to invoke, it returns `false` from its `InvokeAction` method. When this happens, the `Controller` class calls its `HandleUnknownAction` method. By default, this method returns a `404–Not Found` response to the client. This is the most useful thing that a controller can do for most applications, but you can choose to override this method in your controller class if you want to do something special. Listing 19-22 provides a demonstration of overriding the `HandleUnknownAction` method in the `Home` controller.

Listing 19-22. Overriding the HandleUnknownAction Method in the HomeController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Infrastructure;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class HomeController : Controller {

        // ...other action methods omitted for brevity...

        protected override void HandleUnknownAction(string actionName) {
            Response.Write(string.Format("You requested the {0} action", actionName));
        }
    }
}
```

If you start the application and navigate to a URL that targets a nonexistent action method, you will see the response shown in Figure 19-10.

Figure 19-10. *Dealing with requests for action methods that do not exist*

Improving Performance with Specialized Controllers

The MVC Framework provides two special kinds of controllers that may improve the performance of your MVC web applications. Like all performance optimizations, these controllers represent compromises, either in ease of use or with reduced functionality. In the follow sections, I demonstrate both kinds of controllers and outline their benefits and shortcomings.

Using Sessionless Controllers

By default, controllers support *session state*, which can be used to store data values across requests, making life easier for the MVC programmer. Creating and maintaining session state is an involved process. Data must be stored and retrieved, and the sessions themselves must be managed so that they expire appropriately. Session data consumes server memory or space in some other storage location, and needing to synchronize the data across multiple Web servers makes it harder to run your application on a server farm.

In order to simplify session state, ASP.NET will process only one query for a given session at a time. If the client makes multiple overlapping requests, they will be queued up and processed sequentially by the server. The benefit is that you do not need to worry about multiple requests modifying the same data. The downside is that you do not get the request throughput you might like.

Not all controllers need the session state features. In such cases, you can improve the performance of your application by avoiding work involved in maintaining session state. You do this by using *sessionless controllers.* These are just like regular controllers, with two exceptions: the MVC Framework will not load or store session state when they are used to process a request, and overlapping requests can be processed simultaneously.

Managing Session State in a Custom IControllerFactory

At the start of this chapter, I showed you that the `IControllerFactory` interface contained a method called `GetControllerSessionBehavior`, which returns a value from the `SessionStateBehavior` enumeration. That enumeration contains four values that control the session state configuration of a controller, as described in Table 19-4.

Table 19-4. *The Values of the SessionStateBehavior Enumeration*

Value	Description
`Default`	Use the default ASP.NET behavior, which is to determine the session state configuration from `HttpContext`.
`Required`	Full read-write session state is enabled.
`ReadOnly`	Read-only session state is enabled.
`Disabled`	Session state is disabled entirely.

A controller factory that implements the IControllerFactory interface directly sets the session state behavior for controllers by returning SessionStateBehavior values from the GetControllerSessionBehavior method. The parameters to this method are a RequestContext object and a string containing the name of the controller. You can return any of the four values shown in the table, and you can return different values for different controllers. As a demonstration, I have changed the implementation of the GetControllerSessionBehavior method in the CustomControllerFactory class that I created earlier in the chapter, as shown in Listing 19-23.

Listing 19-23. Defining Session State Behavior for a Controller in the CustomControllerFactory.cs File

```
...
public SessionStateBehavior GetControllerSessionBehavior(RequestContext
        requestContext, string controllerName) {
    switch (controllerName) {
        case "Home":
            return SessionStateBehavior.ReadOnly;
        case "Product":
            return SessionStateBehavior.Required;
        default:
            return SessionStateBehavior.Default;
    }
}
...
```

Managing Session State Using DefaultControllerFactory

When you are using the built-in controller factory, you can control the session state by applying the SessionState attribute to individual controller classes, as shown in Listing 19-24 where I have created a new controller called FastController.

Listing 19-24. Using the SessionState Attribute in the FastController.cs File

```
using System.Web.Mvc;
using System.Web.SessionState;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {

    [SessionState(SessionStateBehavior.Disabled)]
    public class FastController : Controller {

        public ActionResult Index() {
            return View("Result", new Result {
                ControllerName = "Fast ",ActionName = "Index"
            });
        }
    }
}
```

The SessionState attribute is applied to the controller class and affects all of the actions in the controller. The sole parameter to the attribute is a value from the SessionStateBehavior enumeration. In the example, I disabled session state entirely, which means that if I try to set a session value in the controller, like this:

```
...
Session["Message"] = "Hello";
...
```

or try to read back from the session state in a view, like this:

```
...
Message: @Session["Message"]
...
```

The MVC Framework will throw an exception when the action is invoked or the view is rendered.

■ **Tip** When session state is Disabled, the HttpContext.Session property returns null.

If you have specified the ReadOnly behavior, then you can read values that have been set by other controllers, but you will still get a runtime exception if you try to set or modify a value. You can get details of the session through the HttpContext.Session object but trying to alter any values causes an error.

■ **Tip** If you are simply trying to pass data from the controller to the view, consider using the View Bag feature instead, which is not affected by the SessionState attribute.

Using Asynchronous Controllers

The underlying ASP.NET platform maintains a pool of .NET threads that are used to process client requests. This pool is called the *worker thread pool*, and the threads are called *worker threads*. When a request is received, a worker thread is taken from the pool and given the job of processing the request. When the request has been processed, the worker thread is returned to the pool, so that it is available to process new requests as they arrive. There are two key benefits of using thread pools for ASP.NET applications:

- By reusing worker threads, you avoid the overhead of creating a new one each time you process a request.
- By having a fixed number of worker threads available, you avoid the situation where you are processing more simultaneous requests than your server can handle.

The worker thread pool works best when requests can be processed in a short period of time. This is the case for most MVC applications. However, if you have actions that depend on other servers and take a long time to complete, then you can reach the point where all of your worker threads are tied up waiting for other systems to complete their work.

■ **Note** In this section, I assume that you are familiar with the Task Parallel Library (TPL). If you want to learn about the TPL, see my book on the topic, called *Pro .NET Parallel Programming in C#*, which is published by Apress.

Your server is capable of doing more work (after all, you are just waiting, which takes up little of your resources), but because you have tied up all of your worker threads, incoming requests are being queued up. You will be in the odd state of your application grinding to a halt while the server is largely idle.

■ **Caution** At this point, some readers are thinking that they can write a worker thread pool that is tailored to their application. *Do not do it*. Writing concurrent code is easy. Writing concurrent code *that works* is difficult. If you are new to concurrent programming, then you lack the required skills. My advice is to stick with the default pool. If you are experienced in concurrent programming, then you already know that the benefits will be marginal compared with the effort of coding and testing a new thread pool.

The solution to this problem is to use an *asynchronous controller*. This increases the overall performance of your application, but does not bring any benefits to the execution of your asynchronous operations.

■ **Note** Asynchronous controllers are useful only for actions that are I/O- or network-bound and *not* CPU-intensive. The problem you are trying to solve with asynchronous controllers is a mismatch between the pool model and the type of request you are processing. The pool is intended to ensure that each request gets a decent slice of the server resources, but you end up with a set of worker threads that are doing nothing. If you use additional background threads for CPU-intensive actions, then you will dilute the server resources across too many simultaneous requests.

Creating the Example

To begin the exploration of asynchronous controllers, I am going to show you an example of the kind of problem that they are intended to solve. Listing 19-25 shows a regular synchronous controller called `RemoteData` that I added to the example project.

Listing 19-25. The Contents of the RemoteDataController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers {
    public class RemoteDataController : Controller {

        public ActionResult Data() {
            RemoteService service = new RemoteService();
            string data = service.GetRemoteData();
            return View((object)data);
        }
    }
}
```

This controller contains an action method, Data, which creates an instance of the model class RemoteService and calls the GetRemoteData method on it. This method is an example of a time-consuming, low-CPU activity. The RemoteService class, which I defined in a class file called RemoteService.cs in the Models folder, is shown in Listing 19-26.

Listing 19-26. The Contents of the RemoteService.cs File

```
using System.Threading;

namespace ControllerExtensibility.Models {
    public class RemoteService {

        public string GetRemoteData() {
            Thread.Sleep(2000);
            return "Hello from the other side of the world";
        }
    }
}
```

Okay, I admit it: I faked the GetRemoteData method. In the real world, this method could be retrieving complex data across a slow network connection, but to keep things simple, I used the Thread.Sleep method to simulate a two-second delay. The last addition I need is a new view. I created the Views/RemoteData folder and added the Data.cshtml view file to it, the contents of which are shown in Listing 19-27.

Listing 19-27. The Contents of the Data.cshtml File

```
@model string

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Data</title>
</head>
<body>
    <div>
        Data: @Model
    </div>
</body>
</html>
```

When you run the application and navigate to the /RemoteData/Data URL, the action method is invoked, the RemoteService object is created, and the GetRemoteData method is called. After two seconds (simulating a real operation), the data is returned from the GetRemoteData method, passed to the view and rendered as Figure 19-11.

Figure 19-11. *Navigating to the /RemoteData/Data URL*

The problem here is that the worker thread that was handling the request was idle for two seconds. It wasn't doing anything useful, and it was not available for handling other requests while it was waiting.

■ **Caution** Using an asynchronous controller frees up the worker thread so that it can process other queries. It doesn't prevent the user from experiencing a two-second wait. After all, that fake data still has to be obtained and processed. There are client-side techniques you can use to make such requests asynchronously in the browser, which allows you to at least keep the user informed about the progress of getting the data and allow them to continue working with another part of the application. See my *Pro ASP.NET MVC 5 Client* book, published by Apress in 2014, for details.

Creating an Asynchronous Controller

Having shown you the problem I was going to solve, I can now move on to create the asynchronous controller. There are two ways to create an asynchronous controller. One is to implement the `System.Web.Mvc.Async.IAsyncController` interface, which is the asynchronous equivalent of `IController`. I am not going to demonstrate that approach, because it requires so much explanation of the .NET concurrent programming facilities.

■ **Tip** Not all actions in an asynchronous controller need to be asynchronous. You can include synchronous methods as well, and they will behave as expected.

I want to stay focused on the MVC Framework, which is why I will demonstrate the second approach: to use the new `await` and `async` keywords in a regular controller.

In previous versions of the .NET Framework, creating asynchronous controllers was a complex process and required deriving the controller from a special class and splitting actions into two methods. The new `await` and `async` keywords, which I described in Chapter 4, have simplified this process a lot: you create a new `Task` object and `await` its response, as shown in Listing 19-28.

■ **Tip** The old method of creating asynchronous action methods is still supported, although the approach I describe here is much more elegant and the one I recommend. One artifact of the old approach is that you can't use action method names that end with `Async` (e.g., `IndexAsync`) or `Completed` (e.g., `IndexCompleted`).

Listing 19-28. Creating an Asynchronous Controller in the RemoteDataController.cs File

```
using System.Web.Mvc;
using ControllerExtensibility.Models;
using System.Threading.Tasks;

namespace ControllerExtensibility.Controllers {
    public class RemoteDataController : Controller {

        public async Task<ActionResult> Data() {
            string data = await Task<string>.Factory.StartNew(() => {
                return new RemoteService().GetRemoteData();
            });

            return View((object)data);
        }
    }
}
```

I have refactored the action method so that it returns a `Task<ActionResult>`, applied the `async` and `await` keywords, and created a `Task<string>`, which is responsible for calling the `GetRemoteData` method.

Consuming Asynchronous Methods in a Controller

You can also use an asynchronous controller to consume asynchronous methods elsewhere in your application. To demonstrate this, I have added an asynchronous method to the `RemoteService` class, as shown in Listing 19-29.

Listing 19-29. Adding an Asynchronous Method in the RemoteService.cs File

```
using System.Threading;
using System.Threading.Tasks;

namespace ControllerExtensibility.Models {
    public class RemoteService {

        public string GetRemoteData() {
            Thread.Sleep(2000);
            return "Hello from the other side of the world";
        }

        public async Task<string> GetRemoteDataAsync() {
            return await Task<string>.Factory.StartNew(() => {
                Thread.Sleep(2000);
                return "Hello from the other side of the world";
            });
        }
    }
}
```

The result from the `GetRemoteDataAsync` method is a `Task<string>`, which yields the same message as the synchronous method when it is completed. In Listing 19-30, you can see how I have consumed this asynchronous method in a new action method that I added to the `RemoteData` controller.

Listing 19-30. Consuming Asynchronous Methods in the RemoteData Controller

```
using System.Web.Mvc;
using ControllerExtensibility.Models;
using System.Threading.Tasks;

namespace ControllerExtensibility.Controllers {
    public class RemoteDataController : Controller {

        public async Task<ActionResult> Data() {
            string data = await Task<string>.Factory.StartNew(() => {
                return new RemoteService().GetRemoteData();
            });

            return View((object)data);
        }

        public async Task<ActionResult> ConsumeAsyncMethod() {
            string data = await new RemoteService().GetRemoteDataAsync();
            return View("Data", (object)data);
        }
    }
}
```

You can see that both action methods follow the same basic pattern and that the difference is where the `Task` object is created. The result of calling either action method is that the worker thread is not tied up while I wait for the `GetRemoteData` call to complete, which means that the thread is available to process other requests which can significantly improve the performance of your MVC Framework application.

Summary

In this chapter, you have seen how the MVC Framework creates controllers and invokes methods. I have explored and customized the built-in implementations of the key interfaces, and created custom versions to demonstrate how they work. You have learned how action method selectors can be used to differentiate between action methods and seen some specialized kinds of controllers that can be used to increase the request processing capability of your applications.

The underlying theme of this chapter is extensibility. Almost every aspect of the MVC Framework can be modified or replaced entirely. For most projects, the default behaviors are entirely sufficient. But having a working knowledge of how the MVC Framework fits together helps you to make informed design and coding decisions (and it is just plain interesting).

In the next chapter, I turn to views. I explain how they work and, as you will have come to expect by now, how to configure and customize the default behaviors.