With MvcContrib, NHibernate, and more

ASP.NET MVC IN ACTION

Jeffrey Palermo
Ben Scheirman
Jimmy Bogard

FOREWORD BY PHIL HAACK

SAMPLE CHAPTER

MANNING

ASP.NET MVC in Action

WITH MVCCONTRIB, NHIBERNATE, AND MORE

JEFFREY PALERMO
BEN SCHEIRMAN
JIMMY BOGARD

MANNING
Greenwich
(74° w. long.)

ASP.NET MVC in Action

by Jeffrey Palermo
Ben Scheirman
and Jimmy Bogard

Chapter 3

brief contents

3 The controller in depth

This chapter covers

- Understanding controller anatomy
- Leveraging viewless controllers
- Testing controllers
- Using form and querystring values
- Binding action parameter
- Developing action filters

The focus of the Model-View-Controller pattern is the controller. With this pattern, every request is handled by a controller and rendered by a view. Without the controller, presentation and business logic would move to the view, as we have seen with Web Forms. With the ASP.NET MVC Framework, every request routes to a controller, which is simply a class that implements the `IController` interface. Microsoft provides the base class `System.Web.Mvc.Controller` to make creating a controller easy. The controller base class you choose is not crucial because most request processing goes into executing the `ActionResult`, which is the type that each *action* returns.

An action is a method that handles a particular request. This method can take no parameters or many, but by the time the action method finishes executing, there ought to be one or many objects ready to be sent to the view, and the name of the view should be selected if the view does not follow the convention of having the same name as the action. Beyond that, the developer is in complete control regarding how to implement a controller and its actions. In chapter 1 we covered using the `IController` interface directly for controllers that need only one action. This chapter will explore controllers that use many actions and inherit from the `System.Web.Mvc.Controller` base class. The meat of the controller is the action.

3.1 The controller action

Any class that inherits from `System.Web.Mvc.Controller` can use action methods to serve web requests. An action method normally returns an `ActionResult` and can take zero or many arguments. Parameters are resolved into the action method by a combination of form values, the `Route` definition, and the querystring, in that order. The requirements for a method to be web-callable as an action method are well documented on http://www.asp.net/mvc. The method

- Must be public
- Cannot be a static method
- Cannot be an extension method
- Cannot be a constructor, getter, or setter
- Cannot have open generic types
- Is not a method of the `Controller` base class
- Is not a method of the `ControllerBase` base class
- Cannot contain `ref` or `out` parameters

An action has a clear purpose and a single responsibility. That responsibility is to accept arguments, if any, coordinate with relevant dependencies, push objects into `ViewData`, and choose a view to render.

Action methods should not be performing functions such as data access or file I/O. Action methods exist to perform presentation coordination for a screen/page. Any supporting logic should be factored into appropriate classes. If you see an action method that does not fit on one screen without scrolling, consider how many responsibilities it has. You will end up with more maintainable software by factoring much of the logic into supporting service classes or presentation model classes. If you think of the whole application's layering, you should be able to take away all the screens, that is, views and controllers, without losing system functionality. In other words, if users upload a batch file to your web app, and the processing of the contents of the batch file is inside an action method, that logic is in the wrong place. As soon as that controller goes away, the application cannot process batch files. This example is a good rule of thumb for determining if the action is trying to do too much.

NOTE In this book, we focus on complex, long-lasting web applications. In line with that, we do not make compromises to optimize the speed of writing the application. Software engineering is full of trade-offs, and software construction techniques are no exception. If you need a small web application, you can probably get away with putting all the logic in the controller action, but realize that you're trading off long-term maintainability for short-term coding speed. If the application will have a long life, this is a bad trade-off. The examples in this book are factored for long life and easy maintenance, so you will notice interfaces employed to separate concerns.

In listing 3.1 we see a simple controller with a single action. This is a trivial example, and we will tackle more complex scenarios later. We begin by ensuring that the action method is `public` and returns `ActionResult`. If the method is not `public`, it will not be called. At this point, we can push some objects into `ViewData` and call the `View()` method with the name of the view that should render. That is the meat and potatoes of what it means to be an action method.

Listing 3.1 The `SimpleController` decides on `ViewData` and renders a view

```
using System.Web.Mvc;

namespace MvcInAction.Controllers
{
    public class SimpleController : Controller
    {
        public ActionResult Hello()
        {
            ViewData.Add("greeting", "Hello Readers!");
            return View();
        }
    }
}
```

The most important point to remember is that the controller action adds objects to the `ViewData` and calls for the rendering of a view by name. A popular convention is to keep the view name the same as the action name for simplicity. If the view name matches the action name, there is no need to specify it when calling `return View()`. `ViewData` is a `IDictionary<string, object>` at its core. The type is `System.Web.Mvc.ViewDataDictionary`, but we only need to worry about the interface. MvcContrib contains a group of extension methods called `ViewDataExtensions` which make `ViewData` easier to work with within a controller as well as a view. The extensions don't take away any functionality; they add functionality. You will see use of these extensions sprinkled throughout this book.

Action methods can return any object type, including void. If the type derives from `System.Web.Mvc.ActionResult`, that result will be executed. If any other type is returned, the framework will call the `ToString()` method on it and return a `ContentResult`. The following listed types are the available derivations of `ActionResult`:

1. *ContentResult*—Represents a text result
2. *EmptyResult*—Represents no result
3. *FileResult*—Represents a downloadable file (abstract class)
4. *FileContentResult*—Represents a downloadable file (with the binary content)
5. *FilePathResult*—Represents a downloadable file (with a path)
6. *FileStreamResult*—Represents a downloadable file (with a file stream)
7. *HttpUnauthorizedResult*—Represents the result of an unauthorized HTTP request
8. *JavaScriptResult*—Represents a JavaScript script
9. *JsonResult*—Represents a JavaScript Object Notation (JSON) result that can be used in an AJAX application
10. *RedirectResult*—Represents a redirection to a new URL
11. *RedirectToRouteResult*—Represents a result that performs a redirection given a route values dictionary
12. *PartialViewResult*—Base class used to send a partial view to the response
13. *ViewResult*—Represents HTML and markup
14. *ViewResultBase*—Base class used to supply the model to the view and then render the view to the response
15. *XmlResult*—Action result that serializes the specified object into XML and outputs it to the response stream (provided by the MvcContrib library)

Each of these types has a corresponding helper method on the `Controller` base class that can be used to easily construct the type and return it. Although most actions will return a `ViewResult` or some other type of result, a controller action is not required to have a view associated with it.

3.2 Simple controllers do not need a view

Most of the examples in this book use a view to render objects to an HTML screen; however, if you want the behavior but not the display, you would do well to use a controller without a view. Suppose we wanted to support some alternative URLs for CodeCampServer. In an installation of CodeCampServer, we can have many conferences listed. Some conferences are past and some future. Suppose we wanted an easy URL to pull up the next Code Camp. Let's say this URL would be: `http://www.codecampserver.org/nextconference`. We would start by defining a special route because this is a special case. This route uses the default API included in the ASP.NET MVC Framework. Both chapter 2 and the CodeCampServer source code make use of the route API wrappers found in MvcContrib. You can configure routes in many ways, so we are showing a variety of techniques, which we explain in depth in chapter 5. The route for our simple controller as defined in listing 3.2 is a simple route for the purposes of this example. It uses the included API from ASP.NET 3.5 Service Pack 1, which includes the routing capability. As covered in chapter 5, when you progress to more control over routes, you will need a richer API, such as the one used by CodeCampServer, or you can create your own wrapper.

Listing 3.2 Adding a route to redirect with a special controller action

```
RouteTable.Routes.MapRoute("next", "nextconference",
                           new
                               {
                                   controller = "redirect",      ❶
                                   action = "nextconference"     ❷
                               });
```

We want the ASP.NET MVC Framework to route this URL to a controller named `RedirectController` ❶. The default action will be `NextConference` ❷. Listing 3.3 shows the full source for the `RedirectController`. Note that there is no need to call the `View()` method because we do not have or need a view.

Listing 3.3 A view is unnecessary when performing a redirect.

```
using System.Web.Mvc;
using CodeCampServer.Core.Domain;
using CodeCampServer.Core.Domain.Model;

namespace MvcInAction.Controllers
{
    public class RedirectController : Controller
    {
        private readonly IConferenceRepository _repository;

        public RedirectController(IConferenceRepository
            conferenceRepository)
        {
            _repository = conferenceRepository;
        }

        public RedirectToRouteResult NextConference()      <-- Return derived type of ActionResult
        {
            Conference conference = _repository.GetNextConference();

            return RedirectToAction("index", "conference",
                                    new {conferenceKey = conference.Key});
        }
    }
}
```

As we walk through the `NextConference` action, we notice that we are coordinating dependencies to get the job done. Figure 3.1 illustrates the controller and its dependencies.

The `IConferenceRepository` instance knows how to retrieve the appropriate conference that is next on the schedule. After we have the conference that is next, we can redirect to the URL that will route to the `ConferenceController`, which knows how to work with a single conference. We will discuss the design of this controller shortly, but notice that the dependency, `IConferenceRepository`, is passed in through the constructor. What is *dependency injection* (DI)? It's a fancy term for passing objects into a constructor or public setters. The default controller factory supplied with the ASP.NET MVC Framework does not know how to resolve constructor dependencies of controllers, but there are several `IControllerFactory` implementations available in the MvcContrib

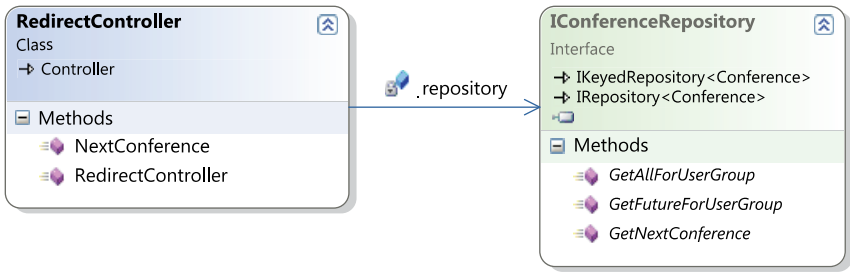

Figure 3.1 The `RedirectController` depends on one interface. It declares the dependency in its constructor which makes dependency injection easier.

open source project. For this example, we use the `StructureMapControllerFactory` source copied from MvcContrib. We register the controller factory with a single line of code as shown in listing 3.4.

Listing 3.4 Using an MvcContrib controller factory enables IoC support.

```
ControllerBuilder.Current.SetControllerFactory(
  new StructureMapControllerFactory());
```

Initialize ControllerFactory in Global.asax

```
public class StructureMapControllerFactory : DefaultControllerFactory
{
    protected override IController GetControllerInstance(
        Type controllerType)
    {
        return (IController) ObjectFactory.GetInstance(controllerType);
    }
}
```

The inversion of control principle and DI

Normally when code executes other code, there is a linear flow of creation and execution. For instance, if I have a class that depends on another class, I will create that class with the `"new"` operator, then execute the class by calling a method. If I used inversion of control (IoC), I would still call methods on the class, but I would require an instance of the class passed into my constructor. In this manner, I yield control of locating or creating my dependency to the calling code. DI is the act of injecting a dependency into a class that depends on it. Often used interchangeably, IoC and DI yield loosely coupled code and are often used with interfaces. With interfaces, classes declare dependencies as interfaces in the constructor arguments. Calling code then locates appropriate classes and passes them in when constructing the class.

IoC containers come into play to assist with managing this technique when used through an application. There are plenty of IoC containers to choose from, but the favorites at this time seem to be StructureMap and Castle Windsor found at http://structuremap.sourceforge.net and http://www.castleproject.org/container/index.html respectively.

We could just as easily have used the `WindsorControllerFactory`, `SpringControllerFactory`, or `UnityControllerFactory`, all supplied with MvcContrib. Using an IoC container to construct the controller allows us to externalize dependency configuration. You're probably wondering how to test this controller that does not have a view because we are doing a redirect directly in the controller. Regardless of the IoC container used, testing is still the same.

3.3 Testing controllers

The focus of this section is testing controllers. Of the different types of automated testing, we are concerned with only one type at this point: unit testing. Unit tests run fast because they do not call out of process. In a unit test, dependencies are simulated so the only production code running is the controller code. For this to be possible the controllers have to be well designed. A well-designed controller

- Is loosely coupled with its dependencies
- Uses dependencies but is not in charge of locating or creating those dependencies
- Has clear responsibilities and only handles logic relevant to serving a web request

A well-designed controller does not do file I/O, database access, web service calls, and thread management. The controller may very well call a dependency that performs these functions, but the controller itself should be responsible only for interaction with the dependency, not for performing the fine-grained work. This is very important to testing because good design and testing go hand in hand. It's very difficult to test poorly designed code.

NOTE Writing automated tests for all code in a code base is a best practice. It provides great feedback when the test suite is run multiple times per day. If you're not doing it now, you should start immediately. Several popular, high quality frameworks for automated testing available include NUnit and MbUnit. At the time of writing, NBehave, MSTest, and xUnit are also available, but they are not as widely adopted as NUnit or MbUnit. All are free (with the exception of MSTest, which requires the purchase of Visual Studio) and they simplify testing code.

In this section, we will walk through testing our viewless `RedirectController`.

3.3.1 Testing the RedirectController

The `RedirectController` must find the next conference and issue a redirect to another URL so that a single conference can be displayed on the screen. This controller must find the conference and ask for a redirect to the action that can take it from there. The ASP.NET MVC Framework provides a redirect mechanism that makes it unnecessary to use `Response.Redirect()`, which is more difficult to test. The action method in question returns an object that has public properties, which can be evaluated in a test. The action result contains an `Execute` method that performs the redirect, but the controller action merely returns an object. This is important for the easy

testing of controller actions. In listing 3.5, we set up a unit test for this code along with fake implementations of the dependencies on which the `RedirectController` relies.

Listing 3.5 `RedirectControllerTester`: ensuring we redirect to the correct URL

```
using System;
using System.Web.Mvc;
using CodeCampServer.Core.Domain;
using CodeCampServer.Core.Domain.Model;
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;

namespace MvcInAction.Controllers.UnitTests
{
    [TestFixture]
    public class RedirectControllerTester
    {
        [Test]
        public void ShouldRedirectToTheNextConference()
        {
            var conferenceToFind =
                new Conference{Key = "thekey", Name = "name"};
            var repository = new
                ConferenceRepositoryStub(conferenceToFind);

            var controller = new RedirectController(repository);

            RedirectToRouteResult result = controller.NextConference();

            Assert.That(result.RouteValues["controller"],
                Is.EqualTo("conference"));
            Assert.That(result.RouteValues["action"],
                Is.EqualTo("index"));
            Assert.That(result.RouteValues["conferenceKey"],
                Is.EqualTo("thekey"));
        }

        private class ConferenceRepositoryStub : IConferenceRepository
        {
            private readonly Conference _conference;

            public ConferenceRepositoryStub(Conference conference)
            {
                _conference = conference;
            }

            public Conference GetNextConference()
            {
                return _conference;
            }

            public Conference[] GetAllForUserGroup(UserGroup usergroup)
            {
                throw new NotImplementedException();
            }

            public Conference[] GetFutureForUserGroup(UserGroup usergroup)
            {
```

Exercise class under test

Create using simulated dependencies

Assert correct results

❶

```
                throw new NotImplementedException();
            }

            public Conference GetById(Guid id)
            {
                throw new NotImplementedException();
            }

            public void Save(Conference entity)
            {
                throw new NotImplementedException();
            }

            public Conference[] GetAll()
            {
                throw new NotImplementedException();
            }

            public void Delete(Conference entity)
            {
                throw new NotImplementedException();
            }

            public Conference GetByKey(string key)
            {
                throw new NotImplementedException();
            }
        }
    }
}
```

Notice that most of the code listing is test double code, and not the `RedirectController` test itself. We have to stub out an `IConferenceRepository` implementation ❶ because calling that interface inside the controller action provides the next conference. How it performs that search is beyond the scope of this chapter and is irrelevant to the controller. When glancing at this test, you probably think that it's too complex for a single unit test. We will see shortly how to reduce the amount of code in the unit test fixture. Reducing code starts with making dependencies explicit.

3.3.2 Making dependencies explicit

There are only three real lines of code in the `RedirectController`. The controllers should all be thin, and this is a good example. The logic for finding the correct `Conference` object is a data access issue, and does not belong in the controller, so it's factored into a repository object. Only logic related to presenting information to the user belongs in the controller. In this case, the user experiences a redirect. This controller demonstrates proper separation of concerns, and it's easily unit tested because it's only involved with a single responsibility. We are able to simulate dependencies using test doubles.

In Figure 3.2, you see the unit test passing because we were able to properly simulate this controller's dependencies and verify that given the dependencies, the controller will do its job correctly.

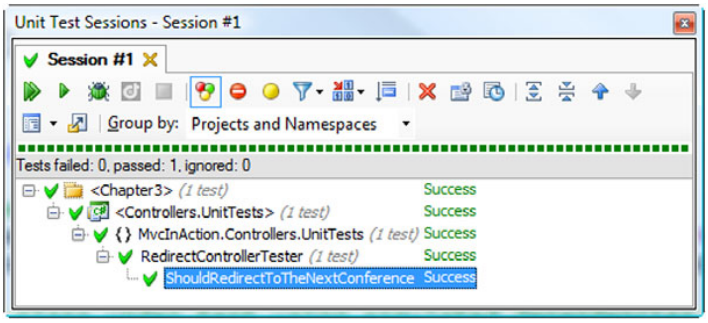

Figure 3.2
Redirect test passing

3.3.3 Using test doubles, such as stubs and mocks

As far as the controller is concerned, its caller is passing in an implementation of the necessary interface. This interface is a dependency, and the controller makes use of it in an action method. How the dependency is passed in or what class implements the interface is irrelevant. At runtime, a production class will be passed into the controller, but at the time of unit testing, we use stand-in objects, or test doubles, to simulate the behavior of the dependencies. There are different types of simulated objects, and some of the definitions overlap. There are entire books written about testing and how to separate code for testing using fakes, stubs, and mocks, and if you're interested in exploring the subject further, we highly recommend reading Michael Feather's *Working Effectively with Legacy Code*. In short, the terms *fake* and *test double* are generic terms for a nonproduction implementation of an interface or derived class that stands in for the real thing. Stubs are classes that return hard-code information solely for the purpose of being called. The `ConferenceRepositoryStub` shown in listing 3.5 is an example of a stub. A mock is a recorder that remembers being called so that we can assert the behavior later on. It remembers arguments passed in and other details depending on what capability has been programmed into it.

One downside to using hand-coded stubs and mocks is that you have many lines of code just to satisfy an interface implementation that may have six methods. This is not the only option, however. A favorite library for automating the creation of mocks and stubs is *Rhino Mocks*, originally written by Oren Eini. Rhino Mocks drastically reduces the number of lines of code in a unit test fixture by streamlining the creating of test doubles. If code is designed so that all dependencies are injected into the constructor, as shown in listing 3.6, unit testing becomes easy and soon becomes a repetitive pattern of faking dependencies and writing assertions. Over time, if you employ this technique, you will see a marked improvement in the quality of your code.

Listing 3.6 Controllers can define dependencies in the constructor.

```
public RedirectController(IConferenceRepository conferenceRepository)
{
    _repository = conferenceRepository;
}
```

Remember how many lines of code we wrote for a stubbed implementation of `IConferenceRepository`? Now, examine listing 3.7 and notice how short this code listing is in comparison. Rhino Mocks supports setting up dynamic stubs as well as dynamic mocks. The lines with `Stub(...)` are used so that a stubbing method or property always returns a given object. By using the Rhino Mocks library, we can provide dependency simulations quickly for easy unit testing.

Listing 3.7 Using Rhino Mocks to streamline code necessary for fakes

```
using System.Web.Mvc;
using CodeCampServer.Core.Domain;
using CodeCampServer.Core.Domain.Model;
using NUnit.Framework;
using NUnit.Framework.SyntaxHelpers;
using Rhino.Mocks;

namespace MvcInAction.Controllers.UnitTests
{
    [TestFixture]
    public class RedirectControllerTesterWithRhino
    {
        [Test]
        public void ShouldRedirectToTheNextConference()
        {
            var conferenceToFind = new Conference
            {
               Key = "thekey", Name = "name"
            };

            var repository =
               MockRepository.GenerateStub<IConferenceRepository>();

            repository.Stub(r =>
                r.GetNextConference()).Return(conferenceToFind);

            var controller = new RedirectController(repository);
            RedirectToRouteResult result = controller.NextConference();

            Assert.That(result.RouteValues["controller"],
                  Is.EqualTo("conference"));
            Assert.That(result.RouteValues["action"],
                  Is.EqualTo("index"));
            Assert.That(result.RouteValues["conferenceKey"],
                Is.EqualTo("thekey"));
        }
    }
}
```

Return a specific conference

Stub using Rhino Mocks

Assert correct results

A dynamic mocking library like Rhino Mocks is not appropriate in every unit testing scenario. The usage in listing 3.7 is the bread-and-butter scenario that reduces the amount of setup code inside unit tests. More complex needs can quickly stress the Rhino Mocks API and become hard to read. Although Rhino Mocks supports almost everything you could want to do, the readability of the tests is important to maintain. When you need to assert method parameters of dependencies or do something special,

do not be afraid to push Rhino Mocks to the side and leverage a concrete mock or stub to keep the test readable.

3.3.4 Elements of a good controller unit test

If you're just getting started with unit testing you might run into common pitfalls and stub your toe. Again, this is not meant to be an entire course on testing. There are already comprehensive books on that, such as *The Art of Unit Testing* by Roy Osherove. This book specifically addresses writing unit tests for controller classes. We focus heavily on testing controller classes because test-driving the controllers ensures they are well designed. It's nearly impossible to test-drive code that ends up with a bad design.

NOTE Poorly designed code tends to be untestable, so observable untestability is a very objective gauge of poorly designed code. A good controller unit test runs fast. We are talking 2000 unit tests all running within 10 seconds. How is that possible? .NET code runs fast, and if you're running unit tests, you're waiting only for the processor and RAM. Unit tests run code only within the AppDomain, so we do not have to deal with crossing AppDomain or Process boundaries. You can quickly sabotage this fast test performance if you break a fundamental rule of unit testing, and that is allowing out-of-process calls. Out-of-process calls are orders of magnitude slower than in-process calls, and your test performance will suffer. Ensure that you're faking out all controller dependencies, and your test will continue to run fast.

You also want your unit tests to be self-sufficient and isolated. You might see repeated code and think you need to refactor your unit tests. Resist this temptation and create only test helpers for the cross-cutting concerns. The DRY principle (Don't Repeat Yourself) does not apply to test code as much as to production code. Rather, keeping test cases isolated and self-contained reduces the change burden when the production code needs to change. It's also more readable if you can scan a unit test and see the context all in one method.

The tests should also be repeatable. That means no shared global variables for the test result state, and no shared state between tests in general. Keep a unit test isolated in every way, and it will be repeatable, order-independent, and stable.

Pay attention to pain. If your tests become painful to maintain, there's something wrong. The tests should enable development, not slow it down. If you start to think that you could move faster without writing the tests, look for technique errors or bad design in the production code. Get a peer to review the code. Correctly managed design and tests enable sustained speed of development whereas poor testing techniques cause development to slow down to a point where testing is abandoned. At that point, it's back to painstaking, time-intensive manual testing. With that critical practice safely stowed in our tool belt, let's explore actions in more detail.

3.4 Simple actions and views

To demonstrate how a controller can do interesting things without a view, our previous controller, `RedirectController`, did not use a view. A controller can work independently from a view. More interesting (and common), though, is a controller that pushes objects into `ViewData` and then returns a `ViewResult` with a named view to render. The only coupling between the controller and a view is the view name declared in the call to `return View()`.

Views can be simple or complex, and complex views can require many objects to be passed in as view data. Simple views, likewise, often require only a single object or no object at all. In listing 3.8, we see an action in the CodeCampServer project that is responsible for fulfilling a request to display the registration form. This screen needs to display header information about the `Conference` and then a list of text boxes to collect an attendee registration. The URL that would be routed to this action would be http://CodeCampServer.org/AustinCodeCamp09/Attendee/New. Because the routes in CodeCampServer specify the first segment named `conferenceKey`, the ASP.NET MVC Framework will extract this portion of the URL and add it to `RouteData` to make it available to model binders. The argument will be used to resolve the action parameter, `conference`, and the action code in listing 3.8 will be able to use it when preparing to render the view.

Listing 3.8 Action passes information so the view can render a data entry screen

```
public ViewResult New(Conference conference)
{
    var model = new AttendeeForm {ConferenceID = conference.Id};
    return View("Edit", model);
}
```

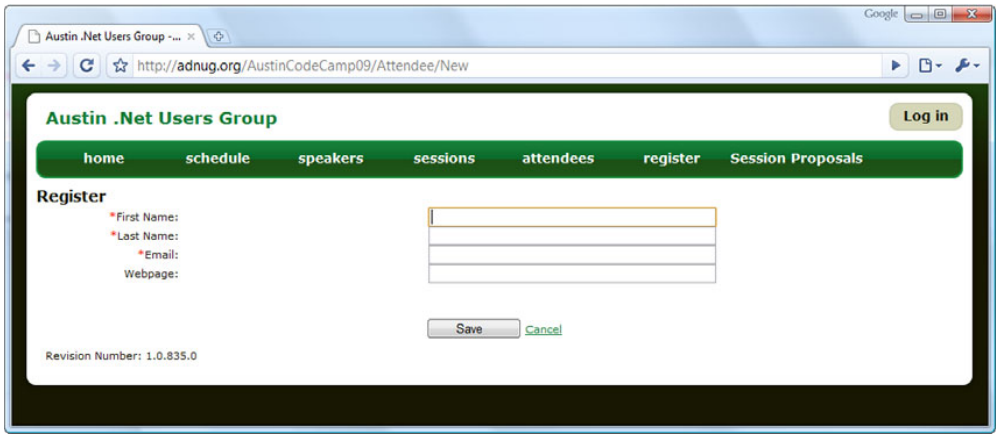

Figure 3.3 The `Edit` view can be used for a new attendee or existing attendee.

When views containing forms, like the one in figure 3.3, are involved, the potential for interactive web applications makes us think about how to pass arguments from the browser back in to a controller. One of the fundamental data transactions of the web is the *form post*.

3.5 Working with form values

An HTML page that has a `<form method="POST"/>` tag will generate a `POST` back to the same URL, unless an alternate action is specified. When a page is posted to a URL, all form fields are translated into a form values collection. For instance, if the page contains `<input type="text" name="FirstName"/>`, user entries in that text box will be entered into the form collection and available to the server code that handles the request. With the ASP.NET MVC Framework, form values are routed automatically into the controller action. The action parameters are matched to values in the form collection by name. In figure 3.2, we see a page containing a form with text boxes that collect attendee registration information for CodeCampServer. Each of these text boxes has a unique name used to match action parameters.

Looking at listing 3.9, we see one option for receiving the form post. Notice the names of the parameters on this method. The parameter `conferenceKey` is still there and will be matched to the appropriate part of the route. Next are `firstName`, `lastName`, and so on. The ASP.NET MVC Framework will search to match parameters to this method. The matching mechanism matches by name. Because `firstName` exists in the form collection, the value will be passed into this method along with the other parameters. The end result is that all the attendee registration information is translated into action parameters and passed in. As developers, we don't have to do a thing to extract the appropriate information from the form collection on the request. We must only ensure the names of the action parameters match up with the names posted to the URL.

Listing 3.9 Form values are automatically passed to the action

```
public ActionResult Save(string conferenceKey, string firstName,
                         string lastName, string email,
                         string webpage)
{
    // Method body omitted
}
```

Already, you're probably wondering about conflicts. What happens if a form or querystring value is named `conferenceKey`? Which value will be used? The ASP.NET MVC Framework matches action parameters in the following order:

1. Form values
2. Route arguments
3. Querystring parameters

Parameters on an action method are always matched by name in the order given. If there is a duplicate, the first one found wins, and subsequent duplicates are ignored. If there is a valid reason to have a form value and route argument named the same, your code will have to handle extracting the form value using `Request.Form["key"]` or the action method can accept `FormCollection` as a parameter. In our sample in listing 3.9, all the parameters are of type `string`. Because web requests are processed in string form, we will need some mechanism to parse them into more complex types. We will cover this in depth in the section on *model binders*.

NOTE Because the form values are processed first, there are certain form parameter names you should not use (they will likely cause unexpected behavior if you do) such as `action` and `controller`.

Already we begin to see the value proposition of the ASP.NET MVC Framework, and we understand why folks also enjoy Ruby on Rails and MonoRail. These frameworks abstract away repetitive plumbing code like mapping query strings and form parameters to variables and leave only the interesting code to be written. We have seen how form and route values are mapped into action methods. Next we will examine doing the same with querystring values.

3.6 Processing querystring parameters

Querystrings are mapped into a controller action in a fashion similar to form values—by name matching. If the ASP.NET MVC Framework does not find a matching value in the form collection, it will then search the querystring for a parameter that matches. Upon finding the matching value, it will pass it through the appropriate action parameter. In this way, we can alter our URLs to provide a dynamic environment within a single controller action.

In listing 3.10, this controller is going to pass along the greeting to the view as the `ViewData.Model`. A URL similar to `~/hello?greeting=Hello+Jeffrey` would cause the page to output "Hello Jeffrey" on the screen. If the querystring value is missing, the parameter will be null. Since `System.String` is a nullable type, we have no problem here.

Listing 3.10 Querystring parameters are passed to the action just like form values.

```
using System.Web.Mvc;

namespace MvcInAction.Controllers
{
    public class HelloController : Controller
    {
        public ActionResult Index(string greeting)
        {
            ViewData.Model = greeting;
            return View();
        }
    }
}
```

NOTE The ASP.NET MVC Framework matches querystring values based completely on name. The order is unimportant. As an exercise, pull down the code for this chapter and change the order of the action parameters. The behavior of the application will be unaffected. Similarly, change the order of the querystring parameters. No change. The name of the parameter is what matters.

So far, all of our action parameters have been strings. In practice, we need to be able to use a diverse set of types in our application. We will tackle that next. Two main concerns exist when binding from the form, route, and querystring. The first is to match the string value based on key. The next is to parse it into the correct object. With string action parameters, there is no parsing because `string` is the native type. Now let's investigate how to parse more complex types.

3.7 Binding more complex objects in action parameters

As soon as we get away from Hello World applications, we are faced with complex types, and we need to be able to accept them in action parameter lists. In listing 3.9, we saw an action method signature that accepted a form posting as a series of string parameters. A better binding method uses a form object, such as the `AttendeeForm` shown in listing 3.11. This class is from CodeCampServer, which leverages a value object supertype from the Tarantino project, `ValueObject<T>`. In your code, this may be irrelevant.

Listing 3.11 A dedicated form object can encapsulate the data of a form post.

```
public class AttendeeForm : ValueObject<AttendeeForm>
{
    public virtual Guid ConferenceID { get; set; }
    public virtual string FirstName { get; set; }
    public virtual string LastName { get; set; }
    public virtual string EmailAddress { get; set; }
    public virtual string Webpage { get; set; }
}

public ActionResult Save(AttendeeForm form){}
```

Resulting action method signature

The mechanism in charge of matching action parameters and pulling them in from the request is the `IModelBinder` interface. Out of the box, the class that matches .NET Framework types, simple or nested, is the `System.Web.Mvc.DefaultModelBinder` class. The `DefaultModelBinder` class can bind any type with a .NET `TypeConverter`, such as `Int32`, `DateTime`, `Guid`, etc. It can also match

- Arrays
- Collections
- Dictionaries
- Complex objects containing any of these types

The built-in binding capabilities are powerful, and they work for all the primitive types, both on their own and when nested within complex types. We still need the capability

to bind our own custom types, such as the `Conference` type in CodeCampServer. For this, we will need to implement our own `IModelBinder` instance. In listing 3.12, we see a controller action that requires a custom type as well as a custom model binder. The listing handles an HTTP request that has a parameter named `conference`, which contains the conference key.

Listing 3.12 Using a custom model binder to take control over binding custom types

```
public class BindConferenceController : Controller
{
    public object Index(Conference conference)
    {
        return conference.Name;
    }
}

using System.Web.Mvc;
using CodeCampServer.Core.Domain;
using CodeCampServer.Core.Domain.Model;

namespace MvcInAction
{
    public class ConferenceModelBinder : DefaultModelBinder
    {
        private readonly IConferenceRepository _repository;

        public ConferenceModelBinder(IConferenceRepository repository)
        {
            _repository = repository;
        }

        public override object BindModel(
            ControllerContext controllerContext,
            ModelBindingContext bindingContext)
        {
            ValueProviderResult providerResult =
                bindingContext.ValueProvider[bindingContext.ModelName];

            Conference conference =
                _repository.GetByKey(providerResult.AttemptedValue);

            return conference;
        }
    }
}
```

Binder has dependencies

Inherit from Default ModelBinder

Find Conference by key

Find value matching name

One more step is needed to hook in this custom model binder. When the application starts up, we need to register our model binder with the ASP.NET MVC Framework. We can do that easily in the Global.asax.cs file. Listing 3.13 shows the one line of code necessary to register our custom model binder. Here you see that we are passing in a stub for the `IConferenceRepository`. In your application, you would probably resolve the model binder with an IoC container or a factory.

Listing 3.13 Register our custom model binder when the web application starts.

```
ModelBinders.Binders.Add(typeof (Conference),
                         new ConferenceModelBinder(
                         new ConferenceRepositoryStub()));
```

It's a good idea to use meaningful types in controller actions. If the action parameters are all `strings`, `ints`, and `Guids`, the action methods will be cluttered with lookup code while the controller struggles to convert a string to a better object. By leveraging the model binder mechanism, we can externalize this lookup and mapping code so that the controller actions can concentrate on making "what" decisions about how the screen will behave. This results in smaller action methods and more maintainable controllers. When you browse through CodeCampServer, look at the types passed into action methods. Rarely are they .NET primitive types.

The resolution of action parameters coupled with model binders makes it easy to craft an action method that takes in information from a web request. We can use the form values, route values, and the querystring to make the action behavior more dynamic. Again, notice how effortless it is to consume this request data. We do not have to write any repetitive code to pull these values in. Rather, the ASP.NET MVC Framework finds the correct parameter and maps it to the action parameter. Our custom model binders take it from there and convert the values to our custom types where necessary. Now that we have objects coming into our action, we will examine how we push objects out to the view.

3.8 Options for passing ViewData

The `System.Web.Mvc.Controller` base class has a `ViewData` property, which is essentially a dictionary. You can use it as is or leverage extension methods in MvcContrib for a richer API on top of the `IDictionary<string, object>` type. `ViewData` has a `Model` property that is a first-class citizen in the view. The primary object passed to the view should go in this property. When more objects are necessary, add them to the dictionary and retrieve them in the view by key name.

We have several options for passing view data from a controller action to a view. After you start creating your own types for view data, the options increase well beyond those presented here. The first option, and the default mechanism you might use at first, is to use the built-in `View()` method parameter. Listing 3.14 demonstrates this. By passing in the object directly to the `View()` method, the framework will automatically assign it to `ViewData.Model`.

Listing 3.14 Callling the `View()` method on the `Controller` base class

```
public ActionResult ViewDataModel(Conference conference)
{
    return View(conference);
}
```

The default mechanism for adding additional objects to a dictionary is to assign each object a key. `ViewData` is no different; however, with MvcContrib, we have an option available that allows us to forgo string keys for access to all objects in `ViewData`. Using `ViewDataExtensions`, we can add an object into `ViewData` without giving it a key. These extensions will implicitly use the type of the variable as the key in the dictionary, and the following code can be used in the view to retrieve the `Conference` added in listing 3.15: `ViewData.Get<Conference>();`. No casting, no dictionary keys. Now you have strong typing on many objects. The only constraint is that you can add only one `Conference` to `ViewData`. If you need to add multiple objects of the same type, you can fall back and assign a unique key to each one.

Listing 3.15 Passing a single object directly to the `View` method

```
public ActionResult MultipleObjectsInViewData(Conference conference, string
     someOtherVariable)
{
    ViewData.Add(new Conference());
    ViewData.Model = someOtherVariable;
    return View();
}
```

Using MvcContrib ViewDataExtensions

For some scenarios, a single object may be sufficient for the view to render. In others, you will need several objects. In our opinion, because `IDictionary` is very flexible, it's appropriate for most uses. The extension methods from MvcContrib enhance the experience even more, and together, we recommend their use in most scenarios. Set the primary object to `ViewData.Model` and put the others in the dictionary. While binding objects into action methods and passing objects to `ViewData`, we often need to insert code in unique places and even share this code among controllers. Filters provide a way to do this.

3.9 Filters

The function of a filter in ASP.NET MVC is similar to its function in the real world. Using filters, we can filter out requests or modify the data that gets through. The notion of a filter applies when using the `System.Web.Mvc.Controller` base class, which, in our experience, is most of the time.

Four interfaces combine to provide filtering support as a controller is executing:

- *`IActionFilter`*—Before and after hooks when an action is executing
- *`IResultFilter`*—Before and after hooks when an action result is executing
- *`IAuthorizationFilter`*—Hooks when ASP.NET is authorizing the current user
- *`IExceptionFilter`*—Hooks when an exception occurs during the execution of a controller

The `Controller` base class implements all these interfaces, so to hook into any of these extensibility points, all you have to do is override the appropriate method in your controller. Overriding controller lifecycle methods is not the engaging part of filters, however. The interesting part is how with filter attributes, you can reuse filters

on many controllers and even pick and choose which filters should apply to which actions. These filter attributes can be applied on the action method or on the controller class definition (which will cause them to apply to all actions in the controller). Filter attributes supplied by the ASP.NET MVC Framework include

- `System.Web.Mvc.ActionFilterAttribute`
- `System.Web.Mvc.OutputCacheAttribute`
- `System.Web.Mvc.HandleErrorAttribute`
- `System.Web.Mvc.AuthorizeAttribute`
- `System.Web.Mvc.ValidateAntiForgeryTokenAttribute`
- `System.Web.Mvc.ValidateInputAttribute`

Implementing one of the filter interfaces is the easiest way to intercept the execution of the controller. It's not the only way to interrupt, or prevent, the execution of a web request. The controller also has a notion of an *action method selector*. The action method selector is in charge of selecting which action method to execute. By decorating action methods with the following attributes, you can alter the default mechanism for action method selection:

- *`System.Web.Mvc.AcceptVerbsAttribute`*—Limits action selection to requests of the specified HTTP verb type
- *`System.Web.Mvc.NonActionAttribute`*—Prevents action method from being selected

To affect the selection of an action method for execution, you can create your own derivations of `ActionMethodSelectorAttribute`. Probably the most useful selector is the `AcceptVerbsAttribute`, where you can limit an action to the "POST" verb so that "GET" requests do not modify server state. It's most important to understand the order of execution of the controller, when many filters are layered on. In listing 3.16, we see a controller that has many filters. You can view the code of `LoggingActionFilterX` in the code that accompanies the book. This class implements each filter interface listed previously and writes to the response on each hook method. The result is the order of operations shown in figure 3.4.

Listing 3.16 Demonstrating many ways filters can be applied to a controller

```
using System.Web.Mvc;

namespace MvcInAction.Controllers
{
    [LoggingActionFilterA(LogMessage = "controller, index 1", Order = 0)]
    [LoggingActionFilterB(LogMessage = "controller, index 2", Order = 1)]
    public class FilterExampleController : Controller
    {
        [LoggingActionFilterA(LogMessage = "action, index 1", Order = 0)]
        [LoggingActionFilterB(LogMessage = "action, index 2", Order = 1)]
        public ActionResult Index()
        {
            Response.Write("Action body executing<br/>");
```

```
            return Content("Action internals<br/>");
        }
    }
}
```

`LoggingActionFilterA` and `LoggingActionFilterB` implement all four of the filter interfaces and output text to the response stream at each hook point. The `Order` property on the attributes controls the order, and we see that the controller always has the first and last word when hooking filter points.

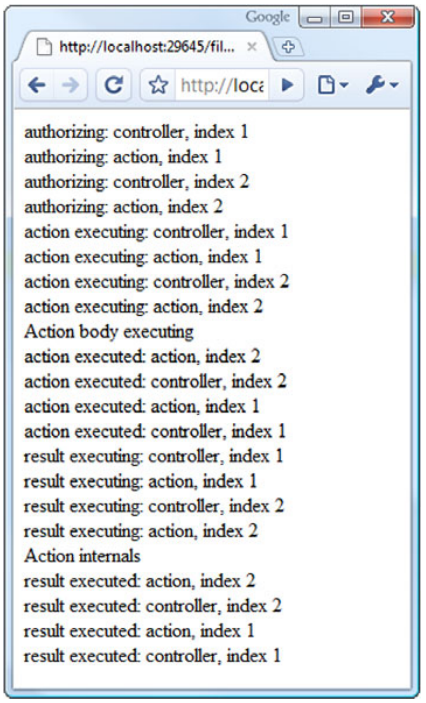

Figure 3.4 The output shows the order of execution of filters when many are applied at both the controller and action level.

3.10 Summary

Controllers are the center of an MVC presentation layer. Controllers handle all the coordination between the Model and the View. Without the controller, we must find another place for this presentation logic. In the ASP.NET MVC Framework, logic is separated into controllers and actions. Actions can accept parameters and can call for the rendering of a view. Actions are not required to have a view, but they commonly do. When using a view, we have several methods for passing view data, and the preferred method is to use an object that best suits your needs. Keep in mind that the default way might not be best for your situation.

Action parameters are matched by name first from the form, then the route, and then querystring. Order is unimportant. If one of the responsibilities of the controller is to perform some task for every action, consider an action filter. In fact, if this task is applicable for many controllers, consider creating a layer supertype that applies this filter. When you're starting a new application based on the ASP.NET MVC Framework, consider creating a layer supertype right from the start. Chances are, the need to make some functionality available to all controllers through inheritance will surface, and it will save you time if you have accumulated many controllers. Of course, YAGNI (You aren't going to need it) applies here, so evaluate your scenario and choose wisely. We often find a layer supertype comes into play at some point. The variable is *when*.

Wielded without caution, controllers have the potential of becoming just as large and convoluted as `Page_Load` methods in Web Forms. Armed with test-driven development and a disciplined approach to separation of concerns, you will ensure the maintainability of your presentation layer. With controller techniques under our belt, we need to fully understand the options for formatting output to the screen using views. This is the topic of chapter 4.

ASP.NET/WEB DEVELOPMENT

ASP.NET MVC IN ACTION

Jeffrey Palermo • Ben Scheirman • Jimmy Bogard
FOREWORD BY Phil Haack

Free eBook
SEE INSERT

ASP.NET MVC implements the Model-View-Controller pattern on the ASP.NET runtime. It works well with open source projects like NHibernate, Castle, StructureMap, AutoMapper, and MvcContrib.

ASP.NET MVC in Action is a guide to pragmatic MVC-based web development. After a thorough overview, it dives into issues of architecture and maintainability. The book assumes basic knowledge of ASP.NET (v. 3.5) and expands your expertise. Some of the topics covered:

- How to effectively perform unit and full-system tests.
- How to implement dependency injection using StructureMap or Windsor.
- How to work with the domain and presentation models.
- How to work with persistence layers like NHibernate.

The book's many examples are in C#.

Jeffrey Palermo is co-creator of MvcContrib. **Jimmy Bogard** and **Ben Scheirman** are consultants and .NET community leaders. All are Microsoft MVPs and members of ASPInsiders.

For online access to the authors and a free ebook for owners of this book, go to manning.com/ASP.NETMVCinAction

"Shows how to put all the features of ASP.NET MVC together to build a great application."
—From the Foreword by Phil Haack
Senior Program Manager
ASP.NET MVC Team, Microsoft

"This book put me in control of ASP.NET MVC."
—Mark Monster
Software Engineer, Rubicon

"Of all the offerings, this one got it right!"
—Andrew Siemer
Principal Architect, OTX Research

"Highly recommended for those switching from Web Forms to MVC."
—Frank Wang, Chief Software Architect, DigitalVelocity LLC

ISBN 13: 978-1-933988-62-7
ISBN 10: 1-933988-62-2
54499
9 781933 988627

MANNING $44.99 / Can $62.99 [INCLUDING eBOOK]