4

Controller

This chapter covers

* Understanding the controller anatomy
* Storyboarding the application.
* Mapping the presentation model
* Using input from the browser
* Passing view metadata
* Testing the controller

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. This chapter will explore controllers that use many actions and inherit from the System.Web.Mvc.Controller base class. Chapter 9 will cover advanced topics regarding controllers. Let's dive in to controller anatomy.

4.1 The Anatomy of a Controller

A controller is simply a class that inherits from System.Web.Mvc.Controller that contains one or more methods that act as actions. An action method is used to serve a single web requests that normally returns an ActionResult and can take zero or many arguments. Parameters are resolved into the action method by the model binders, by using this automatic binding to complex types action methods can focus on the concerns of controlling the application logic rather than spending time: translating input from the browser into input models or mapping domain models into presentation models.

A well written action should have a clear purpose and a single responsibility. That responsibility is to accept input from the browser and coordinate the flow of the application. Along the way, the action should rely on application services to perform tasks such executing business logic, performing data access, or file I/O.

In listing 4.1 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. Now that the makeup of a controller has been defined, we will explain how a controller implements an applications storyboard.

Listing 4.1 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();

}

}

}

Note:

System.Web.Mvc.Controller is only one option to choose as a base class for your controllers. As mentioned earlier, it’s often appropriate to create your own Layer Supertype for all of your controllers. This type can inherit from System.Web.Mvc.Controller, implement IController or derive from any other controller base class.

4.2 Storyboarding the Application

Action methods exist to perform presentation coordination for a screen/page. This coordination is the glue that puts ties together the storyboard of the application. Imagine drawing the flow of application screens on a whiteboard. Each place that a user has the ability to input information through a form or click a button, there are at least two possible outcomes. First, the input data could be correct, satisfying all data type validation and business rules. In this case, the request will be fully processed, and the controller will redirect to the next page. Second, the form submission could have an error. Whether an invalid date was entered, or the input breaks a business rule, the controller needs to render the original page again with the appropriate error messages.

By focusing on implementing storyboard of the application in the controllers action, there are some great side effects. Actions tend to become smaller and focused. By moving business logic out of the action and into supporting services, the actions are less complex and easier to test. A lean action should result in two possible outcomes: happy path ( a successfully processed request) or a alternate path. If an action starts branching to handle multiple alternate paths this is sign that the action method is handling too much and some design should be put into the storyboard of the application.

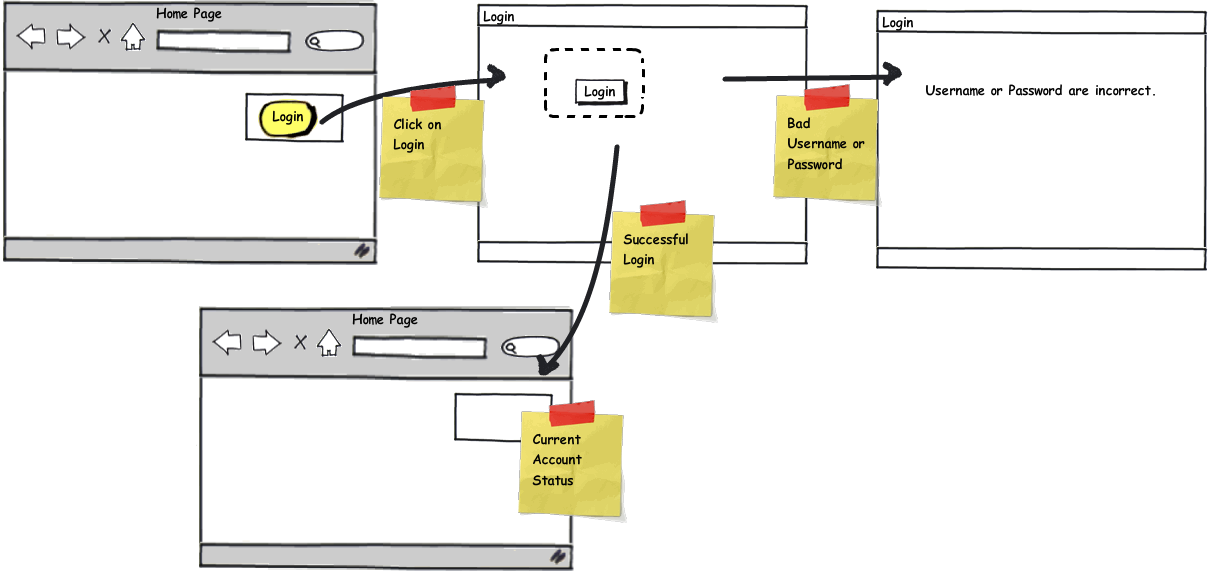


Figure 4.1 Storyboard of an applications user interactions

Figure 4.1 shows a sample storyboard of how a user would login to a web application and see some customized content after successfully logging in. The action that handles the login form post would decide to redirect the user to the homepage or re-render the login form with a message that explains the user needs to enter a correct username and password combination. While this seems like a very obvious path that needs to be developed, it is very easy to overlook the alternate paths when you do not storyboard them. By using a storyboard the design and requirements of your actions just fall off of the storyboard. This technique helps developers and designers communicate how the screens would work before writing a single line of code.

The Happy Path

ASP.NET MVC developers (and developers using other convention-centric frameworks) will often mention the Happy Path. This refers to the notion that following the MVC Framework’s conventions will make the developer’s experience both enjoyable and relatively painless. The MVC Framework does not require you to adhere to any particular convention, but the farther you stray from the Happy Path the greater the effort will be required by the developer. The MvcContrib project enhances the path, and you’ll certainly find ways to enhance it in your system. Staying on the path gains you a great deal in consistency.

4.3 Transforming a model to a view model

A common role of an action is to simply do the work necessary to mold a domain model into a presentation model for a view, JSON, or other output type. This type of action handles a GET request to the web server and in its simplest form returns html to the browser.

Cue Balls for the Code in Text

Listing 4.2 - An Action that prepares a presentation model for a view

public ActionResult Index()

{

IEnumerable<User> users = UserRepository.GetAll(); | #A

UserDisplay[] viewModel = users.Select( | #B

user => new UserDisplay | #B

{ | #B

Username = user.Username, | #B

Name = | #B

user.FirstName + " " + | #B

user.LastName | #B

}).ToArray(); | #B

return View(viewModel); | #C

}

In the code example listing 4.2, this action retrieves a collection of user domain model objects and transforms the objects into a presentation model of type UserDisplay[]. The action relies on a (#A) UserRepository class to handle all the communication to the database and turning the native database objects into the User collection. Next, the action uses some System.Linq syntax (#B) to minimize the noise in performing this type of transformation. The last line of the action (#C) returns the presentation model to a View helper method and returns a ViewResult to the MVC framework. Since a view name was not specified the framework uses a convention and looks for a view that matches the Action name. In this case it would look for a view called Index.

4.3 Accepting Input

An action method receives input from the web browser via its method arguments. The controller uses the model binder feature to convert values from web requests into CLR objects that match the named of parameters of an action method. The internals of how this works is covered in Chapter 14. For now, it is important to understand that a convention is used to match up form values by their name to the parameter name of an action.

Cue balls for code in text

Listing 4.3 A value object bound to an Action from a query string

[HttpGet]

public ActionResult Edit(int Id) |#A

{

User users = UserRepository.GetById(Id); |#B

....

}

The code in listing 4.3 shows a value object being bound from the query string of a url. The url to send an id with the value 4 would be http://localhost/User/Edit/4. The model binder automatically binds this value to the action's parameter(#A). The action can use the value to perform it's work (#B) without having to deal with the concerns of pulling values out of the HttpContext. When the code in an action accesses the Request object to translate data into an object, that is a sign that the action is doing work that it should not be. Actions need to be focused on the storyboard instead of translating input data.

Cue balls for code in text

Listing 4.4 A complex object bound to an Action from a Form Post

public class UserInput |#A

{ |#A

[Required] |#A

public string Username { get; set; } |#A

public string FirstName { get; set; } |#A

public string LastName { get; set; } |#A

} |#A

[HttpPost]

public ActionResult Edit(UserInput input) |#B

{

...

}

Listing 4.4 demonstrates an action method that accepts a complex type as a parameter. ASP.NET MVC will automatically convert the form values into CLR objects by matching on the property names. In this case, the form post data is converted into a UserInput object (#A). The action method can just accept the complex type (#B) as a parameter.

NOTE:

Along with the MVC Framework, Microsoft has wrapped some of the ASP.NET code and provided abstract classes to some of the key APIs such as HttpResponseBase, HttpRequestBase, and most importantly, HttpContextBase. A Google search will reveal how many people have had trouble testing against HttpContext because of its sealed and static members. Providing abstract classes for these key APIs loosens the coupling to them, increasing testability.

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.

4.3.1 Handling the Successful Storyboard path in an Action

Now that you understand how to accept user input, let's move on to implementing the application's storyboard. In the case of accepting user input from a form post, the decision to determine the success or alternate path can be determined by data type validation. When the criteria for success is met, then the action can coordinate the success activities and control the flow to the next screen or action.

Cue balls for listing in text

Listing 4.5 The success path in an Action

[HttpPost]

public ActionResult Edit(UserInput input) |#A

{

if (ModelState.IsValid) |#B

{

UpdateUserFromInput(input); |#C

TempData["message"] = "The user was updated"; |#D

return RedirectToAction("index"); |#E

}

return View(input);

}

private void UpdateUserFromInput(UserInput input) |#F

{ |#F

User user = |#F

UserRepository.GetByUsername(input.Username); |#F

user.FirstName = input.FirstName; |#F

user.LastName = input.LastName; |#F

UserRepository.Save(user); |#F

}

Listing 4.5 shows that the success path is determined by the call to the ModelState.IsValid property (#B). The model binder translates the form post data into the UserInput object (#A) also populates the ModelState object with metadata about the data type validation of the object. When all of the validation passes than the IsValid property is true. In this case, the (#C) UpdateUserFromInput method is called. This method updates the User object from the input model (#F). Once the update occurs a success message is put into TempData (#D). The TempData allows transient data to be passed between two consecutive calls to the webserver. In this case after the redirect to the next action call will have the TempData available to display the message to the user. The last line of code in the success path (#E) returns a RedirectToAction ActionResult. The RedirectToAction("Index") returns a browser redirect to the UserController.Index action. This approach keeps the Action simple and concise.

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 will 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.

4.3.2 Using the Post-Redirect-Get ( PRG ) Pattern

The code in Listing 4.5 demonstrates a pattern call Post-Redirect-Get. You saw this briefly in Chapter 1. This pattern was first published in 2003 by Michael Jouravlev. The pattern is used to prevent some common problems that occur after a user has posted a form to a web server. If a view is rendered directly from a form post than the user may attempt to refresh the browser or bookmark the page. This problem can cause double form submissions or other erroneous behavior. By redirecting after a form Post to a url that uses a Get request, the problem is eliminated. This makes the user experience consistent and deterministic. This pattern should be used when handling form posts.

The screenshots in Figure 4.2 and 4.3 demonstrate a form to collect user input used by an edit action. The success path of the action redirects to the Index page and the page pulls the success message from TempData. The ASP.Net MVC framework provides the components like TempData and the RedirectToAction method to support the PRG pattern. This pattern helps us keep controller actions simple and concise, and it facilitates the application storyboard.

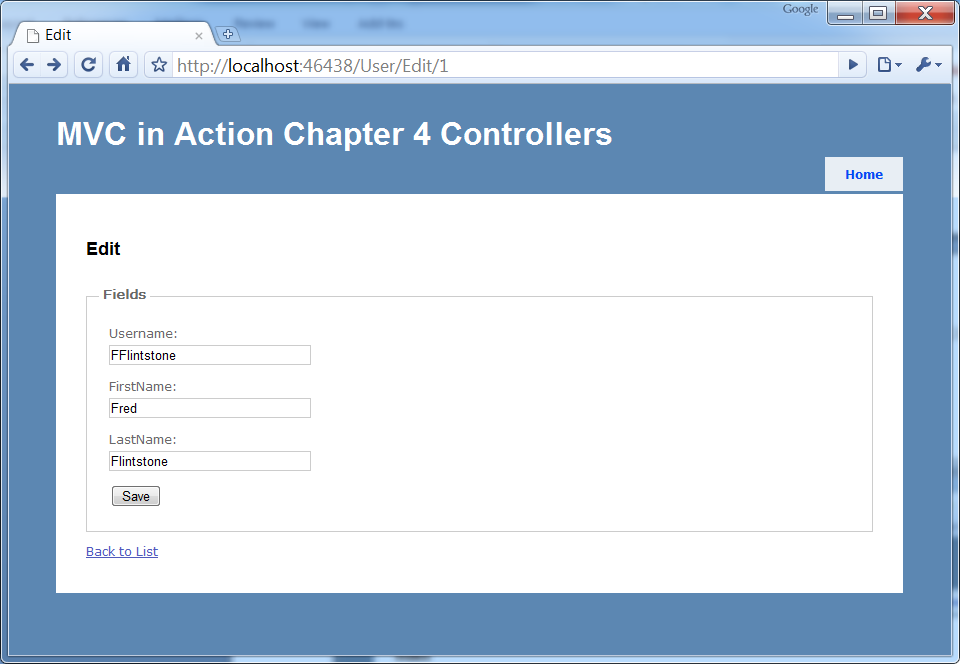


Figure 4.2 screenshot of the user edit view

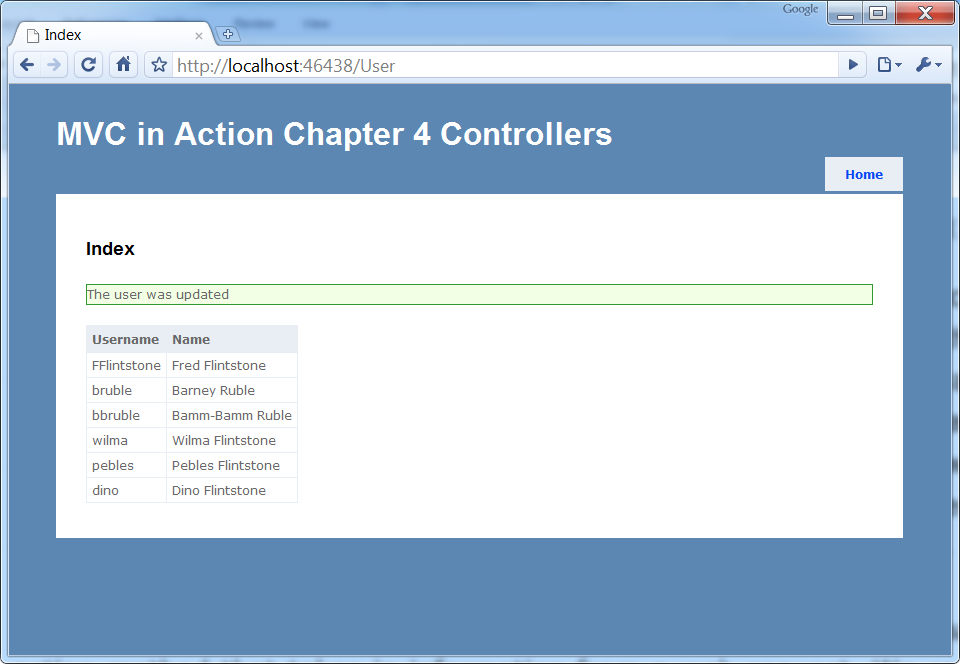


Figure 4.3 Screenshot of the redirected action showing a message from TempData

4.3.3 Handling the Failure processing of the Action Input

The code listing 4.6 has the alternate path when the ModelState.IsValid returns a value of false. This occurs when the Username is not entered in the form. The ModelBinder utilizes the built in validation framework that uses DataAnnotation attributes to designate how the Input Model should be validated. When an empty value for the Username is posted to the server, an error message is automatically added to the ModelState object. Leaning on the Model Binders allows the code in the failure path to be very simple and clean. In listing 4.6 , The alternate path is simply the return View(input). The ModelState is sent the view so no additional code is required in the Action to send a useful error message to the user. Figure 4.4 shows a screenshot of the form with the ModelState validation errors show on the screen.

Listing 4.6 The alternate path

public class UserInput |#A

{ |#A

[Required] |#A

public string Username { get; set; } |#A

public string FirstName { get; set; } |#A

public string LastName { get; set; } |#A

}

[HttpPost]

public ActionResult Edit(UserInput input)

{

if (ModelState.IsValid) |#B

{

...

}

return View(input); |#C

}

#A InputModel bound from a form Post

#B The IsValid property is false

#C Return the input model back to the view

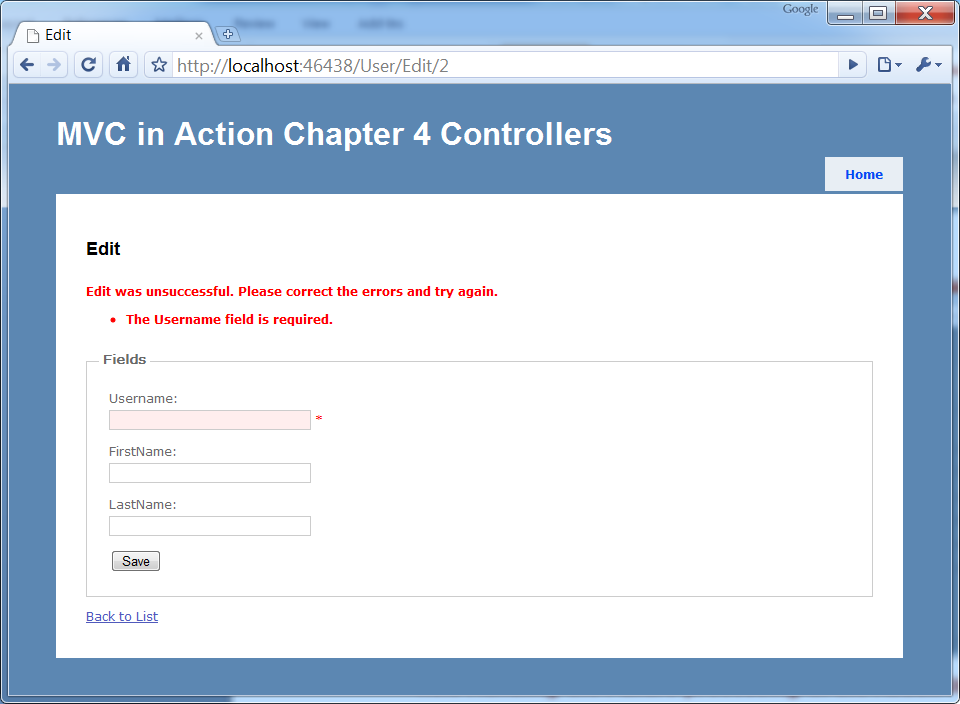


Figure 4.4 Screenshot of the alternate path showing validation messages

Handling the alternate path of the storyboard, in terms of the code is pretty simplistic. That is by design. Do not let yourself be fooled by this simplicity, it is still important to unit test your controller actions.

4.4 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 are small, scripted tests, usually written in the same language as the production code. They set up and exercise a single component's function. 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.

4.4.1 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.

A cueball in code and text

Listing 4.7 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); #A

RedirectToRouteResult result = controller.NextConference(); #B

Assert.That(result.RouteValues["controller"], |#C

Is.EqualTo("conference")); |#C

Assert.That(result.RouteValues["action"], |#C

Is.EqualTo("index")); |#C

Assert.That(result.RouteValues["conferenceKey"], |#C

Is.EqualTo("thekey")); |#C

}

private class ConferenceRepositoryStub : IConferenceRepository #1

{

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)

{

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();

}

}

}

}

#A Create using simulated dependencies

#B Exercise class under test

#C Assert correct results

Notice that most of the code listing is test double code, and not the RedirectController test itself. Test doubles are classes that stand in for object dependencies. They simulate collaborators so that we can control the test environment. For more information on test double, Roy Osherove has written a very nice book called The Art of Unit Testing.

We have to stub out an IConferenceRepository implementation (1) because calling that interface inside the controller action provides the next conference. How it performs that data query is beyond the scope of this chapter and is irrelevant to the controller; however, you may briefly skip ahead to Chapter 23 if you are curious about how to write data access code when using ASP.NET MVC. 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.

4.4.2 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 4.5, 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.

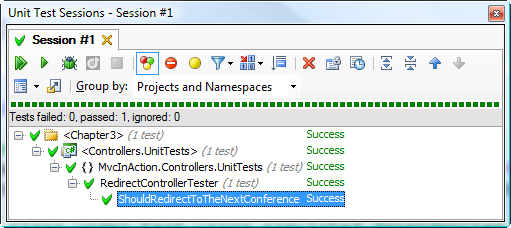


Figure 34.5 This window shows a controller unit testing passing.

4.4.3 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 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 4.7 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 and found at <http://www.ayende.com/projects/rhino-mocks.aspx>. Rhino Mocks drastically reduces the number of lines of code in a unit test fixture by streamlining the creation of test doubles. If code is designed so that all dependencies are injected into the constructor, as shown in listing 4.8, 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 4.8 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 4.9 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 4.9 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 = |#A

MockRepository.GenerateStub<IConferenceRepository>(); |#A

repository.Stub(r => |#B

r.GetNextConference()).Return(conferenceToFind); |#B

var controller = new RedirectController(repository);

RedirectToRouteResult result = controller.NextConference();

Assert.That(result.RouteValues["controller"], |#C

Is.EqualTo("conference")); |#C

Assert.That(result.RouteValues["action"], |#C

Is.EqualTo("index")); |#C

Assert.That(result.RouteValues["conferenceKey"], |#C

Is.EqualTo("thekey")); |#C

}

}

}

#A Stub using Rhino Mocks

#B Return a specific conference

#C Assert correct results

A dynamic mocking library like Rhino Mocks is not appropriate in every unit testing scenario. The usage in listing 4.9 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.

4.4.4 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 entire books on that, and we again recommend reading The Art of Unit Testing. This 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.

4.5 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 model binders. This leave the action methods free to concentrate on implementing an applications storyboard. By handling the happy path and the alternate path, it becomes very easy to spot actions that are taking on too many branches of logic.

Wielded without caution, controllers have the potential because 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.