26

Testing Practices

This chapter covers:

* Designing and testing routes
* Unit testing controllers
* Unit testing custom model binders
* Unit testing action filters

Testing is a key tenet of any type of engineering, and software engineering is no different. Because software needs to be fully retested on every new build, the act of executing test cases can be slow and error-prone if done by hand. Creating automated tests is an accepted best practice, and ASP.NET MVC eases the effort in creating automated test cases.

Because controllers are normal classes, and actions are merely methods, we can load and execute actions and then examine the results. Even though testing controllers is simple, we must consider an important caveat. When we test a controller action, we are only able to write assertions for the behavior we can observe. The true test of a working application is running it in a browser, and there are significant differences between viewing a page in a browser and asserting results in a controller action test.

First, we don’t even know if a particular URL will even end up executing our controller unless we test it. We can make sure that the correct view is chosen, but we cannot assert that the correct view is shown at runtime. We can assert that we put correct information into ViewData, but we cannot ensure that the view uses all of the information we give it. We also cannot assert that all possible controller code paths place the necessary objects into ViewData. With action filters, it is quite possible that a view will need data that is not present. Controller action tests don’t run the entire MVC engine, so things like ActionFilters are not executed. Although action unit tests add value, they don’t replace end-to-end application-level testing. This chapter will dive into writing automated unit tests for some of the most common bits of code you will write in an ASP.NET MVC application. If you are new to automated unit testing, be sure to pick up *The Art of Unit Testing* by Roy Osherove, published by Manning Publications. We will begin by exploring how to create automated test cases for routes.

26.1 Testing routes

Routing is perhaps the biggest innovation of the ASP.NET MVC project—so big, in fact, it was included in the .NET Framework 3.5 SP1 release, well ahead of the ASP.NET MVC release. With .Net 4.0, routing is merged into System.Web.dll and will be considered a core part of ASP.NET. Given that it will become part of the standard ASP.NET request pipeline just like http modules, you are well-served to invest some learning in how to design routes for testability as well as how to test them.

Like any new tool, routing is easy to abuse. Unless routes are tested thoroughly, changes to routes can break existing URLs. When URLs become public, changing them can break links, bookmarks, lower search rankings, and anger end users. Designing of custom routes and URL patterns should come from actual business requirements. In this section, we’ll examine some practices for testing routes to ensure we don’t break our application.

When we do need custom routes, we need to ensure both that the routes we are creating are correct, and any existing routes are not modified. We can start off with the built-in routes, and lock those down with tests. The default route is shown in listing 26.1.

Listing 26.1 Default routes in a new application

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

"Default", // Route name

"{controller}/{action}/{id}", // URL with parameters

new {controller = "Home", action = "Index", id = UrlParameter.Optional}

// Parameter defaults

);

For many applications, this route is sufficient and does not necessarily need to be tested on its own. If we added additional routing behavior, we would want to ensure that existing routes that follow this format are not broken. Before we start writing tests, we need to think of a few scenarios. The following URLs shown in table 26.1 should work in the default sample application.

Table 26.1 The default sample application supports these urls.

|  |  |
| --- | --- |
| URL | Result |
| “/” | HomeController.Index() |
| “/home” | HomeController.Index() |
| “/home/index” | HomeController.Index() |
| “/home/index/5” | HomeController.Index(5) |
| “/home/index?id=5” | HomeController.Index(5) |
| “/home/about” | HomeController.About() |

To make things more interesting, we’ll add a simple ProductController to list, view, and search products, as shown in listing 26.2.

Listing 26.2 Simplified product controller

using System.Web.Mvc;

using Routes.Models;

namespace Routes.Controllers

{

public class ProductController : Controller

{

public ViewResult Index() #1

{

var products =

new[]

{

new Product {Name = "DVD Player"},

new Product {Name = "VCR"},

new Product {Name = "Laserdisc Player"}

};

return View(products);

}

public ViewResult Show(int id) #3

{

return View(new Product {Name = "Hand towels"});

}

public ViewResult Search(string name) #2

{

return View("Show", new Product {Name = name});

}

}

}

Product controller supports a default list **(1)**, a search **(2)**, and a show **(3)** function. Each action uses the default view name. By default, the actions will be exposed by the default route, but, we want to support more interesting URL scenarios:

* “/product/show/5” maps to ProductController.Show
* “/product/SomeProductName” maps to ProductController.Search(SomeProductName)

Out of the box, the built-in routes support the first scenario, but not the second. Before we start messing around with our routes, we need to add tests to our existing scenarios. Testing routes is possible, but much easier with the testing extensions of the open source project, MvcContrib. We’ll test the first scenario, as shown in listing 26.3.

Listing 26.3 Testing a blank URL

[Test]

public void Should\_map\_blank\_url\_to\_home()

{

"~/".Route().ShouldMapTo<HomeController>(c => c.Index());

}

Using extension methods contained in the namespace MvcContrib.TestHelper, the test first transforms a string into a Route object with the Route extension method. Next, we use the ShouldMapTo extension method to assert that a route maps to the Index method on HomeController. ShouldMapTo<T> is a generic method, taking an expression. It is similar to other expression-based methods such as Html.TextBoxFor<T>. The expression is used to perform strongly typed reflection, as opposed to doing something like passing the controller and action name in as strings, which will fail under refactoring scenarios. Unfortunately, this test does not pass yet, as we have not called anything to set up our routes. We’ll accomplish this in a test setup method to be executed before every test, as shown in listing 26.4.

Listing 26.4 Registering the routes in a setup method

[SetUp]

public void Setup()

{

RouteTable.Routes.Clear();

MvcApplication.RegisterRoutes(RouteTable.Routes);

}

With our setup in place, our test now passes. The next scenarios we want to test are the other built-in scenarios, as shown in listing 26.5.

Listing 26.5 Testing the built-in routing scenarios

[Test]

public void Should\_map\_home\_url\_to\_home\_with\_default\_action()

{

"~/home".Route().ShouldMapTo<HomeController>(c => c.Index());

}

[Test]

public void Should\_map\_home\_about\_url\_to\_home\_matching\_method\_name()

{

"~/home/about".Route().ShouldMapTo<HomeController>(c => c.About());

}

[Test]

public void

Should\_map\_product\_show\_with\_id\_to\_product\_controller\_with\_parameter()

{

"~/product/show/5".Route().ShouldMapTo<ProductController>(

c => c.Show(5));

}

With default scenarios added, we can now proceed with modifying our route to support the special case of a search term directly in the URL. Before we get there, let’s make sure our routes don’t already support this scenario by adding a test to verify the functionality. After all, if this test passes, our work is done! The new test is shown in listing 26.6.

Listing 26.6 New scenario routing product search terms

[Test]

public void

Should\_map\_product\_search\_to\_product\_controller\_with\_parameter()

{

"~/product/SomeProductName" #1

.Route()

.ShouldMapTo<ProductController>(

c => c.Search("SomeProductName")); #2

}

This new test tries to prove that a route with some product name **(1)** in it will map to the “search” action **(2)** passing in the product name. Alas, our test fails, and our work is not yet done. The test fails with the message “MvcContrib.TestHelper.AssertionException : Expected Search but was SomeProductName.” To make our test pass, we need to add the appropriate changes to the routes, as shown in listing 26.7.

Listing 26.7 Additional route for searching products

routes.MapRoute(

"SearchProduct",

"product/{name}",

new {controller = "Product", action = "Search"}

);

With this addition to our routes, our new test passes, along with all the other tests. We were able to add a new route to our routing configuration with the assurance that we would not break the other URLs. Since URLs are now generated through routes in an MVC application, testing our routes becomes of utmost importance. The test helpers in MvcContrib wrapped all the ugliness that usually comes along with testing routes. In the next section, we’ll examine how to avoid unnecessary test complexity.

26.2 Avoiding test complexity

Any behavior decision an application makes must be tested: either manually or through a automated test. If you add complexity to an application, you add to the testing burden. By keeping the behavior simple, you drastically reduce the number of test cases that you have to write. This applies very specifically to how routes leverage controller and action names.

Although the default routes in an MVC application match a URL to a method name on a controller, the defaults can be changed. As shown in section 12.3.1, we can map the second URL segment to a parameter on a specific action. When using the MVC extension points of the ActionNameSelectorAttribute and ActionMethodSelectorAttribute, the name of an action method on a controller does not exactly match the method name. The two concepts of action name and action method name are completely separate, and can be configured independently. We can configure an action method as shown in listing 26.8 to modify the action name.

Listing 26.8 Modifying the action name for an action method

using System.Web.Mvc;

namespace Routes.Controllers

{

public class ChangedActionNameController : Controller

{

[ActionName("Foo")] #2

public ActionResult Index() #1

{

return View();

}

}

}

In the controller shown in listing 26.8, we specified that the action method name **(1)** should be different from the action name **(2)**. The action name, originally “Index”, is now “Foo”. Navigating to “/changedactionnname” or “/changedactionname/index” now results in a 404 Not Found error. The action name is now “Foo”, and we can only access this action through “/changedactoinname/foo”. As view names correspond to action names, not action method names, our view is named “Foo.aspx”.

But in most applications, we are better served adhering to the convention that action names match action method names. If method names differ from action names, we can no longer use expression-based URL generators. Our URL generation is now susceptible to subtle refactoring and renaming errors. This can be alleviated by introducing global constants for action names, but it still creates a string-based system, with another level of indirection between action methods and action names, that is not needed in many cases. In short, unless there is no other way, do not use ActionnNameAttribute.

26.3 Testing controllers

For controllers to be maintainable, they should be as light and skinny as possible, delegating all real domain work to other objects. Our controller tests will reflect this choice, as assertions will be small and target only the following:

* What ActionResult was chosen
* What information was passed to the view, in ViewData or TempData

All other web-related information, whether it is security, cookies, or session variables should be encapsulated in a domain-specific and domain-relevant interface. Although it eases testing, encapsulation and separation of concerns are the most significant reasons to leave these other HttpContext-related items out of controllers. The simplest example of a controller action is one that simply passes data into a view, as shown in listing 26.9.

Listing 26.9 A simple action

public ViewResult Index()

{

Product[] products = \_productRepository.FindAll();

return View(products);

}

In this example, \_productRepository is a private field of type IProductRepository, as shown in listing 26.10.

Listing 26.10 The controller with its dependency

namespace UnitTestingExamples.Controllers

{

public class ProductsController : Controller

{

private readonly IProductRepository \_productRepository;

public ProductsController(IProductRepository productRepository)

{

\_productRepository = productRepository;

}

. . . snip . . .

}

}

When we test the ProductsController, we don’t need to supply the actual implementation of the IProductRepository interface. For the purposes of a unit test, we are testing only the ProductsController and no external dependency used. To maximize the localization of defects, our unit tests should test only a single class. We don’t want a controller unit test to fail because we have a problem with our local database. In a unit test, we’ll have to pass a test double into the ProductsController repository. A test double is a stand-in for an actual implementation, but one that we can manipulate to force our class under test to execute specific code paths. Our controller unit test will need to set up the stubbed IProductRepository with dummy data, and then assert that the right action result is used, the right view is chosen, and the right data is passed to the view, as shown in listing 26.11.

Cueballs in code and text

Listing 26.11 Testing our Index action

[Test]

public void Index\_should\_use\_default\_view\_and\_repository\_data()

{

var products = new[] #1

{

new Product {Name = "Keyboard"},

new Product {Name = "Mouse"}

};

var repository = |#2

MockRepository.GenerateStub<IProductRepository>(); |#2

repository.Stub(rep => rep.FindAll()).Return(products); #3

var controller = new ProductsController(repository); #4

ViewResult result = controller.Index(); #5

Assert.AreEqual("", result.ViewName); #6

Assert.AreEqual(products, result.ViewData.Model); #7

}

We set up product data for our test (1). The values inside don’t matter for the purposes of our unit test, but aid in debugging if our test fails for an unknown reason. We create a stub of our IProductRepository by calling a RhinoMocks API (2). Rhino Mocks is a popular test double creation and configuration framework. You can find this library, created by Oren Eini, at <http://www.ayende.com/projects/rhino-mocks.aspx>. After we create a test double of our IProductRepository, we stub out the call to FindAll to return our array of Products we created earlier (3). With the stubbed IProductRepository, we create a ProductsController (4).

With all of the classes and test doubles set up for our unit test, we can execute our controller action and capture the resulting ViewResult object (5). We assert that the ViewName should be an empty string (6) (signifying we use the Index view), and that the model passed to the view is our original array of products (7). Our test passes with the implementation of our action shown in listing 26.9.

A two-line action method is tested easily, but is not very interesting. In a more interesting scenario, we edit a model, then post it to a form. We expect several things to happen:

* Check the model state for errors
* If errors exist, show the original view
* If not, save the model and redirect back to the index

Let’s start with the error path, where a user entered incorrect information. We’ll assume that model state errors are populated through other means as a result of validation, perhaps through a model binder or action filter. For the purposes of our test, shown in listing 26.12, the means of validation is not important, but rather, how the controller behaves under this condition.

Cueballs in code and text

Listing 26.12 Testing the edit action when errors are present

[Test]

public void Edit\_should\_redirect\_back\_when\_model\_errors\_present()

{

var badProduct = new Product {Name = "Bad value"};

var repository =

MockRepository.GenerateStub<IProductRepository>();

var controller = new ProductsController(repository); |#1

controller.ModelState |#1

.AddModelError("Name", |#1

"Name already exists");

ActionResult result = controller.Edit(badProduct); #2

Assert.AreEqual("", result.AssertViewRendered().ViewName); #3

repository.AssertWasNotCalled(rep => rep.Save(badProduct)) #4;

}

This test uses the MvcContrib.TestHelper library for an easy test API. To force our controller into an invalid model state, we need to add a model error to ModelState with the AddModelError method (1). After setting up our controller, we invoke the Edit action (2), and examine the result returned. We assert that a view is rendered with the AssertViewRendered method (3), which returns a ViewResult object. The ViewName on the ViewResult should be an empty string, signifying the Edit view is rerendered. Finally, we assert that the Save method on our repository was not called (4). This negative assertion ensures we don’t try to save our Product if it has validation problems. Normally, we would create a separate presentation model specifically for the form, but in this example, we use our domain model directly. We tested the error condition, and now we need to test our controller in the positive condition that our model didn't have any validation problems, as shown in listing 26.13.

Listing 26.13 Testing our controller action when no errors are present

[Test]

public void

Edit\_should\_save\_and\_redirect\_when\_no\_model\_errors\_present()

{

var goodProduct = new Product {Name = "Good value"};

var repository =

MockRepository.GenerateStub<IProductRepository>();

var productsController = new ProductsController(repository);

ActionResult result = productsController.Edit(goodProduct); #1

repository.AssertWasCalled(rep => rep.Save(goodProduct));

var redirectResult = result as RedirectToRouteResult; |#2

Assert.IsNotNull(redirectResult); |#2

Assert.AreEqual(1, redirectResult.RouteValues.Count); |#3

Assert.AreEqual("index", redirectResult.RouteValues["action"]); |#3

}

In this test, we set up our dummy product and controller in a manner similar to the last test, except this time we don’t add any model errors to our ModelState. We invoke the Edit action with the product we created (1), and then verify values on the result. We cast to a RedirectToRouteResult to ensure the type we expect **(2)**. Then we assert that the correct action name is in the route values **(3)**. To make both of these tests pass, our action looks like listing 26.14.

Cueballs in code and text

Listing 26.14 Implementation of the Edit action

[HttpPost]

public ActionResult Edit(Product product)

{

if (!ModelState.IsValid) #1

{

return View(product); #2

}

\_productRepository.Save(product);

return RedirectToAction("index"); #3

}

In our Edit action, we check for any ModelState errors with the IsValid property (1), and return a ViewResult with our original Product. Our Edit view likely will use styling to highlight individual model errors and display a validation error summary. If there are no validation errors, we save the Product and redirect back to the Index action (3). With our controller’s behavior locked down sufficiently, we can feel confident we can modify our Edit action in the future and know if our change breaks existing functionality. In the next section, we’ll examine strategies for testing custom model binders.

26.4 Testing model binders

Custom model binders eliminate much of the boring plumbing that often clutters action methods with code not pertinent to the true purpose of the action method. But with this powerful tool comes the need for thorough testing. Our infrastructure needs to be rock solid, as it can execute on a large majority of requests.

Testing model binders is not as straightforward as testing action methods, but it is possible. The amount of testing needed varies depending on what you are doing with your custom model binder. Simply implementing the IModelBinder interface likely means you'll only need to worry about one single BindModel method and only a ModelBindingContext during testing. Inheriting from DefaultModelBinder is a bit more challenging, as any code we add will execute alongside other code that we don’t own. We must ensure that any behavior we add works correctly in the context of the other responsibilities of the base DefaultModelBinder class. The DefaultModelBinder class design has extensibility in mind, and key extension points are available through specific method overrides, but we still need to test these methods in the context of an entire binding operation (such as a single BindModel call).

In section 12.1.3, we examined creating a custom model that bound entities from a repository, as shown in listing 26.15. If you have implemented a custom model binder in ASP.NET MVC 1, you will notice the redesigned value provider API.

Listing 26.15 Implementing an entity model binder

using System;

using System.Web.Mvc;

using UnitTestingExamples.Models;

using UnitTestingExamples.Services;

namespace UnitTestingExamples.Helpers.Binders

{

public class EntityModelBinder : IModelBinder

{

public object BindModel(

ControllerContext controllerContext,

ModelBindingContext bindingContext)

{

ValueProviderResult value =

bindingContext.ValueProvider

.GetValue(bindingContext.ModelName);

if (value == null) #1

return null;

if (string.IsNullOrEmpty(value.AttemptedValue)) #1

return null;

Guid entityId;

entityId = new Guid(value.AttemptedValue);

Type repositoryType =

typeof (IRepository<>).MakeGenericType(

bindingContext.ModelType);

var repository = (IRepository) IoC.Resolve(repositoryType);

PersistentObject entity = repository.GetById(entityId);

return entity;

}

}

}

We have several guards **(1)** protecting against bad input. However, we didn't include the check for a user or part of our application puting an invalid GUID into the querystring (or form variable). Rather than allow an exception to be thrown during binding, we would like to handle this by returning null, as shown in the test in listing 26.16.

Cueballs in code and text

Listing 26.16 Test for bad GUID values

[Test]

public void Should\_bind\_to\_null\_when\_guid\_not\_in\_correct\_format()

{

var collection = new NameValueCollection(); |#1

collection.Add("NotAGuid", "NotAGuid"); |#1

var provider = new NameValueCollectionValueProvider( |#1

collection, CultureInfo.InvariantCulture); |#1

var bindingContext = new ModelBindingContext |#2

{ |#2

ModelName = "ProductId", |#2

ValueProvider = provider |#2

}; |#2

var binder = new EntityModelBinder();

object model = binder.BindModel(null, bindingContext);

Assert.IsNull(model);

}

Our model binder uses only a ModelBindingContext, not the ControllerContext. We need only focus on creating a ModelBindingContext representative of an invalid GUID value. First, we create a value provider (1). For the key and value in the value provider’s collection we’ll substitute bad GUID values, to force our model binder to throw an exception. We can now create our ModelBindingContext (2), using the same ModelName as was used in our value provider. Because we use the ModelName directly to look up values in our model binder, any mismatch will cause our custom model binder to not execute the code we are interested in. When we execute this unit test, it fails with a System.FormatException, because our model binder is not yet able to handle invalid GUIDs. To make our test pass, we can either parse the input string using regular expressions, or use a try..catch block. For simplicity, we’ll use the exception handling method, with the additions shown in listing 26.17.

Cueball in code and text

Listing 26.17 Modifying the GUID parsing code to handle invalid values

Guid entityId;

try

{

entityId = new Guid(value.AttemptedValue);

}

catch (FormatException) #1

{

return null;

}

With these changes, our test now passes. We surrounded our original GUID constructor with a try..catch block for the specific FormatException type thrown when the parsed value is not of the right format (1). There are other interesting scenarios we could add tests for, but all of them employ the same technique of creating a ModelBindingContext representative of a certain model binding scenario. Unit tests for model binders go quite a long way to proving the design of a model binder, but still don’t guarantee a working application.

NOTE

Guid.TryParse and Enum.TryParse<T> are both being added to .NET 4. Up through .NET 3.5 SP1, there has not been a built-in way to see if a string is a valid GUID. There are plenty of regular expression solutions that you can find on the web. If you would like to look into this issue yourself, please browse the original Microsoft Connect issue and workarounds logged in 2004 at <http://connect.microsoft.com/VisualStudio/feedback/ViewFeedback.aspx?FeedbackID=94072>

Model binders are one cog in a larger machine, and only through testing that larger part can we have complete confidence in our model binders. It can often take quite a bit of trial and error to get the model binder to function correctly. When it is working correctly, we need only to construct the context objects used by our model binder in our unit test to recreate those scenarios. Unfortunately, merely looking at a model binder may not show you how to construct the context objects it uses. A common test failure is a NullReferenceException, where a call to an MVC framework method requires other supporting objects in place. The easiest way to determine what pieces your model binder needs in place is to simply write a test and see if it passes. If it does not pass because of an exception, keep fixing the exceptions, often by supplying test doubles, until your test passes or fails due to an assertion failure. In the next section, we’ll examine testing action filters.

26.5 Testing action filters

The story for testing action filters is very similar to that for testing model binders. Unit testing is possible, and its difficulty is directly proportional to how much the filter relies on the context objects. Generally, the deeper the filter digs in to the context object, the more we'll need to be set up or mocked in a unit test. Table 26.1 illustrates the types of filters and the context objects used for each.

Table 26.1 Filters and their supporting context objects

|  |  |  |
| --- | --- | --- |
| Filter Type | Method | Context Object |
| IActionFilter | OnActionExecuted | ActionExecutedContext |
| OnActionExecuting | ActionExecutingContext |
| IAuthorizationFilter | OnAuthorization | AuthorizationContext |
| IExceptionFilter | OnException | ExceptionContext |
| IResultFilter | OnResultExecuted | ResultExecutedContext |
| OnResultExecuting | ResultExecutingContext |

Each context object has its own difficulties for testing, as each has its own dependencies for usage. All context objects have a no-argument constructor, and a unit test may be able to use the context object as is without needing to supply it with additional objects. Although your filter may use only one piece of the context object, you may find yourself needing to supply mock instances of more pieces, as many of the base context object constructors have null argument checking. You may find yourself far down a long path that leads to supplying the correct dependencies for a context object, and these dependencies may be several levels deep. Let’s add tests to the filter shown in listing 26.18.

Listing 26.18 Creating a simple action filter

public class CurrentUserFilter : IActionFilter

{

private readonly IUserSession \_session;

public CurrentUserFilter (IUserSession session)

{

\_session = session;

}

public void OnActionExecuting(ActionExecutingContext filterContext)

{

ControllerBase controller = filterContext.Controller;

User user = \_session.GetCurrentUser();

if (user != null)

{

controller.ViewData.Add(user);

}

}

public void OnActionExecuted(ActionExecutedContext filterContext)

{

}

}

In this filter, we have the requirement that a User object is needed for a component in the view, likely for displaying the current user in a widget. Our CurrentUserFilter depends on an IUserSession, whose implementation contains the logic for storing and retrieving the current logged in user from the session. Our filter retrieves the current user and places it into the controller’s ViewData. The controller is supplied through the ActionExecutingContext object. If possible, during unit testing, we prefer to use the no-argument constructor and supply any additional pieces by merely setting the properties on the context object. The ActionExecutingContext type has setters for the Controller property, so we’ll be able to use the no-argument constructor and not worry about the larger, parameter-full constructor. Our complete unit test, shown in listing 26.19, is able to create a stub implementation for only the parts used in our filter.

Cueballs in code and text

Listing 26.19 Action filter unit test

using System.Web.Mvc;

using MvcContrib;

using NUnit.Framework;

using Rhino.Mocks;

using UnitTestingExamples.Helpers.Filters;

using UnitTestingExamples.Models;

using UnitTestingExamples.Services;

namespace UnitTestingExamples.Tests

{

[TestFixture]

public class CurrentUserFilterTester

{

[Test]

public void Should\_pass\_current\_user\_when\_user\_is\_logged\_in()

{

var loggedInUser = new User();

var userSession = |#1

MockRepository.GenerateStub<IUserSession>(); |#1

userSession.Stub(session => session.GetCurrentUser()) |#2

.Return(loggedInUser); |#2

var filterContext = new ActionExecutingContext

{

Controller = MockRepository.GenerateStub<ControllerBase>()#4

};

var currentUserFilter = new CurrentUserFilter(userSession);|#5

currentUserFilter.OnActionExecuting(filterContext); |#5

var user = filterContext.Controller.ViewData.Get<User>(); |#6

Assert.AreEqual(loggedInUser, user); |#6

}

}

}

Our CurrentUserFilter depends on an implementation of an IUserSession interface (1), which we supply using Rhino Mocks. Next, we stub the GetCurrentUser method on our stub IUserSession to return the User object created earlier (2). Because the actual implementation of IUserSession requires the full HttpContext to be up and running, by supplying a fake implementation, we get much finer control over the inputs to our filter object.

Next, we create our ActionExecutingContext (3), but call only the no-argument constructor. The controller can be any controller instance, and we again use Rhino Mocks to create a stub implementation of ControllerBase (4). Rhino Mocks creates a subclass of ControllerBase at runtime, which saves us from using an existing or dummy controller class. In any case, the ControllerBase provides ViewData, so we don’t need to provide any stub implementation for that property. With our assembled ActionExecutingContext and stubbed implementation of IUserSession, we can create and exercise our CurrentUserFilter (5). The OnExecutingMethod does not return a value, so we need to examine only the ActionExecutingContext passed in. We assert that the controller’s ViewData contains the same logged in user created earlier (6), and our test passes!

Getting to this point required trial and error to understand what the context object requires for execution. Because filters are integrated and specific to the MVC framework, it can be fruitless to try to write filters test-first, as only the fact that the complete website is up and running proves the filter is working properly. We supplied dummy implementations of the context objects, but constructed them in a way that the MVC framework will likely not use.

26.6 Summary

In this chapter, we explored testing some of the most popular types of code you will write with the ASP.NET MVC framework. You learned how to test routes using the test helpers available in MvcContrib. You also learned how to create automated tests for controllers, model binders, and action filters. Each of these types of code has special behavior, and each of these needs automated test cases. Because code can be executed on every request, it is vital to ensure code behaves as desired. The true test of a working MVC application is using it in a browser. Refer back to Chapter 20 on full-system testing for more than just unit testing. In Chapter 27, you will learn how to apply jQuery to create an autocomplete textbox.