26

Testing Practices

This chapter covers:

* Designing and testing routes
* Testing MVC components

12.3 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. 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.

12.3.1 Testing routes

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 12.26.

Listing 12.26 Default routes in a new application

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

routes.MapRoute(

"Default",

"{controller}/{action}/{id}",

new { controller = "Home", action = "Index", id = "" }

);

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 should work in the default sample application:

* “/” maps to HomeController.Index()
* “/home” maps to HomeController.Index()
* “/home/about” maps to HomeController.About()

To make things more interesting, we’ll add a simple ProductController to list, view, and search products, as shown in listing 12.27.

Listing 12.27 Simplified product controller

public class ProductController : Controller

{

public ViewResult Index()

{

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)

{

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

}

public ViewResult Search(string name)

{

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

}

}

With our new controller, 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 12.28.

Listing 12.28 Testing a blank URL

[Test]

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

{

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

}

Using extension methods, 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 is a generic method, taking an expression. It is similar to other expression-based methods such as Html.ActionLink. 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 12.29.

Listing 12.29 Registering the routes in a setup method

[TestFixtureSetUp]

public void Setup()

{

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 12.30.

Listing 12.30 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 12.31.

Listing 12.31 New scenario routing product search terms

[Test]

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

{

"~/product/SomeProductName"

.Route()

.ShouldMapTo<ProductController>(c => c.Search("SomeProductName"));

}

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 12.32.

Listing 12.32 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 action names and custom routes

12.3.2 Action naming

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 12.33 to modify the action name.

Listing 12.33 Modifying the action name for an action method

public class ChangedActionNameController : Controller

{

[ActionName("Foo")]

public ActionResult Index()

{

return View();

}

}

In the controller shown in listing 12.33, we specified that the action method name should be different from the action name. The action name, [originally the action method name (“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.

Consistency in action naming can reduce the complexity in your system. If your system is generally resource-based; that is, controllers are designed around individual entities (a ProductController and a UserController), RESTful style action names introduce both discoverability on the client side, and predictable design on the developer side. Given a controller designed for products, the user interactions we might want to support include:

* Listing products
* Showing one product
* Creating a new product
* Editing an existing product
* Deleting a product

Translated into controller actions, these would map to:

* Index
* Show
* New
* Edit
* Delete

Because MVC action methods can be configured to accept only certain HTTP verbs, such as POST, we can design our controller with a set of overloaded action methods, one for viewing a form and one for receiving the posted form. If we had some sort of Widget resource in our system, our WidgetsController would look similar to listing 12.34.

Cueballs in code and text

Listing 12.34 A RESTful style controller for managing Widget resources

public class WidgetsController : Controller

{

public ActionResult Index()

{

return View();

}

public ActionResult Show(int id)

{

return View();

}

public ActionResult New() #1

{

return View();

}

[AcceptVerbs(HttpVerbs.Post)] |#2

public ActionResult New(WidgetForm widget) |#2

{

return RedirectToAction("Index");

}

public ActionResult Edit(int id)

{

return View();

}

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Edit(WidgetForm widget)

{

return RedirectToAction("Index");

}

}

Actions that support GET and POST verbs, the New and Edit actions, can be overloaded so that one method responds to GET requests (1) and the other responds to POST requests (2). To be clear, these actions don’t match the definition of REST. However, the design would be simpler if all of the controllers dealing with a resource looked the same. If we went to a real RESTful architecture, using MvcContrib’s SimplyRestfulRouteHandler, we could support all of the standard REST actions and corresponding HTTP verbs. Regardless of whether we want to adopt REST, having every action that shows a single entity or resource called Show makes new features easier to learn and makes the application easier to maintain.

12.4 Testing

The separation of concerns that the MVC pattern provides significantly increases testability for .NET web applications. 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. We can assert 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. Before we examine the last mile of testing in UI tests, let’s see how we can lock down the behavior in the rest of our MVC application through unit testing.

12.4.1 Controller unit tests

For controllers to be maintainable, they should be as light and skinny as possible, delegating all real domain work to other services. 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 12.35.

Listing 12.35 A simple action

public ViewResult Index()

{

var products = \_productRepository.FindAll();

return View(products);

}

In this example, productRepository is a private field of type IProductRepository, as shown in listing 12.36.

Listing 12.36 The controller with its dependency

public class ProductsController : Controller

{

private readonly IProductRepository \_productRepository;

public ProductsController(IProductRepository productRepository)

{

\_productRepository = productRepository;

}

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 12.37.

Cueballs in code and text

Listing 12.37 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 = Stub<IProductRepository>(); #2

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

var productsController = new ProductsController(repository); #4

ViewResult result = productsController.Index(); #5

result.ViewName.ShouldEqual(string.Empty); #6

result.ViewData.Model.ShouldEqual(products); #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 method on our base test class (2). This method is a wrapper around Rhino Mocks, a popular test double creation and configuration framework. 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 12.35.

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 12.38, the means of validation is not important, but rather, how the controller behaves under this condition.

Cueballs in code and text

Listing 12.38 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 = Stub<IProductRepository>();

var productsController = new ProductsController(repository);

productsController |#1

.ModelState.AddModelError("Name", "Name already exists"); |#1

var result = productsController.Edit(badProduct); #2

result.AssertViewRendered().ViewName.ShouldEqual(string.Empty); #3

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

}

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 12.39.

Listing 12.39 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 = Stub<IProductRepository>();

var productsController = new ProductsController(repository);

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

result

.AssertActionRedirect() #2

.ToAction<ProductsController>(c => c.Index()); #3

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

}

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 use the MvcContrib project’s AssertActionRedirect (2) to assert that the result of our action redirects to another action, specifically to the Index action. The ToAction method allows us to assert that we redirect to a specific action using a strongly typed expression (3). Because we use expressions here, our test won’t break if we rename the Index action method name. To make both of these tests pass, our action looks like listing 12.40.

Cueballs in code and text

Listing 12.40 Implementation of the Edit action

[AcceptPost]

public ActionResult Edit(Product product)

{

if (!ModelState.IsValid) #1

{

return View(product); #2

}

\_productRepository.Save(product);

return this.RedirectToAction(c => c.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.

12.4.2 Model binder unit tests

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 12.41.

Listing 12.41 Entity model binder implementation

public object BindModel (

ControllerContext controllerContext,

ModelBindingContext bindingContext)

{

ValueProviderResult value =

bindingContext.ValueProvider [bindingContext.ModelName];

if (value == null)

return null;

if (string.IsNullOrEmpty(value.AttemptedValue))

return null;

var 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 didn't add any tests in our original example, so let’s add some now. We have several guard-clauses 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 12.42.

Cueballs in code and text

Listing 12.42 Test for bad GUID values

[Test]

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

{

var valueProviderDictionary = new ValueProviderDictionary(null)

{

{

"ProductId",

new ValueProviderResult ("NotAGuid", "NotAGuid", null) #1

}

};

var bindingContext = new ModelBindingContext #2

{

ModelName = "ProductId",

ValueProvider = valueProviderDictionary

};

var binder = new EntityModelBinder ();

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

model.ShouldBeNull();

}

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 ValueProviderDictionary, with a single entry for a ProductId parameter (1). For the raw and attempted values in the ValueProviderResult, we’ll substitute bad GUID values, to force our model binder to throw an exception. With our ValueProviderDictionary assembled, we can create our ModelBindingContext (2), using the same ModelName as was used in our ValueProviderDictionary. Because we use the ModelName directly to look up ValueProviderResults 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 12.43.

Cueball in code and text

Listing 12.43 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.

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.

12.4.3 Action filter unit tests

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 12.1 illustrates the types of filters and the context objects used for each.

Table 12.1 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 12.44.

Listing 12.44 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 12.45, is able to create a stub implementation for only the parts used in our filter.

Cueballs in code and text

Listing 12.45 Action filter unit test

[TestFixture]

public class CurrentUserFilterTester : TestClassBase

{

[Test]

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

{

var loggedInUser = new User();

var userSession = Stub<IUserSession>(); #1

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

.Return(loggedInUser); |#2

var filterContext = new ActionExecutingContext #3

{

Controller = Stub<ControllerBase>() #4

};

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

currentUserFilter.OnActionExecuting(filterContext); |#5

filterContext.Controller.ViewData |#6

.Get<User>().ShouldEqual(loggedInUser); |#6

}

}

Our CurrentUserFilter depends on an implementation of an IUserSession interface (1), which we supply using the Stub method. The Stub method comes from the TestClassBase class, and is a wrapper around Rhino Mocks’ CreateStub method. 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. In the next section, we’ll examine how to automate tests with the entire website up and running through automated UI tests.

12.4.4 Testing the last mile with UI tests

In this chapter thus far, we examined testing individual components of ASP.NET MVC, including routes, controllers, filters and model binders. Although unit testing each component in isolation is important, the final test of a working application is interaction with a browser against a live instance. With all of the components that make up a single request, whose interaction and dependencies can become complex, it is only through browser testing that we can ensure our application works as desired from end-to-end. While developing an application, we often launch a browser to manually check that our changes are correct and produce the intended behavior.

In many organizations, manual testing is formalized into a regression testing script to be executed by development or QA personnel before a launch. Manual testing is slow and quite limited, as it can take minutes to execute a single test. In a large application, regression testing is minimal at best and woefully inadequate in most situations. Fortunately, many free automated UI testing tools exist. Some of the more popular tools are:

* WatiN (<http://watin.sourceforge.net/>)
* Watir (<http://wtr.rubyforge.org/>)
* Selenium (<http://seleniumhq.org/>)
* QUnit (<http://docs.jquery.com/QUnit>) —for testing JavaScript

In addition to these open source projects, many commercial products on the market provide additional functionality or integration with bug reporting systems or work item tracking systems, such as Microsoft’s Team Foundation Server. However, the tools are not tied to any testing framework, so integration with an existing project is rather trivial.

In this section, we’ll examine UI testing with WatiN, which provides easy integration with unit testing frameworks. WatiN, an acronym of Web Application Testing In .NET, is a .NET library that provides an interactive browser API to both interact with the browser, by clicking links and buttons for example, as well as find elements in the DOM.

Testing with WatiN usually involves interacting with the application to submit a form, then checking the results in a view screen. Because WatiN is not tied to any specific unit testing framework, we can use any unit testing framework we like. The testing automation platform Gallio (<http://www.gallio.org/>) provides important additions that make automating UI tests easier:

* Test steps for logging individual interactions in a single test
* Running tests in parallel
* Ability to embed screenshots in the test report (for failures)

To get started, we need to download and install Gallio. Gallio includes an external test runner (Icarus), as well as integration with many unit testing runners, including TestDriven.NET, ReSharper, and others. Also included in Gallio is MbUnit, a unit testing framework which we’ll use to author our tests. With Gallio downloaded and installed, we need to create a Class Library project and add references to both Gallio.dll and MbUnit.dll. Next, we need to download WatiN and add a reference in our test project to the WatiN.Core.dll assembly. With our project references done, we are ready to create a simple test. One of the most basic, but useful scenarios in our application is to test to see if we can log in to our application. Testing manually, this would mean:

1. Navigating to the login URL

2. Entering username and password

3. Clicking the Log in button

4. Checking that the login widget at the top of the screen has the correct name

Because we’ll want common functionality and configuration in all of our test classes that use WatiN, we’ll create a base test class, as shown in listing 12.46.

Listing 12.46 Web test base class

[TestFixture]

[ApartmentState (ApartmentState.STA)]

public class WebTestBase

{

}

The first attribute on our WebTestBase class should be familiar; it is the MbUnit attribute for tagging a class as a TestFixture. The next attribute is not as well known. Because WatiN uses COM to communicate with Internet Explorer (IE), and the COM IE wrapper is not thread-safe. We must configure our unit test runner to use a single-threaded apartment (STA). Each unit test runner is configured differently and in MbUnit’s case, we use the ApartmentStateAttribute with and ApartmentState value of ApartmentState.STA. With this attribute applied to our WebTestBase class, we need to configure this setting only once in our test project, as long as all of our tests use WebTestBase as a base class.

Next, we can create a new test that performs the steps listed earlier in this section, as shown in listing 12.47.

Listing 12.47 Testing the login screen

public class LoginScreen : WebTestBase

{

[Test]

public void Can\_log\_in\_successfully()

{

using (var ie = new IE ("http://localhost:8082/Login")) #1

{

ie.TextField (Find.ByName ("Username")).TypeText("admin"); #2

ie.TextField (Find.ByName ("Password")).TypeText("password");

ie.Button(Find.ByName ("login")).Click();

Assert.IsTrue(ie.ContainsText ("Joe User"));

}

}

}

Our LoginScreen class inherits from the WebTestBase class created earlier. In the LoginScreen test class, we define one test, “Can\_log\_in\_successfully.” Inside this test, we first create a new instance of the WatiN IE object (1). The IE class has a constructor that takes a URL as a parameter, which causes the IE browser to immediately launch at the specified URL. We hardcoded the correct starting URL so that the IE browser immediately navigates to the login screen. If the starting URL needs to be configured, we could pull this information from a configuration file. The lifetime of the IE object is wrapped in a using statement block, to ensure that our COM resources are disposed of properly.

The IE object is our primary source of interaction with the browser. It includes a variety of methods to locate elements in the DOM, as well as methods to interact with the browser’s periphery, such as cookies, dialog boxes, and so on. Our interaction will deal mainly with locating and manipulating DOM elements, but other browser interaction is available if needed. Back in our test, the next two lines use the TextField method (2) to locate the HTML INPUT elements of type TEXT. The TextField method takes a variety of arguments, each enabling a different way to search for elements. With ASP.NET MVC, we can use the Constraint overload, and use the Find static class to build a Constraint object to match the element we need. Other options include a string for an element ID, a regular expression, or a custom callback function. For our purposes, we’ll stick mainly with the Find.ByName constraint. With ASP.NET Web Forms, it was more common to use regular expressions, as element ID and names were not entirely deterministic. The MVC framework gives us complete control over element IDs and names.

The TextField method returns a single TextField object. We use the TypeText method to fill in text into both the username and password fields. In this test, we didn't set up any login information beforehand, and we know that this login information will work for a clean build of CodeCampServer. Typically, we’ll set up all entities needed for a test in a setup method. After filling in the username and password, we use the Button method in combination with the Find.ByName constraint to locate the login button and click it with the Click method. If our login is correct, we’ll be redirected in the browser to the home page, and our user’s name will appear at the top. To verify this, we use the ContainsText with our user’s name and assert that our user’s name is found.

With our basic test in place, we can execute this test in the Gallio Icarus test runner, shown in figure 12.3.

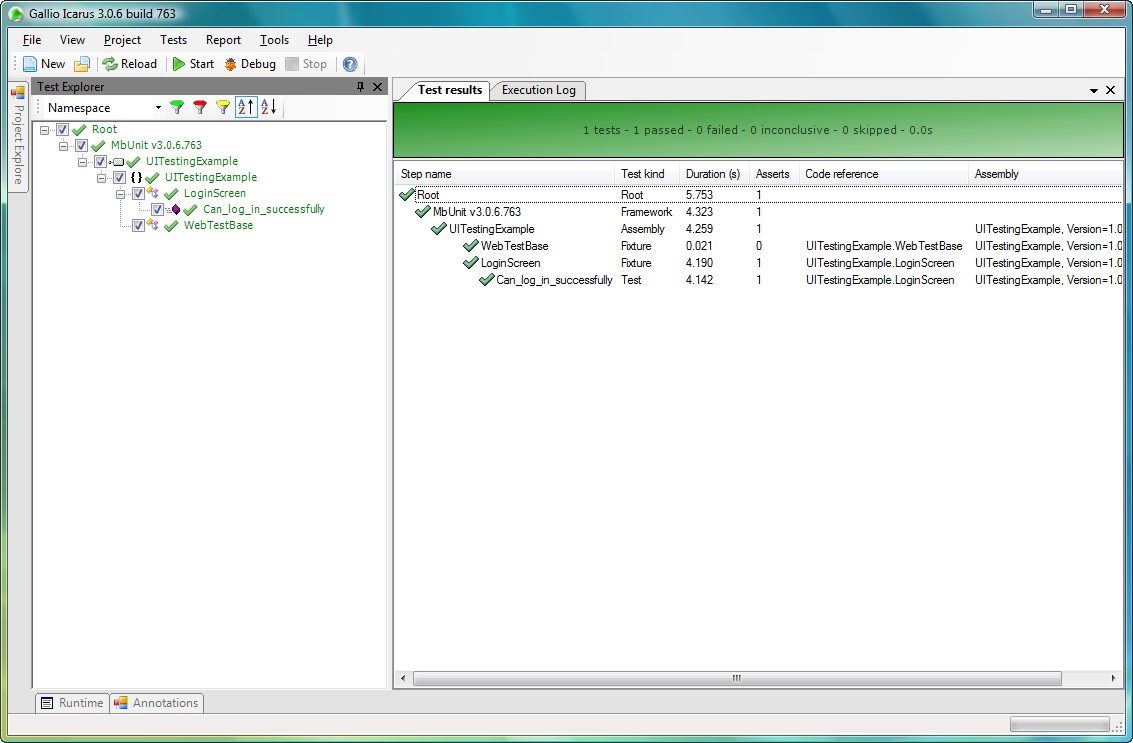


Figure 12.3 Simple passing login screen test

In our test, we referenced all of the input elements by name, but how did we know what name to look for? In older browsers, this meant viewing the HTML source. In modern browsers, including IE8 and Chrome, a built-in HTML inspector picks HTML elements by clicking them to bring the specific HTML element into a readable interface. Google’s Chrome HTML inspector, shown in figure 12.4, allows us to click an element on the browser to determine relevant information, such as element names.

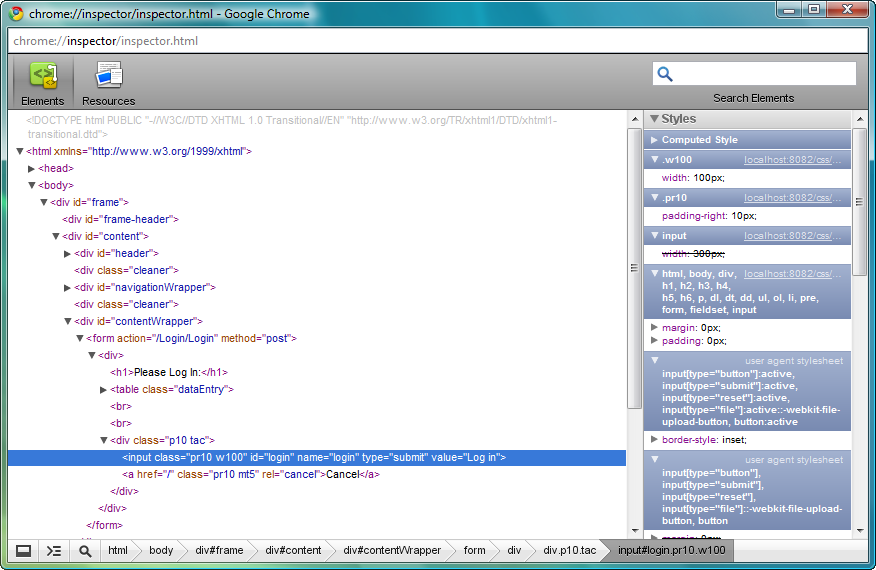


Figure 12.4 Google Chrome HTML inspector with our login button highlighted

Other browsers have extensions for this purpose, including Firebug (<http://getfirebug.com/>) for Firefox, and the IE Web Developer Toolbar (<http://www.microsoft.com/downloads/details.aspx?familyid=E59C3964-672D-4511-BB3E-2D5E1DB91038>) for versions previous to IE8. With these tools, we need only to click elements on a live browser and inspect their element names or IDs for our UI tests. In figure 12.2, we have the login button from our login screen selected, leaving guesswork or hunting through our project behind.

But what if our test fails? One of the features of MbUnit is the ability to embed images into test reports, and one of the features of WatiN is to capture images. First, we’ll create a failing test in as shown in listing 12.48.

Listing 12.48 Intentional test failure

[Test]

public void Intentional\_failure()

{

using (var ie = new IE ("http://localhost:8082/Login"))

{

ie.TextField (Find.ByName ("Username")).TypeText("admin");

ie.TextField (Find.ByName ("Password")).TypeText("password");

ie.Button(Find.ByName ("login")).Click();

Assert.IsTrue(ie.ContainsText ("Joe Schmoe"));

}

}

For this intentionally failing test, we change the text of the name asserted to an incorrect name, “Joe Schmoe.” Running this test proves our failure, but we would like to capture the screenshot as part of the failure. Because we created the WebTestBase class earlier, we can centralize all failure behavior in one place. We can create a teardown method, run after every test, and check to see if there were any failures in our test. If so, we take a screenshot using WatiN and embed the image into Gallio’s test results. To accomplish all of this, we’ll need to make more modifications to our WebTestBase class, as taking a screenshot requires the original instance of the IE object. Because our original test had the IE object in a using block, it won’t be available to our teardown method without modifications to our test. Instead of instantiating our IE object in each test, we’ll do so in our WebTestBase in a SetUp method, as shown in listing 12.49.

Listing 12.49 Modified WebTestBase setting up the IE object

protected IE Browser { get; private set; }

[SetUp]

public void SetUp()

{

Browser = new IE ("http://localhost:8082/Login");

}

Before each test executes, we create an IE instance and assign it to our protected Browser property. Our original failing test now needs to use the Browser property instead of creating the IE object itself, as shown in listing 12.50.

Listing 12.50 Modifying the failing test to use the Browser property

[Test]

public void Intentional\_failure()

{

Browser.TextField (Find.ByName ("Username")).TypeText("admin");

Browser.TextField (Find.ByName ("Password")).TypeText("password");

Browser.Button(Find.ByName ("login")).Click();

Assert.IsTrue(Browser.ContainsText ("Joe Schmoe"));

}

With our IE object now managed by our base test class, we can introduce a TearDown method to check for test failures and capture screenshots. Even if we didn't include the screenshot concept, we still need to add code in a teardown method to dispose of our IE instance properly. Our TearDown method is shown in listing 12.51.

Cueballs in code and text

Listing 12.51 Teardown method with image capturing and logging

[TearDown]

public void TearDown()

{

try

{

if (TestContext.CurrentContext

.Outcome.Status == TestStatus.Failed) #1

{

var writer = TestLog.Writer.Default; #2

using (writer.BeginSection("Test failed on this page")) #3

{

writer.Write("Url: ");

using (writer.BeginMarker(Marker.Link(Browser.Url))) #4

{

writer.WriteLine(Browser.Url);

}

var imageCapturer = new CaptureWebPage (Browser); #5

var image = imageCapturer #6

.CaptureWebPageImage(false, false, 100); #6

writer.EmbedImage("Failure.png", image);

}

}

}

finally

{

Browser.Close(); #7

Browser = null;

}

}

In a try-finally block, we separate the image capturing and logging from managing the IE instance. The IE browser should always be discarded at teardown, regardless of whether an exception happens during image capturing (7). The try-finally block ensures our IE instance is disposed properly. Inside the try block, we first check Gallio’s test status in the TestContext object (1). We only want to capture screenshots in the event of a failing test. Next, we create a reference to the default log writer for Gallio (2). Gallio supports multiple nested log streams for complex test reports, but in our case, the default will suffice.

To create sections in our log output, we use the BeginSection method (3). We might have more sections logged detailing the steps executed in our test, so a separate section for the error helps distinguish it in the final report. We also write the original URL of the screen with the error for informational purposes. Using the Marker.Link method (4) generates a clickable link in the final report, helpful to quickly traverse to the failing screen. We are ready to capture the image.

We create a CaptureWebPage object (5), passing in the IE instance stored in our test class. Next, we create an Image object and capture a screenshot using the CaptureWebPageImage method. We use the EmbedImage method (6) on our log writer object, providing the image object and a file name. Running this test in our Icarus test runner gives us a nice screenshot of our failure, as shown in figure 12.5.

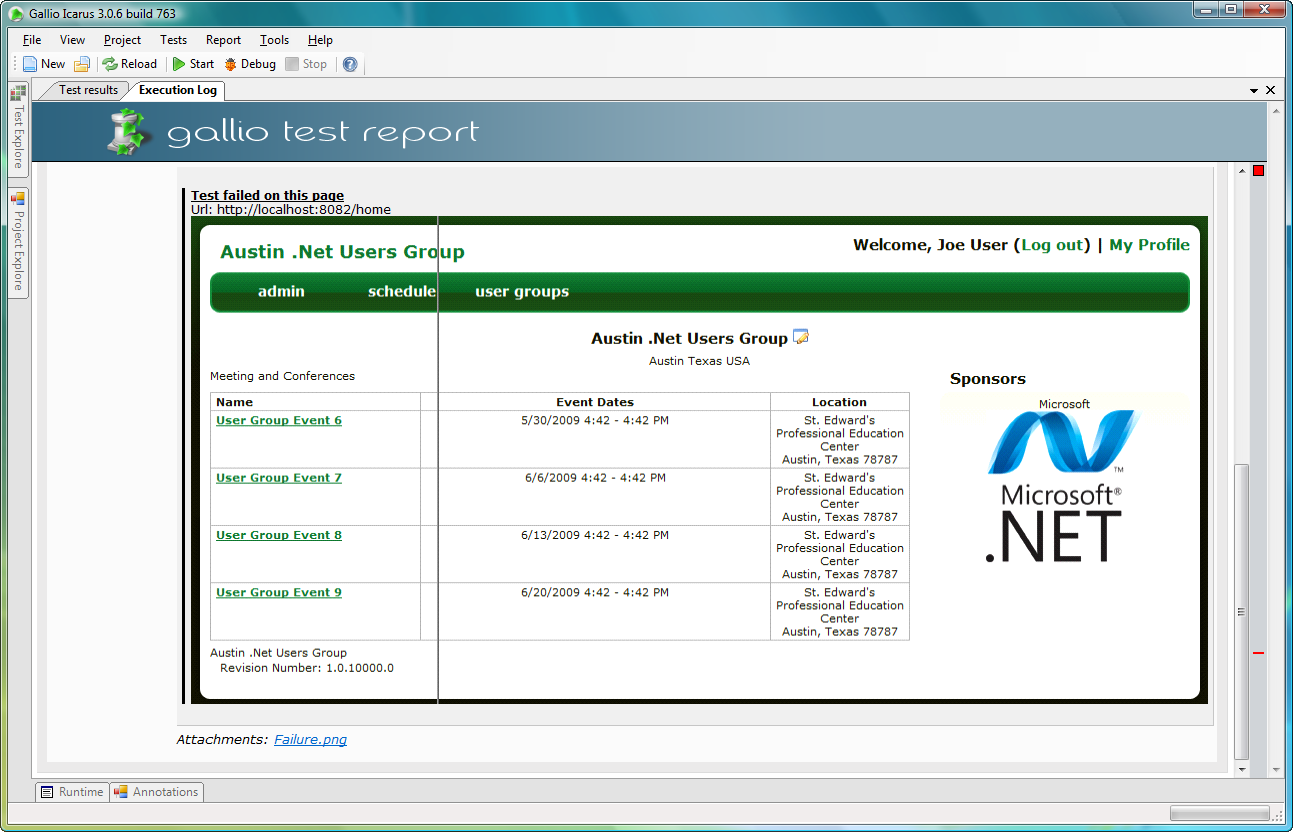


Figure 12.5 Our test report including a screenshot

Gallio is a powerful tool for creating UI tests when combined with WatiN. We can create a wrapper over the WatiN browser calls, which can be difficult to read, as well as more fluent calls that take advantage of strongly typed views, expressions, and Gallio’s test steps. With test steps and a simple wrapper, we can log all interaction with IE as a sort of test script, so that we can easily read exactly what our test performed, and exactly where it failed in the context of a user’s actions, instead of a stack trace. We might rather know that a test failed when the user clicked Submit Order, rather than receive a line number in a file. With Gallio and WatiN, this is possible. Tests that might take weeks to execute manually can finish in an hour.

UI tests are much, much slower than unit and integration tests, but they are vital in ensuring our application works end-to-end. Because of the speed of these tests, their use should be reserved for scenario-based tests, happy-path or black-box testing, and regression tests. Unless care is taken to ensure strongly typed tests and to avoid the magic strings we examined earlier, UI tests become quite brittle. It's worth noting that most applications are not easily testable without modifications. Just as we have to design our code for testability, we need to design our UI for testability. This might include putting IDs or special class names around certain data-driven elements, or sharing the view types with our UI tests to ensure that the exact same HTML element names are used for both HTML generation and UI testing. These changes don’t affect the end-user experience, and allow us as developers to focus on adding value, rather than fixing brittle tests.

12.5 Summary

In this chapter, we explored many of the extension points and major feature areas of ASP.NET MVC and discovered how best to take advantage of these areas in a maintainable manner. Although not every practice applies in every context, it is important to consider all the options available, and the benefits and tradeoffs of each before proceeding with a design. If you go down a path with filters and magic strings in ViewData, you might not like the end result. Instead, we can consider the long-term viability of each option and choose the most appropriate path for each situation. Some practices are strongly recommended for a maintainable and easily testable codebase, such as strongly typed views. Others, such as convention-based, REST-style action names are appropriate only in resource-centric applications.

Duplication is one of the biggest causes of development attrition, whether using ASP.NET MVC or another framework. The techniques used to remove duplication have changed from classic Web Forms, from custom model binders, to action filters and partials in our views. Although each of these extension points is powerful, none is appropriate in every context. We examined many of the options for eliminating duplication in our controllers and views, as well as elaborating on the right contexts for each of these options.

We focused on testing these extension points. Because these extension points can be executed on every srequest, it is vital to ensure that these extension points behave as desired. However, the true test of a working MVC application is using it in a browser. We finished our testing discussion by examining UI testing with WatiN and Gallio, taking advantage of features in both products to capture screenshots from failures and logging meaningful test messages. In the next chapter, we’ll examine a variety of real-world scenarios in the form of in-depth recipes.