20

Full System Testing

This chapter covers:

* Testing a web application with browser automation
* Examining simple, but brittle tests
* Building maintainable, testable navigation
* Leveraging strongly-typed views and expression-based helpers in tests

ASP.NET MVC ushered a new level of testability in .NET web applications. Typical Web Forms applications can have quite complex UI interaction in a code-behind. While testing a controller action is valuable, the controller action itself is only one piece of a rather large pipeline. Various extension points can be used, such as action filters, model binders, custom routes, action invokers, controller factories and so on. Our view can also contain complex rendering logic, unavailable in a normal controller action unit test. With all of these moving pieces, we need some sort of user interface testing to ensure that an application works in production as expected. The normal course of action is to design a set of manual tests in the form of test scripts and hope that our QA team executes them correctly. Many times, the execution of these tests is outsourced, increasing the cost of testing because of the increased burden on communication. Testing is manual because of the perceived cost of automation as well as experience of brittle user interface tests. However, this does not need to be the case. With the features in ASP.NET MVC 2, we can design maintainable, automated user interface tests.

20.1 Testing the User Interface Layer

In this book thus far, we examined many of the individual components and extension points 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
* Lightweight Test Automation Framework (http://aspnet.codeplex.com/wikipage?title=ASP.NET%20QA)

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. A basic, but useful scenario in our application is to test to see if we can edit product information. Our sample application allows the user to view and edit product details, a critical business feature. Testing manually, this would mean:

1. Navigating to the home page
2. Clicking the Products menu navigation, shown in figure 20.1

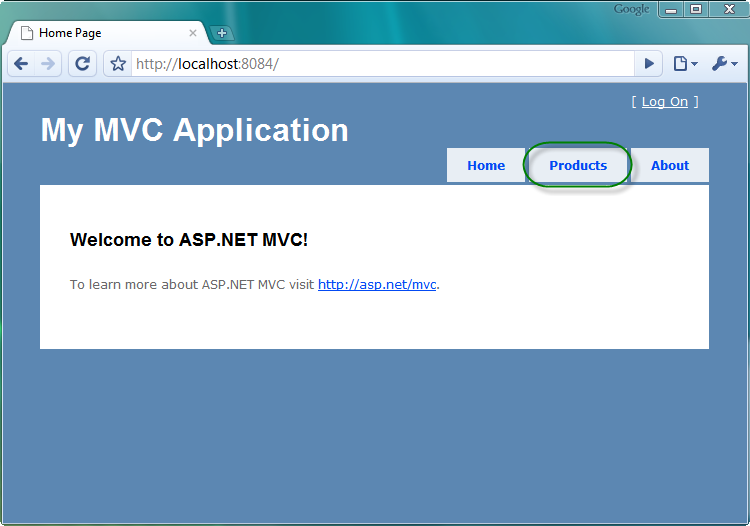


Figure 20.1 Clicking the Products link

1. Clicking the Edit link for one of the products listed, shown in figure 20.2

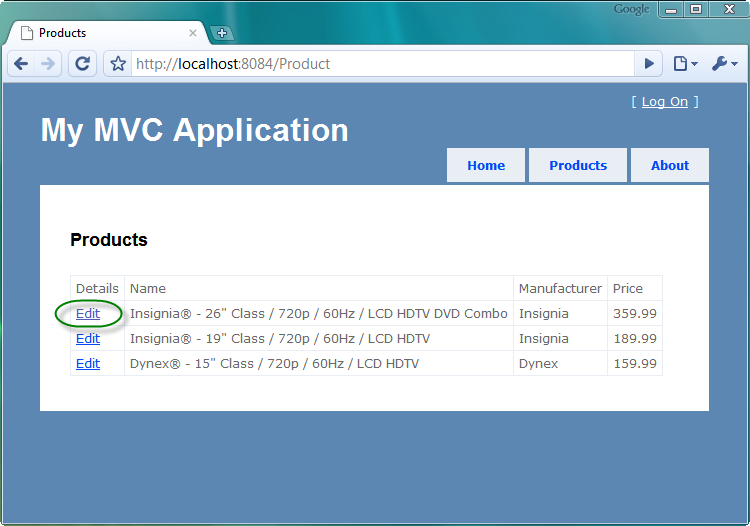


Figure 20.2 Clicking the Edit link for a product

1. Modifying the product information and clicking Save, shown in listing 20.3

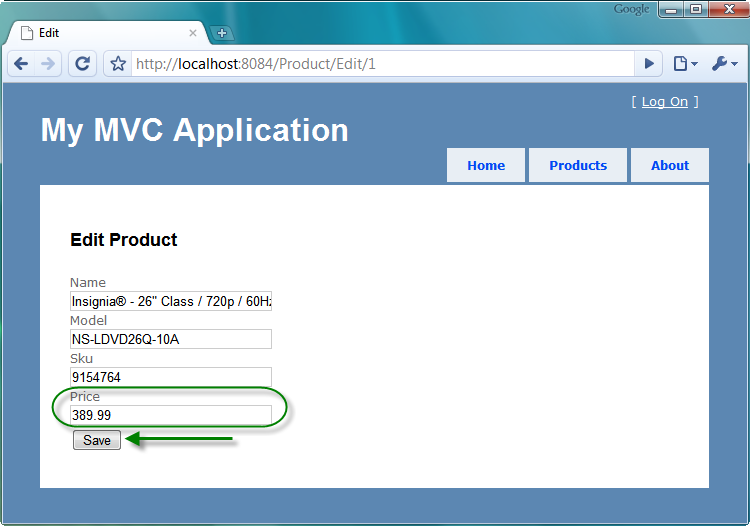


Figure 20.3 Modifying product information and saving

1. Checking that we were redirected back to the product listing page
2. Checking that the product information updated correctly, shown in figure 20.

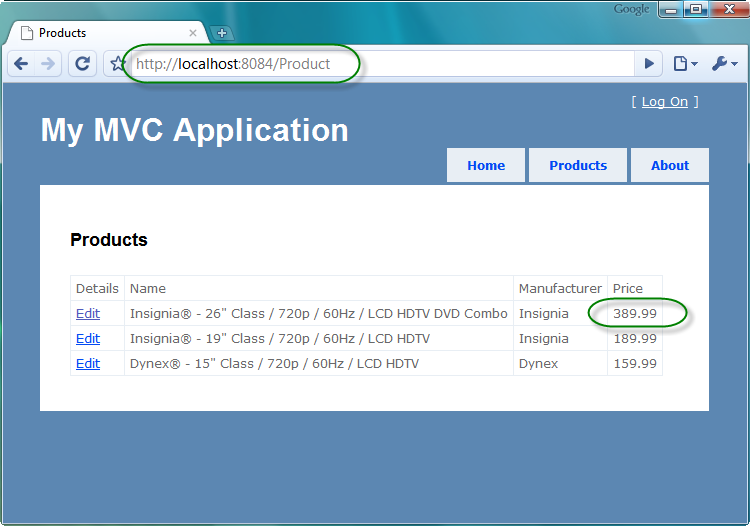


Figure 20.4 Verifying the correct landing page and changed information

Once we have our test scenario behavior described, we can author a test to execute this scenario. Our first pass at this UI test is in listing 20.1 below.

Listing 20.1 - First pass at our UI test

[TestFixture] #1

[ApartmentState(ApartmentState.STA)] #2

public class ProductEditTester

{

[Test] #3

public void Should\_update\_product\_price\_successfully()

{

using (var ie = new IE("http://localhost:8084/")) #4

{

ie.Link(Find.ByText("Products")).Click(); #5

ie.Link(Find.ByText("Edit")).Click(); #6

var priceField = ie.TextField(Find.ByName("Price")); #7

priceField.Value = "389.99"; #8

ie.Button(Find.ByValue("Save")).Click(); #9

ie.Url.ShouldEqual("http://localhost:8084/Product"); #10

ie.ContainsText("389.99").ShouldBeTrue(); #11

}

}

}

We first create a class and decorate the class with the TestFixtureAttribute (1). Like most automated testing frameworks in .NET, MbUnit requires you to decorate test classes with an attribute. MbUnit looks for these attributes to determine which classes to execute in its testing harness. Next, we decorate the test class with the ApartmentState (2) attribute. This attribute is necessary because WatiN uses COM to automate the IE browser window. Each test we author is a public void method decorated with the Test attribute (3). MbUnit will execute every method with the Test attribute and record the result.

With our test class and method in place, we now need to use WatiN to execute our test scenario. First, we instantiate a new IE object in a using block (4). When the IE object is instantiated, a browser window immediately launches and navigates to the URL specified in the constructor. We need to enclose the IE lifecycle in a using block to ensure that the COM resources WatiN uses are properly disposed of. The IE object is our main gateway to browser automation with WatiN.

To interact with the browser, the IE object exposes methods to find, examine and manipulate DOM elements. We use the Link method above (5) to find the Products link by its text, and then click it with the Click method. The Link method includes many overloads and we use the one to select based on a WatiN BaseConstraint object. The Find static class includes helper methods to build constraints which are used to filter the elements in the DOM. Once we click the Products link, we navigate to the first Edit link on the page and click it (6). After clicking this link, we are now on the edit screen for a single product.

We now need to find and fill in the input element for the price. Looking at the source, we can see that the input element has a Name attribute value of "Price". We search by name attribute to locate the correct Price input element (7). To modify the value of the element, as if we were typing in the value in a browser manually, we set the Value property to a new value (8). With the value changed, we can now find the Save button by name and click it (9).

If our save completes successfully, we should be redirected back to the products list page. If we encounter a validation error, we will stay on the product edit screen. In our scenario, we entered all valid data, so we check to make sure we are redirected back to the products list page (10). Finally, we can check that our product value is updated by searching for the price value on the page (11). When we execute this test, we will see our browser pop up and perform all of the interactive tasks that we would normally accomplish manually, but in an automated fashion instead. It can be quite impressive to see our test running and passing successfully. A suite of manual tests is slow and error-prone, and automation eliminates the human error of manual site manipulation.

Unfortunately, our confidence will wane as our page starts to change. The test created in this section functions well, but is quite brittle in the face of change. The breakages from change in our application include:

* Product link text could change
* Edit link text could change
* Test picks the first product, product list could change
* Input element name could change
* Save button text could change
* Landing URL could change, from the port number, controller name or action name
* Price text could be anywhere, including on an existing product

These are all legitimate changes that normally occur over the lifetime of our project. However, none of these changes should result in the breakage of our test. Ideally, our test should fail because of an assertion failure, not in the setup or execution phases. The solution for brittle tests at any layer is to design for testability. In our first example of the test scenario, we treated our application as a black box. The test only used the final rendered HTML to build an interaction with the application. Instead of treating our application as a black box, we can design our user interface for testability for stable, valuable user interface tests. In the next section, we will look at maintainable navigation elements to interact with our site.

20.2 Maintainable navigation

Our original test navigated to a very specific URL inside the test. While this might not change, we do not want each test to duplicate the starting URL. Things like port numbers and home page URLs can change over time. Instead, we can create a base test class that extracts the common setup and cleanup of our IE browser object, shown in listing 20.2.

Listing 20.2 Creating our base test class

[TestFixture]

[ApartmentState(ApartmentState.STA)]

public class WebTestBase

{

private IE \_ie;

[SetUp]

public virtual void SetUp() #1

{

\_ie = new IE("http://localhost:8084/");

}

[TearDown]

public virtual void TearDown() #2

{

if (\_ie != null)

{

\_ie.Dispose();

\_ie = null;

}

}

protected IE Browser #3

{

get { return \_ie; }

}

Our new base test class creates the IE browser object with the correct starting URL (1). If we need different starting URLs, we would still want to eliminate any duplication of the host name and port number. We create a setup method that executes before every test, storing the created IE object in a local field. At the conclusion of every test, our TearDown method executes (2). The original test wrapped the IE object's lifetime in a using block. Since the removal of the using block does not eliminate the need for our test to dispose of the IE object, we need to manually dispose of our browser object in the TearDown method. Finally, to allow derived test classes to have access to our created IE object, we expose this field with a protected property (3). With this change, our UI test already becomes easier to read, shown in listing 20.3.

Listing 20.3 The test modified to use the base test class and Browser property

[TestFixture]

[ApartmentState(ApartmentState.STA)]

public class ProductEditTester : WebTestBase

{

[Test]

public void Should\_update\_product\_price\_successfully()

{

Browser.Link(Find.ByText("Products")).Click();

Browser.Link(Find.ByText("Edit")).Click();

var priceField = Browser.TextField(Find.ByName("Price"));

priceField.Value = "389.99";

Browser.Button(Find.ByValue("Save")).Click();

Browser.Url.ShouldEqual("http://localhost:8084/Product");

Browser.ContainsText("389.99").ShouldBeTrue();

}

}

With very exceptions, each of our user interface tests will need to navigate our site by clicking various links and buttons. We could manually navigate through URLs directly, but that would bypass the normal navigation the end user would use. In our original test, we navigated links strictly by the raw text shown to the end user. However, this text can change fairly easily. Our customers might want to change the "Products" link text to "Catalog", or the "Edit" link to "Modify". Each of these changes would break our test, but they do not have to. We can embed extra information in our HTML to help our test to navigate the correct link by its semantic meaning, instead of text shown to the user. In many sites, text shown to end users is data driven through a database or content-management system (CMS). This makes navigation by raw link text even more difficult and brittle.

The anchor tag already includes a mechanism to describe the relationship of the linked document to the current document, the rel attribute. We can take advantage of this informative, but non-visual attribute to precisely describe our link. If there are two links with the text "Products", we can distinguish them with the rel attribute. However, we do not want to fall into the same trap of searching for the final, rendered HTML. We can instead provide a shared constant for this link, shown in listing 20.4 below.

Listing 20.4 Adding the rel attribute to the products link

<ul id="menu">

<li><%= Html.ActionLink("Home", "Index", "Home")%></li>

<li><%= Html.ActionLink("Products", "Index", "Product",

null, new { rel = LocalSiteMap.Nav.Products })%></li> #1

<li><%= Html.ActionLink("About", "About", "Home")%></li>

</ul>

The LocalSiteMap class is a static class exposing a simple navigational structure through constants, shown in listing 20.5.

Listing 20.5 The LocalSiteMap class exposing a simplified navigational structure

public static class LocalSiteMap

{

public static class Nav

{

public static readonly string Products = "products";

}

Since we do not want to fall into the same trap of hard-coding "rel" values in our test and view, we create a simple constant that can be shared between our test code and view code. This allows the "rel" value to change without breaking our test, shown in listing 20.6.

Listing 20.6 The UI test using a helper method to navigate links

[TestFixture]

[ApartmentState(ApartmentState.STA)]

public class ProductEditTester : WebTestBase

{

[Test]

public void Should\_update\_product\_price\_successfully()

{

NavigateLink(LocalSiteMap.Nav.Products);

The NavigateLink method is a helper method wrapping the work of finding a link by the rel attribute and clicking it, in listing 20.7.

Listing 20.7 The NavigateLink method in our WebTestBase class

protected virtual void NavigateLink(string rel)

{

var link = Browser.Link(Find.By("rel", rel));

link.Click();

}

By encapsulating the different calls to the IE browser object in more meaningful method names, we make our UI test easier to read, author and understand. Since both our view and our test share the same abstraction of representing navigational structure, we strengthen the bond between code and test. This strengthening lessens the chance of our UI tests breaking because of orthogonal changes that should not affect the semantic behavior of our tests. Since our test is merely attempting to follow the product link, it should not fail if the semantics of the product link do not change. In the next few sections, we will continue this theme of enforcing a connection between test and UI code, moving away from black-box testing.

20.3 Interacting with forms

In this book, we eschewed the value of embracing strongly-typed views and expression-based HTML helpers. This allowed us to take advantage of modern refactoring tools that can update our view code automatically in the case of member name changes. Why then revert back to hard-coded magic strings in our UI tests? For example, our edit view already takes advantage of strongly-typed views in displaying the edit page, shown in listing 20.8.

Listing 20.8 The strongly-typed view using editor templates

<%@ Page Title="" Language="C#"

MasterPageFile="~/Views/Shared/Site.Master"

Inherits="System.Web.Mvc.ViewPage<ProductForm>" %> #1

<%@ Import Namespace="UITesting.Models" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

Edit

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<h2>Edit Product</h2>

<form action="<%= Url.Action("Save") %>" method="post">

<%= Html.EditorForModel() %> #2

<input type="submit" value="Save" />

</form>

</asp:Content>

Our edit view is a strongly-typed view for a ProductForm view model type (1). We use the ASP.NET MVC 2 feature of editor templates (2) to remove the need of hand-coding the individual input and label elements. The EditorForModel method also lets us change the name of any of our ProductForm members without breaking our view or controller action. In our UI test, we can take advantage of strongly-typed views by using a similar approach of expression-based helpers, shown in listing 20.9 below.

Listing 20.9 Using a fluent API and expression-based syntax to fill out forms

[Test]

public void Should\_update\_product\_price\_successfully()

{

NavigateLink(LocalSiteMap.Nav.Products);

Browser.Link(Find.ByText("Edit")).Click();

ForForm<ProductForm>() #1

.WithTextBox(form => form.Price, 389.99m) #2

.Save(); #3

This simple fluent interface starts with specifying the ViewModel type by calling the ForForm method (1). The ForForm method builds a FluentForm object, which we will examine shortly. Next, the WithTextBox method chains off of the ForForm method and accepts an expression to a model member property, as well as a value to fill in the input element. Finally, the Save method clicks the save button on the form. Let's examine what happens behind the scenes, first with the ForForm method call, shown in listing 20.10.

Listing 20.10 The ForForm method on the WebTestBase class

protected FluentForm<TForm> ForForm<TForm>() #1

{

return new FluentForm<TForm>(Browser); #2

}

The ForForm method accepts a single generic parameter, the form type (1). It returns a FluentForm object, which wraps a set of helper methods designed for interacting with a strongly-typed view. The ForForm method instantiates a new FluentForm object (2), passing the IE object.

Listing 20.11 The FluentForm class and constructor

public class FluentForm<TForm>

{

private readonly IE \_browser;

public FluentForm(IE browser) #1

{

\_browser = browser; #2

}

The FluentForm constructor, shown in listing 20.11, accepts an IE object (1) and sets it to a private field (2) for subsequent interactions. The next method called in listing 20.9 is the WithTextBox method, shown in listing 20.12.

Listing 20.12 The expression-based WithTextBox method

public FluentForm<TForm> WithTextBox<TField>( #1

Expression<Func<TForm, TField>> field, #2

TField value) #3

{

var name = UINameHelper.BuildNameFrom(field); #4

\_browser.TextField(Find.ByName(name)).TypeText(value.ToString()); #5

return this; #6

}

Our FluentForm method contains another generic type parameter (1), TField, which helps with compile-time checking of form values. The first method parameter is an expression of a function that accepts an object of the TForm form type and returns a TField type (2). This is a common pattern for accomplishing strongly-typed reflection, and accepting an expression that is used to navigate to a type's members. The second parameter (3), of type TField, will be the value actually set on the input element. To correctly locate the input element based on the expression given, we use a helper class (4) to build the UI element name from an expression. For our original example, the code snippet form => form.Price will result in an input element with a name value of "Price". With the correct, compile-safe input element name, we use the IE object to locate the input element by name and type the value supplied (5). Finally, to enable chaining of multiple input element fields, we return the FluentForm object itself (6).

The benefits of this approach are the same as they are for strongly-typed views and expression-based HTML generators. We can refactor our model objects with the assurance that our views will stay up to date with any changes. By sharing this technique in our UI tests, our UI tests will no longer break if our model changes. If we remove a member from our view model, in the case it is no longer displayed, our UI test will no longer compile. This early feedback that something has changed is much easier to detect and fix than waiting for a failing test.

The code to turn an expression into an HTML element name is quite complex, and can be found in the full sample online. After we have the input element populated, we need to click the save button in our Save method, shown in listing 20.13.

Listing 20.13 The FluentForm Save method

public void Save()

{

\_browser.Forms[0].Submit(); #1

}

While the Save method above only submits the first form found (1), we can use a variety of other methods if there is more than one form on the page. Similar to our technique for locating links, we can add contextual information to the form's class attribute if need be. For our scenario, we only encounter one form per page, so submitting the first form found will suffice. Now that we have our form submitting correctly, and in a maintainable fashion, we now need to assert the results of the form post.

20.4 Asserting results

When it comes to making sure our application works as expected, we have several general categories of assertions. We typically ensure our application redirected to the right page and shows the right information. In more advanced scenarios, we might assert on specific styling information that would further relate information to the end user. In our original test, we asserted a correct redirect by checking a hard-coded URL. However, this URL can also change over time. We might change the port number, host name, or even controller name. Instead, we want to build some other representation of a specific page. Similar to representing links in our site, we can build an object matching the structure of our site. The final trick will be to include something in our HTML indicating which page is shown.

While we could do this by attaching IDs to the body element, in practice this becomes quite ugly as this tag is typically in a master page. Another tactic is to create a well-known input element, excluded from any form, shown in listing 20.14.

Listing 20.14 Providing a page indicator in our markup

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<input type="hidden" name="pageId"

value="<%= LocalSiteMap.Screen.Product.Index %>" /> #1

<h2>Products</h2>

In the example above, we include a well-known hidden input element with a name of "pageId" and a value referencing our site structure as a constant. The navigational object structure is designed to be easily recognizable. The above example indicates the product index page. The actual value is a simple string, shown in listing 20.15 below.

Listing 20.15 Site structure in a well-formed object model

public static class LocalSiteMap

{

public static class Screen

{

public static class Product

{

public static readonly string Index = "productIndex";

}

}

Our site structure is exposed as a hierarchical model, finally exposing a constant value. It is this constant value that is used in the hidden input element. With this input element in place, we can now assert our page simply by looking for this element and its value, shown in listing 20.16.

Listing 20.16 Asserting for a specific page

[Test]

public void Should\_update\_product\_price\_successfully()

{

NavigateLink(LocalSiteMap.Nav.Products);

Browser.Link(Find.ByText("Edit")).Click();

ForForm<ProductForm>()

.WithTextBox(form => form.Price, 389.99m)

.Save();

CurrentPageShouldBe(LocalSiteMap.Screen.Product.Index); #1

The CurrentPageShouldBe method encapsulates the work of locating the well-known input element, and asserting its value. We pass in the same constant value (1) to assert against as was used to generate the original HTML. Again, we share information between our view and test to ensure that our tests do not become brittle. The CurrentPageShouldBe method, shown in listing 20.17 below, is defined on the base WebTestBase class so that all UI tests can use this method.

Listing 20.17 The CurrentPageShouldBe method

protected void CurrentPageShouldBe(string pageId)

{

Browser.TextField(Find.ByName("pageId")).Value.ShouldEqual(pageId);

}

Finally, we need to assert that our application actually changed the price value correctly. This will require some additional work in our view, as it is currently quite difficult to locate a specific data-bound HTML element. The original test merely searched for the price text anywhere in the page. But this means that our test could pass even if the price wasn't updated as the text for the price might show up for something else completely unrelated, such as another product, the version text at the bottom of the screen, the shopping cart total and so on.

Instead, we need to use a similar tactic of displaying our information as we did for rendering our edit templates. We will use the expression-based display templates, as shown in listing 20.18 below.

Listing 20.18 Using expression-based display templates

<table>

<thead>

<tr>

<td>Details</td>

<td>Name</td>

<td>Manufacturer</td>

<td>Price</td>

</tr>

</thead>

<tbody>

<% var i = 0; %>

<% foreach (var product in products) { %>

<tr>

<td><%= Html.ActionLink("Edit", "Edit", new { id = product.Id }) %></td>

<td><%= Html.DisplayFor(m => m[i].Name) %></td> #1

<td><%= Html.DisplayFor(m => m[i].ManufacturerName)%></td>

<td><%= Html.DisplayFor(m => m[i].Price)%></td>

</tr>

<% i++; } %>

</tbody>

</table>

We need to utilize the full expression, including the array index, for use with the expression-based display templates (1). Out of the box, the display templates for strings are just the string value itself. We want to decorate this string with identifying information, in the form a span tag. This is accomplished quite easily by overriding the string display template. First, we need to add a new string template file in our shared display templates folder, shown in figure 20.5 below.

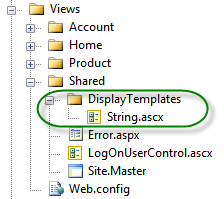


Figure 20.5 Adding the new string template

The string.ascx template is modified below in listing 20.19 to include a span tag with an ID derived using the TemplateInfo.GetFullHtmlFieldId method.

Listing 20.19 The updated string display template

<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>

<span id="<%= ViewData.TemplateInfo.GetFullHtmlFieldId(null) %>"> #1

<%= Html.Encode(ViewData.TemplateInfo.FormattedModelValue) %> #2

</span>

The span tag wraps the entire value displayed, with a well-formed ID derived from the expression originally used to display this template. In the case above, the original expression m => m[i].Name would result in a runtime span ID of "[0]\_Name". Because the array index is included in the span ID, we can distinguish this specific model value apart from any other product shown on the screen. We do not need to search for items matching generic values; we can navigate directly to the correct rendered model value.

In our test, we build a similar abstraction as our FluentForm object, but now we create a FluentPage object for asserting information displayed on our screen. In listing 20.20, our test uses the ForPage and FindText methods to assert a specific product price value.

Listing 20.20 The final test code using expression-based display value assertions

[Test]

public void Should\_update\_product\_price\_successfully()

{

NavigateLink(LocalSiteMap.Nav.Products);

Browser.Link(Find.ByText("Edit")).Click();

ForForm<ProductForm>()

.WithTextBox(form => form.Price, 389.99m)

.Save();

CurrentPageShouldBe(LocalSiteMap.Screen.Product.Index);

ForPage<ProductListModel[]>() #1

.FindText(products => products[0].Price, "389.99"); #2

}

The ForPage method takes a single generic argument, specifying the ViewModel type for the particular page being viewed at the moment. Next, we find a specific text value with the FindText method (2). This method accepts an expression for a specific model value and the value to assert. We look for the first product's price, and assert that its value is the same value supplied in our earlier form submission. The ForPage method builds a FluentPage object, shown in listing 20.21 below.

Listing 20.21 The FluentPage class

public class FluentPage<TModel> #1

{

private readonly IE \_browser;

public FluentPage(IE browser) #2

{

\_browser = browser; #3

}

public FluentPage<TModel> FindText<TField>( #4

Expression<Func<TModel, TField>> field, #5

TField value) #6 {

var name = UINameHelper.BuildIdFrom(field); #7

var span = \_browser.Span(Find.ById(name)); #8

span.Text.ShouldEqual(value.ToString()); #9

return this; #10

}

}

The FluentPage class has a single generic parameter (1), TModel, for the page's view model type. The FluentPage constructor accepts an IE object (2) and stores it in a private field (3). Next, we define the FindText method (4) in a similar manner as our WithTextBox method earlier. It contains a generic parameter against the field type, and accepts a single expression to represent accepting a form object and returning a form member (5). Finally, the FindText method accepts the expected value (6). In the body of the method, we first need to build the ID from the expression given (7). Next, we find the span element using the ID built from the expression (8). The span object contains a Text property, representing the contents of the span tag. We assert that the span contents match the value supplied in the Fluent Page method (9). Finally, to allow for multiple assertions using method chaining, we return the FluentPage object itself (10). With our test now strongly-typed, expression based and sharing knowledge with our views, our tests are much less likely to break. In practice, we found that tests built with style now break because of our application's behavior changing, rather than just the rendered HTML.

20.5 Summary

In this chapter, we started with an automated user interface test that passed, but could become quite brittle. If we designed an entire suite this way, the burden of getting value out of our tests would grow until we would likely abandon the entire effort, or invest expensive resources into keeping the tests passing. At this point, our automated tests would cease to provide the value the promise of automated testing usually brings. The answer to this problem is design for testability in our user interface. This started with designing a maintainable navigation structure. We continued with strongly-typed views and expression-based HTML helpers. The same mechanism for designing views in this manner can then be applied to our UI tests, so that our UI tests become only as brittle as our views. Finally, we showed ways to assert outcomes in our browser, including redirecting to the right page and displaying the right information. In all of our improvements, the common theme was designing our UI for testability, by sharing design information that can be used in our tests. As we encounter new scenarios, we simply need to be wary of testing strictly off of the rendered HTML and instead investigate how we can share knowledge between our views and our tests.