20

Full System Testing

This chapter covers:

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
3. Clicking the Edit link for one of the products listed
4. Modifying the product information and clicking Save
5. Checking that we were redirected back to the product listing page
6. Checking that the product information updated correctly

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();

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

priceField.Value = "389.99";

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

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

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

}

}

}

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.

20.1.1 Maintainable navigation

20.1.2 Interacting with forms

20.1.3 Asserting results

20.1.4 Dealing with AJAX

20.2 Subcutaneous Testing

20.2.1 Contrasting with UI Testing

20.3 Summary