Extracted from:

Web Development Recipes

This PDF file contains pages extracted from *Web Development Recipes*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2010 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Web Development Recipes

Brian P. Hogan,
Chris Warren,
Mike Weber,
Chris Johnson,
and Aaron Godin

edited by Susannah Davidson Pfalzer

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at http://pragprog.com.

The team that produced this book includes:

Susannah Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Copyright © 2012 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-93435-683-8
Printed on acid-free paper.
Book version: P1.0—January 2012

Problem

Testing is a hard and tedious process. As websites become more complex, it becomes more important to have tests that are repeatable and consistent. Without automated testing, our only chance at having a consistent working website was to have a top-notch QA person that worked long hours and had very long checklists. That process could be painfully slow. We need to speed up the testing process and create tests we can run on-demand so that we can verify things work the way we want today, as well as several months from now when we start adding new features.

Ingredients

- Firefox[7]
- Selenium IDE[8]
- QEDServer (for our test server); see *QEDServer*, on page ?

Solution

We can use automated tools to test our web projects in addition to manual testing. The Selenium IDE plug-in for Firefox lets us build tests in a graphical environment by recording our actions as we use a website. As we move through a site, we can create *assertions*, or little tests that ensure that certain things exist on the pages. We can then play them back any time we want, creating a set of automated, repeatable tests.

Our development team has built a product management website, and our boss wants some safeguards in place to make sure this will always work. The development team has added some unit testing to their business logic underneath, but we're tasked with building some automated tests for the user interface. Automated testing will give both the development team and us peace of mind if we make changes to the UI down the road.

Setting Up Our Test Environment

First, we need to install the Firefox web browser. Go to the Firefox website and follow the instructions for your operating system.

Once we have Firefox working, we need to get the Selenium IDE installed. Open Firefox, visit the Selenium website,[9] and download the latest version.[10]

7. http://getfirefox.com
8. http://seleniumhq.org/download/
9. http://seleniumhq.org/download/
10. If you have Firefox 4, you may need to install the Add-On Compatibility Reporter Extension version 0.8.2.

With the tools installed, let's write our first test.

Creating Our First Test

We'll create our test by recording our movements with the Selenium IDE against our test server, which we'll run on our own machine using QEDServer. Start QEDServer and then launch Firefox. Go to http://localhost:8080 to bring up the test server, where you'll see an interface like the one in Figure 52, *Our home page screen*, on page 7.

Since this is a product management application, we'll start off with a test to make sure we always have the "Manage products" link on the home page and that the link goes where we expect it to go.

Open up the Selenium IDE by selecting it from the Tools menu in Firefox. To start recording, we need to make sure the Record button is active. Then, in the browser, we click the "Manage products" link. As we click the link, we'll see some items begin to show up in Selenium IDE, just like in Figure 53, *Our first test with the Selenium IDE*, on page 7. At the top, the Base URL is now set to `http://localhost:8080`, and then we see one of the most useful commands in the Selenium IDE: the `clickAndWait()` method. When we use web applications, we spend a lot of time clicking on links or buttons and waiting for pages to load. That is exactly what this command does. Every time we click a link, the Selenium IDE adds this method to our test along with some text that identifies the link. When we play the test back, it uses this method and the associated link text to drive the browser.

The Selenium IDE shows us the three parts of a Selenium test action. The first is Command, which is the action that Selenium is performing. The second is Target, which is the item that Selenium is performing the action on. The third is Value, which we'll use to set a value for fields that take inputs, such as when we're filling out a text box or selecting a radio button.

A powerful part of Selenium is its locator functions. We can use these to find an element on the page not only by its `id` but also via the DOM, an XPath query, a CSS selector, or even plain text. When we clicked "Manage products," the target we used is `link= Manage products`. The `link=` is the selector that allows us to choose a block of text to perform an action on. One thing we should keep in mind is that locators default to looking for an `ID` first followed by that string of text. Identifying elements for testing with IDs is a great way to speed up your tests and improve accuracy, but it can make tests harder to read.

Now that we have an understanding of locators, let's look at what the commands do. Commands are the actions that Selenium performs when we run

• Click HERE to purchase this book now. discuss

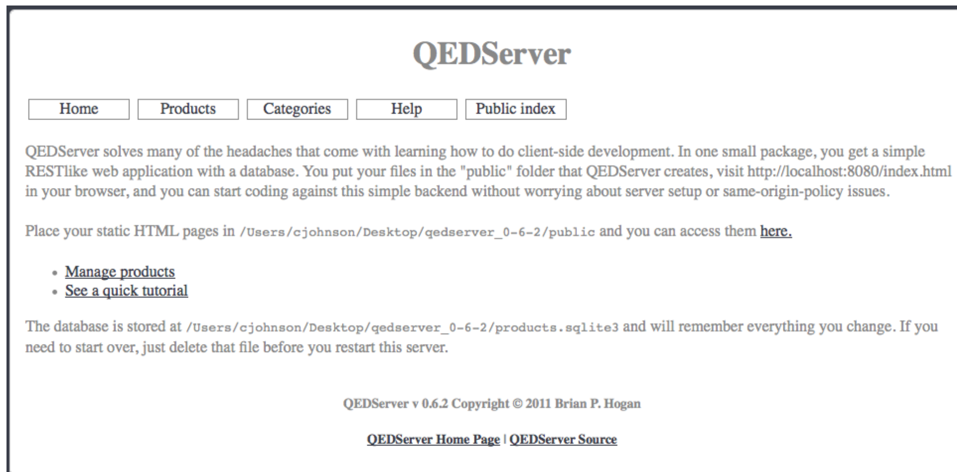

Figure 52—Our home page screen

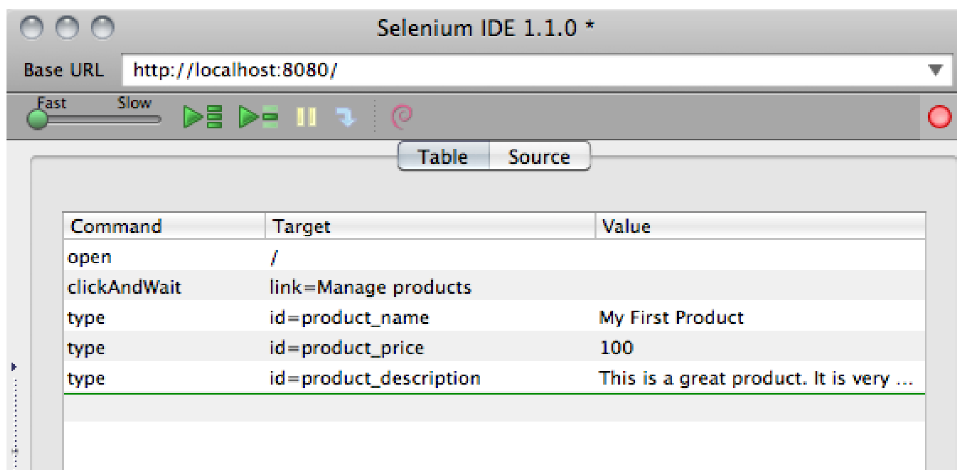

Figure 53—Our first test with the Selenium IDE

a test. Selenium can do anything a human would do, with one small exception—it can't upload a file without some significant modifications. The ability to manipulate a browser the way a human would allows us to simulate human interaction, giving our tests the ability to flex our code realistically.

Let's test that once we click “Manage products” we are taken to a page where “Products” is present. First we need to click the “Manage products” link. Next we want to make sure the word “Products” is on the screen. To add a test for

that, locate Products on the web page and right-click it. Choose the command verifyTextPresent from the context menu. We could also do this by using the Selenium IDE and clicking in the whitespace just below the clickAndWait() command and using the form fields to choose our command, target, and value, but the Selenium IDE adds some test helpers to the context menu, which makes the process much faster.

We can save this test by choosing Save Test Case from the Selenium IDE's File menu. We can then run the test by clicking the play button below the Base URL window. As the test runs, the browser moves through our pages, and the background color for each step changes to green as it passes. If a step fails, it will turn red and will also show some bold red text in the log window below that with descriptions of what went wrong. This visual cue lets us know something went wrong so we can address it.

Creating an Advanced Test

We want to make sure that our product management application functions, and we can create a new product and delete a product. We also want to make sure we can view the details of a product. This is a multistep process; let's use Selenium to automate it.

Let's go back to the home page at http://localhost:8080 and start up the Selenium IDE. Click the "Manage products" link and wait for the page to load. Then select the "New Product" text, right-click it, and select "verifyTextPresent New Product." Next, leave the form blank and click the Add Product button. Our application requires that we fill in at least some of the product details, so we now have a form that did not submit, along with an error message on the screen.

Let's make this part of our test. Right-click the "The product was not saved" error message, and use the verifyTextPresent command to add an assertion that verifies that the error message show up on the Products page.

Now that we've shown that our error message works, we can now fill out all of the information in the form and submit it. The Selenium IDE adds a row to our test for each field we fill out. It also shows the value we typed in.

When we submit the form this time, it takes us back to the products page where we'll see the message "Created." We can use the verifyTextPresent() command again to make sure this text is displayed.

Now we have a feature-rich example that we can save and run later. If anyone changes the site, we'll know what's broken, simply by replaying the test.

• Click HERE to purchase this book now. discuss

Further Exploration

Now that we have test coverage, we can take this to the next level by automating our entire test suite. We currently have to run each test individually by loading it into the Selenium IDE, and this breaks down when we have a lot of tests. You'll want to investigate Selenium Remote Control and Selenium Grid,[11] which let you build automated test suites that run against multiple browsers.

And while Selenium IDE is primarily a testing tool, you could use it as an automation tool as well. For example, if you have a process that has a less-than-friendly user interface, such as a time-tracking system or a repetitive and clunky management console, you might try using Selenium IDE to save you some keystrokes and mouse clicks.

Also See

- Recipe 34, *Cucumber-Driven Selenium Testing*, on page ?
- Recipe 35, *Testing JavaScript with Jasmine*, on page ?

11. http://selenium-grid.seleniumhq.org/

• Click HERE to purchase this book now. discuss