3. Getting our feet wet with Watir

Our first script

As we saw in the first chapter, Cucumber uses other ruby gems to interact with the application we wish to test. One way to control a browser and thereby test a web application is by using a ruby gem named *Watir*. This chapter will introduce us to *Watir* and provide enough experience with this gem to allow us to test basic web applications.

Before we start writing scripts we need to create a directory where we will keep them. If you are using *RubyMine* begin by creating a new project. You do this by selecting *New Project* from the *File* menu. Name the project `learn_watir`. If you are not using *RubyMine* just create a directory named `learn_watir`.

Our first example will be very simple. We will just open a browser and go to the Apple website. Start by creating a new file named `first_script.rb` in the `learn_watir` directory. If you are using *RubyMine* you can right-click on the *learn_watir* project name and select *New -> File* and name the file `first_script.rb`. Once you have created the file you may add the following:

```
require "rubygems"
require "watir"
browser = Watir::IE.new
browser.goto "http://www.apple.com"
```

The first two lines of the script inform ruby of the libraries that need to be loaded in order to run the script. The first line loads the rubygems library which provides the ability to load gems. The second line loads the Watir gem.

The third line is where things get interesting. On this line we are creating a new Watir Internet Explorer object and giving it the name `browser`. This is a common pattern we will see throughout our scripts. When we create the new browser object it opens the Browser application on our computer.

The final line of the script is where we inform the browser to go to the Apple website. The way we do this is by sending a message to the `browser` object. That message is `goto`. As you might imagine, the remainder of the line is passed along with the message informing the browser which URL to go to.

This script will only run on a Windows platform since it requires Internet Explorer. Don't worry if you are not using Windows. Just move on to the next script and you will get your satisfaction. If you are on Windows go ahead and run the script. If you are using *RubyMine* you can right click on the file and select the *Run "first_script"* option near the bottom of the menu. If you are using the command-line then you can execute the

following command:

```
ruby first_script.rb
```

You should see the browser open and go to the Apple website. We have just created our first script.

Let's create a slight modification to this script. The following script will run the same test in Firefox if you installed the jssh plugin as described in the previous chapter.

```
require "rubygems"
require "firewatir"
browser = FireWatir::Firefox.new
browser.goto "http://www.apple.com"
```

If you are on OS X and have installed the safariwatir gem you can also write the following script:

```
require "rubygems"
require "safariwatir"
browser = Watir::Safari.new
browser.goto "http://www.apple.com"
```

You will notice that the last line in each script is identical. This is part of the power of the Watir gems. For the most part when you write a script that works with Firefox it will also work with Internet Explorer. There are some differences but I will show you how to minimize the differences as we encounter them.

Our test application

The source code distributed with this book has a couple of example applications that we will be writing scripts against. If you did not download the source you can find it at this url: REMOVED until I have the new example ready. Please download and unzip the file to a convenient location.
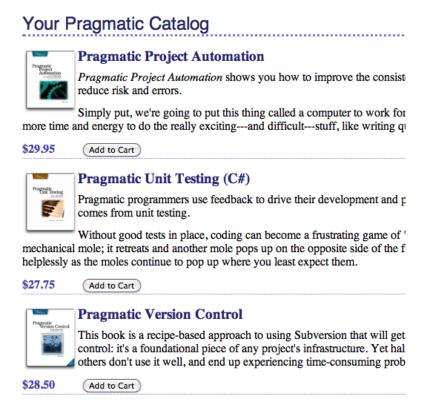
Within the source code distribution you will find a directory named depot. Please open a terminal or command window and change to that directory. The directory contains a sample book store application. You may start the application by executing the following command from your command window if you are on Windows:

```
ruby script\server
```

or the following from the terminal window if you are on OS X or Linux:

```
script/server
```

Once the server is started the application can be found at http://localhost:3000/store.

Go ahead and take the time to explore the depot application. Purchase a book and check out.

Getting started with depot

Now it is time to write our first script for the depot application. This script is going to be incredibly similar to the first script we wrote. If you have the previous script open go ahead and shut it and try to do this exercise from memory. If you are unsure of what to do please refer to the *Watir Quick Reference* appendix. The following list details the steps you script must complete.

1. Open the browser
2. Go to the depot store page
3. Wait 5 seconds
4. Close the browser

Waiting for five seconds is accomplished by the following call:

```
sleep 5
```

Once you have the script ready go ahead and run it. This script doesn't do very much at this time. I think it is time to change that. But first you need to learn how to work with elements on pages.

Finding and interacting with elements on a page

Open your browser and go to the depot application. Purchase a book and continue to the checkout page. Once you are on the checkout page start the developer tools for the browser you are using and press the "select element" button.

Now move your cursor around the page. You should see a blue box that surrounds elements as you hover over them. Select the "Place Order" button. You will be able to see details about the element. On Firefox using FireBug you should see the following:

Watir has a clean consistent way of locating elements. For our purposes we will always use the following three step pattern. Imagine we have created our browser object (like the example above) and are on the checkout page for our application. In order to press the "Place Order" button we would type the following:

```
browser.button(:value => "Place Order").click
```

First we must send a message to the `browser` object telling it what type of element we wish to interact with. *Watir* contains methods for nearly all of the items you will find on a typical web page. In this case we are calling the `button` method.

Now that we have indicated what we want to interact with we need to specify how to find it on the page. We do this by identifying a characteristic of the element we are trying to use. *Watir* supports finding elements by many identifiers. You can refer to the appendix to see a table of the characteristics that can be used. In this case we want to find the button that has a `value` of `Place Order`. We got that information from the developer tool when we selected the button. For a button, the value is the text that is displayed on the button. If there is more than one element matching the characteristic, *Watir* will

4

return the first one on the page by default. To select the second button on the page with a label of "Search" we could use `button(:value => "Search", :index => 2)`.

Finally we need to do something to the element that we have located. In this case we are sending the `click` message to the button.

Purchasing a book

Let's goo ahead and modify the previous script so that it completes the purchase of a book. This means you will add a book to the cart and then complete the checkout page. Again, use the browser developer tools to identify the elements you want to interact with and the *Watir Quick Reference* at the back of the book to learn how. Try to complete the script without looking at the finished script below.

The finished script

Here is an example of the finished script using Firefox.

```
require "rubygems"
require "firewatir"
browser = FireWatir::Firefox.new

browser.goto "http://localhost:3000/store"
browser.button(:value => "Add to Cart").click
browser.link(:text => "Checkout").click
browser.text_field(:id => "order_name").set("Cheezy")
browser.text_field(:id => "order_address").set("123 Main St.")
browser.text_field(:id => "order_email").set("cheezy@foo.com")
browser.select_list(:id => "order_pay_type").select("Check")
browser.button(:value => "Place Order").click
```

The first three lines sets us up to use Firefox. This should look very familiar to you as we have seen it several times already. You may change these lines to use the browser of your choice.

After that we go to the store page and click the "Add to Cart" button. Any time we are locating an item by *value* or *text* it is case sensitive. In other words we would not be able to find the button if we used `:value => "add to cart"`. Since we did not specify any additional parameters *Watir* will select the first button with the value specified. If we want to select a specific book we could provide additional parameters. We will see how to do this later. The next call selects the link that takes us to the ckeckout page.

The remainder of the script fills in and submits the order form. Selecting an element in a `select_list` is also case sensitive. We must match the entry exactly.

Run the script. You should see the message "Thank you for your order".

Complete the script

Our script does a nice job of adding a book to the cart and completing a checkout. It doesn't verify it completed successfully. Our script is incomplete if it doesn't verify the task we automated actually worked. Let's add that to our script. If you have experience writing scripts in another language you might be tempted to write something like the following:

```ruby
if ! browser.text.include? "Thank you for your order"
  fail
end
```

In this example we are calling the text method on the `browser` object which returns a `String` with all of the contents of the page. We are next calling the `include?` method on the returned `String` passing it the value "Thanks you for your order". This method call will return *true* if the `String` is included and *false* if it isn't. In Ruby it is a standard practice to prepend a question mark to method names that return *true* or *false*.

Ruby provides us with a more concise way to say the same thing. Here is the same code written *the Ruby way.*

```ruby
fail unless browser.text.include? "Thank you for your order"
```

This new way is more concise and easier to read. Let's run the application once again.

It works. Let's change the message to read "Thank you for your coffee" and run again. This time it fails but the message is not very clear. We can clear this up by passing a text message to the `fail` method. Let's replace our *fail* line with this:

```ruby
fail "Browser text did not match expected value" unless
browser.text.include? "Thank you for your order"
```

Now we will get a much nicer message when our script fails.

In twelve lines of script we have completed a script that purchases a book and verifies a success message. Not bad. But it will get better!

Purchasing two books

If you love reading as much as I do you may be feeling like one book is not enough. Why purchase one book when you can purchase two? Let's write a new script that

purchases two books. In fact, we want to purchase two different books. This means that our current method of adding a book to the cart - `browser.button(:value => "Add to Cart")` - will not work since it always selects the first book. Fear not, *Watir* provides a solution. Many of the *Watir* methods allow you to provide multiple parameters for identification. The button method has this capability and therefore if you wish to click the second "Add to Cart" button on the page you can use `browser.button(:value => "Add to Cart", :index => 2)`. Let's put this new script in a file called `second_script.rb`.

> **Note => Multiple Parameters**
> Refer to the Watir Quick Reference in the appendix to identify which methods allow multiple parameters.

You now know everything you need to complete this new script that purchases two books. Go ahead and give it a try. I promise it will not hurt. This new script is nearly identical to the last script. But is this a good thing?

Removing duplication (D.R.Y.)

Duplication is one of our biggest enemies when writing scripts. Duplication means that we have to change things in multiple places when it needs to change. This leads to scripts that cost a lot to maintain. One of the largest risks to any automation effort is high costs to maintain existing scripts. In Agile we have the term DRY which means Don't Repeat Yourself. We want to make sure are scripts are always DRY.

Let's look at where we left off with the last script to see if we can discover duplication.

```ruby
require "rubygems"
require "watir"
browser = FireWatir::Firefox.new

browser.goto "http://localhost:3000/store"
browser.button(:value => "Add to Cart" :index => 1),.click
browser.link(:text => "Continue shopping").click
browser.button(:value => "Add to Cart" :index => 2),.click
browser.link(:text => "Checkout").click
browser.text_field(:id => "order_name").set("Cheezy")
browser.text_field(:id => "order_address").set("123 Main St.")
browser.text_field(:id => "order_email").set("cheezy@foo.com")
browser.select_list(:id => "order_pay_type").set("Check")
browser.button(:value => "Place Order").click

fail unless browser.text.include? "Thank you for your order"
```

At first glance you might not see duplication in our script. What about the call to add a

book? Other than the book number they are identical. But what can we do to remove this duplication?

In Ruby we have a concept called a *method*. We have already discussed calling *methods* but it is also possible to create your own. You may think of this as a place to put code that can be used from multiple locations. It takes the form of:

```
def methodname
  …
end
```

Let's create a method for purchasing a book and thereby eliminate the duplication in our script. In order to handle the different book numbers we will pass the book number into our method. We will also need to pass our `browser` object. The method could look like this:

```
def purchase_book_number(browser, num)
  browser.button(:value => "Add to Cart", :index => num).click
end

browser.goto "http://localhost:3000/store"
purchase_book_number(browser, 1)
browser.link(:text => "Continue shopping").click
purchase_book_number(browser, 2)
…
```

This seems a little better but it feels strange having to pass the `browser` object. Since the `browser` object is a *local variable* (See note below) we are forced to pass it to the method. Is there anything we can do to simplify this further? Absolutely. We can change the *local variable* to an *instance variable* by placing the @ symbol in front of it. If we change the *local variable* `browser` to the *instance variable* `@browser` the method can access it directly. Here is what this would look like:

```
def purchase_book_number(num)
  @browser.button(:value => "Add to Cart", :index => num).click
end

@browser.goto "http://localhost:3000/store"
purchase_book_number 1
@browser.link(:text => "Continue shopping").click
purchase_book_number 2
…
```

This seems more natural. Also notice that we did not put parenthesis around the number in the call to `purchase_book_number`. Ruby allows us to optionally not use

parenthesis and sometimes it adds to the readability. We decided to not use them here.

> **Note => Difference between local and instance variables**
> In Ruby there are several types of variables. The two most common are local and instance variables. The main difference between the two is how long they live and where you can access them. A local variable is only available within the context in which it was created and ceases to exist once you exit that context. For example, the `browser` local variable in the previous example was not created in the `purchase_book_number` method. Therefore, the `browser` variable could not be accessed in the method unless we passed it in. On the other hand, instance variables are available anywhere in the `class` to which they belong. We will learn more about classes in the next chapter but for now it is important to understand that our entire script has access to any instance variable we create.

Reusable parts

In the last section we learned about making methods to eliminate duplication. Another reason we might create methods is to make our scripts more expressive and readable. For example, it is far clearer to read `continue_shopping` than to read `@browser.link(:text => "Continue shopping").click`. In this way we can create a set of reusable methods that add clarity and readability to our script. Let's add this method.

```ruby
def continue_shopping
  @browser.link(:text => "Continue shopping").click
end

…
purchase_book_number 1
continue_shopping
purchase_book_number 2
…
```

Now that is much better. Let's keep going and see if we can make more methods to add clarity to the script. At the same time we are also creating a set of higher level methods that we could use in other scripts. Go ahead and make the changes yourself to see what you come up with.

My updates
After this change your script should look something like this:

```ruby
require 'rubygems'
require 'watir'

def go_shopping
  @browser = Watir::IE.new
```

```ruby
  @browser.goto "http://localhost:3000/store"
end

def purchase_book_number(num)
  @browser.button(:value => "Add to Cart", :index => num).click
end

def continue_shopping
  @browser.link(:text => "Continue shopping").click
end

def checkout_with(name, address, email, pay_type)
  @browser.link(:text => "Checkout").click
  @browser.text_field(:id => "order_name").set(name)
  @browser.text_field(:id => "order_address").set(address)
  @browser.text_field(:id => "order_email").set(email)
  @browser.select_list(:id => "pay_type").set(pay_type)
  @browser.button(:value => "Place Order").click
end

def verify_page_contains(text)
  fail unless @browser.text.include? text
end

def close_the_browser
  @browser.close
end

go_shopping
purchase_book_number 1
continue_shopping
purchase_book_number 2
checkout_with("Cheezy", "123 Main", "cheezy@foo.com", "Check")
verify_page_contains "Thank you for your order"
close_the_browser
```

This example is much longer that our previous example. The part I want you to focus on is the last seven lines. When writing *Watir* scripts it is our end goal to be able to write something as simple as this. The methods above these lines are there to enable us to write simple tests. The only problem we have at this time is that the reusable methods are only available to the one script in which they are contained. In the next section we will focus on how to make the methods available to any script.

Sharing methods with multiple scripts

In the previous section we wrote a set of reusable methods that we can use in our scripts. The problem is that in their current form they are not available to new scripts. Ruby has a solution for this and it is called a `Module`. We can define a `Module` like this:

```
module DepotHelper
   …
end
```

Go ahead and create a new file named `depot_helper.rb` in the same directory as your other scripts and create a module. Next you need to move all of the methods in our script to the module. Once the methods have been moved to the module we can reuse them in any script. Here's how. After moving the methods we need to require the module filename and then include the `Module` in the test script. Our updated script will look like this:

```
require 'rubygems'
require 'watir'
require 'depot_helper'

include DepotHelper

go_shopping
purchase_book_number 1
continue_shopping
purchase_book_number 2
checkout_with("Cheezy", "123 Main", "cheezy@foo.com", "Check")
verify_page_contains "Thank you for your order"
close_the_browser
```

The two lines of interest are lines three and five. Armed with this wonderful knowledge I would now like you to go back and refactor the first script that purchased one book and modify it to use our `DepotHelper module`. Much simpler, isn't it?

Shipping our orders

We are going to write one more script in this chapter. The depot application has an admin function that allows us to ship orders. Point your browser at http://localhost:3000/admin/login. You may login with the username "steve" and the password "secret". Once you login you will see the *Orders* link on the left of the screen. Select that link and you will be taken to a page displaying all orders place in the system. Go ahead and select one of the checkboxes and then press the "SHIP CHECKED ITEMS" button.

Your task is to write a script that ships orders. I want your script to:

1. Go to the page where we can ship orders
2. Select and ship one order
3. Verify the message when shipping one order
4. Select and ship two orders
5. Verify a different message when shipping two orders.

Make sure you supply the correct value for the button that ships the order.

Shipping orders
What did you learn? Run the script a second time. If you did not logout at the end of your script then you might have problems. This brings us to a very important point. Each script you write must put the application and environment back into a know state upon completion. In this case, if we do not logout and the next script assumes that the user is not logged in you might see failing tests. We will be discussing keeping our environment in a know state more over the next few chapters.

Also you may have learned that things are not always as they appear. The value for the button in our example is " SHIP CHECKED ITEMS ". The spaces before and after the text are important and the script will not find the button without them.

Doing something multiple times
There is one final thing I wish to do to our last script. I want you to create a method to ship the orders. I want your method to provide the ability to ship one, two, or any number of orders. In Ruby, we have numerous ways of doing something repeatedly. One simple way is with the `times` method.

```
5.times do
  puts "Hello, World!"
end
```

The previous listing will print the text "Hello, World!" five times. There is another from of this loop that passes the index to the block.

```
5.times do | index |
  puts "Number #{index}"
end
```

There are two new things introduced in this short example. The first is that we are passing the index for the block to the loop. The first time through the loop the `index` variable is set to zero. Each time through the loop the `index` variable is incremented. The other new thing in this script is the way we are inserting the value of `index` into the String passed to the `puts` method. The value of this String in the loop the first time is

"Number 0".

In order to write our new method we will need to use a number passed in to setup the loop, use the loop to determine which checkboxes to set and click the button to ship the items. Notice that I am adding 1 to the index since it is set to 0 the first time through the loop.

```
def ship_this_many_orders(num_orders)
  num_orders.times do | index |
    @browser.checkbox(:index => index + 1).set
  end
  @browser.button(:value => "  SHIP CHECKED ITEMS  ").click
end
```

I think we now have enough *Watir* experience to move on.

Appendix B

RSpec Matchers