



2nd Edition

Practical Web Test Automation

Test Web Applications Wisely



ZHIMIN ZHAN

Practical Web Test Automation

Test web applications wisely with Selenium WebDriver

Zhimin Zhan

This book is for sale at <http://leanpub.com/practical-web-test-automation>

This version was published on 2018-09-23



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2012 - 2018 Zhimin Zhan

I dedicate this book to my mother and father for their unconditional love.

Contents

Preface	i
Who should read this book?	ii
How to read this book?	iii
What's inside the book?	iii
Test scripts, Screencasts and other resources	iv
Send me feedback	iv
Acknowledgements	v
1. What is Web Test Automation?	1
1.1 Test automation benefits	1
1.2 Reality check	2
1.3 Reasons for test automation failures	3
1.4 Successful web test automation	5
1.5 Learning approach	6
1.6 Next action	7
2. First Automated Test	8
2.1 Test Design	8
2.2 Installing TestWise (about 2 minutes)	9
2.3 Create test	10
2.4 Create test case from recorded test steps	13
2.5 Run test in browser	15
2.6 When a test failed...	16
2.7 Wrap up	17
3. How Automated Testing works	19
3.1 Web test drivers	20
3.2 Automated testing rhythm	20
3.3 Test frameworks	23

CONTENTS

3.4	Run tests from command line	27
4.	TestWise - Functional Testing IDE	29
4.1	Philosophy of TestWise	29
4.2	TestWise project structure	30
4.3	Test execution	31
4.4	Keyboard navigation	32
4.5	Snippets	33
4.6	Script library	33
4.7	Test refactoring	34
4.8	Wrap up	34
5.	Case Study	35
5.1	Test site	35
5.2	Preparation	35
5.3	Create test project	36
5.4	Test Suite: Sign in	37
5.5	Test Suite: Select Flights	42
5.6	Enter passenger details	46
5.7	Book confirmation after payment	48
5.8	Run all tests	51
5.9	Wrap up	52

Preface

On April 3 2013, Wired published an article “[The Software Revolution Behind LinkedIn’s Gushing Profits](http://www.wired.com/business/2013/04/linkedin-software-revolution/)”¹. The revolution “completely overhauled how LinkedIn develops and ships new updates to its website and apps, taking a system that required a full month to release new features and turning it into one that pushes out updates multiple times per day.” LinkedIn is not alone, Google has accomplished this long before that. As a matter of fact, LinkedIn’s success is tracked back to luring a Google veteran in 2001. “[Facebook is released twice a day](http://www.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920)”² and they claimed “[keeping up this pace is at the heart of our culture](http://www.seleniumconf.org/speakers/)”³.

Release software twice a day! For many, that’s unimaginable. You may wonder how they could ensure quality (and you know the high standard from them). The answer is, as the article pointed out, to use “automated tests designed to weed out any bugs.”

After working on numerous software projects for a number of years, I witnessed and had been part of many what I call ‘release panic syndromes’. That is, with the deadline approaching, the team’s panic level rises. Many defects were found from the last round of manual testing by the testers. The manager started prioritizing the defects (or adjusting some to features), and programmers rushed to fix just the critical ones. Testers restarted the testing on the new build that had fixed some but not all the defects. Then here came the bad news: several previously working features are now broken, Argh!

I believe there is a better way to do software development that does not have to involve this kind of stress and panic. This is how my interest in automated testing started (in 2006). I made the right decision to use free, open source and programming based test frameworks. (It is quite obvious now, as Selenium WebDriver is the best sought after testing skill on the job market. Back then, people turned to record/playback commercial tools with vendor proprietary test script syntax). The first test framework I used (for my pet projects) was Watir. I was quickly convinced that this approach was the answer.

In 2007, I had the opportunity to put my approach into practices in a government project. The outcome was beyond everyone’s expectation: over two years and countless releases, there were no major defects reported by customers. The team had high confidence in the product. These automated tests also provided the safety net for some major refactorings,

¹<http://www.wired.com/business/2013/04/linkedin-software-revolution/>

²<http://www.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920>

³<http://www.seleniumconf.org/speakers/>

which would have not been possible without them. A business analyst once said, “before every demonstration to our customers, it is a good feeling of knowing every release has been thoroughly tested.” The synergy of flexible test framework, maintainable test design, team collaboration with the same simple testing tool and continuous integration supporting functional test execution really made a big difference.

There is now a clearly converging trend in web application development on technology choices, such as cloud deployment, light and productive web frameworks such as “[Ruby on Rails](#)”⁴, JQuery JavaScript Library, Twitter Bootstrap UI themes, Font Awesome icons, ..., etc. The competitions among web applications are less on technologies, but weigh more on the development process to ensure pushing out high quality releases frequently. A fact: Facebook was not the first social networking web site.

A friend of mine, who developed a quite successful public web application, told me in an uneasy tone that he just found out another competitor product at a cheaper price. This is inevitable, the competition among web applications is global, which means, there are people working at 10% of your hourly rate to compete against you. The only way to win the race, in my opinion, is to greatly enhance your productivity and reduce maintenance cost. This can be achieved by applying test automation and continuous integration with instant benefits without much effort (if doing it properly). My reply to my friend: “If your competitors start to invest in test automation seriously, you shall be worried.”

In Appendix II, I share my experience on developing ClinicWise, a modern web-based clinic management system. Thanks to comprehensive automated UI testing, ClinicWise is frequently released (daily) with new features and updates. ClinicWise is developed and maintained in my spare time.

The purpose of this book is to share my journey of test automation for web applications: from writing the first test to developing and maintaining large number of automated test scripts.

Who should read this book?

Everyone who works on a software team (including testers, programmers, business analysts, architects and managers) builds a web application and wants to improve the quality of software while saving time and money can benefit from reading this book. It may sound like a bold statement, but it is the outcome I obtained from some projects whose team members have embraced the techniques and practices presented in this book. Those projects delivered reliable software releases frequently, stress free. You can achieve this too.

⁴<http://yourstory.com/2013/02/startup-technologies-top-6-technologies-used-at-startups/>

Prior experience with automated testing is not necessary. Basic programming concepts will help, but again, not necessary.

How to read this book?

I strongly recommend readers to read through Chapters 1-9 in order, only skip Chapter 4 if you have decided on the testing editor or IDE. Chapters 10-15 are largely independent from one another. You can therefore read them in the order that suits your interests. Readers can also skim through and come back for details later if necessary.

Some chapters contain hands-on exercises (with step by step guides). Typically it will take about 10-30 minutes to complete an exercise. Readers can choose to follow the exercises while or after reading a chapter. The main point is: to master test automation, you have to do it.

What's inside the book?

In part 1, I introduce Web Test Automation and its benefits, which many believe but few actually achieve it. I use a metaphor to illustrate practical reasons why most software projects conduct functional testing manually despite knowing the great benefits of test automation. Then the journey starts with a case study to help write your first Watir automated test in about 10 minutes.

In part 2, I present a brief introduction of test frameworks and tools, followed by a case study showing the development of Selenium WebDriver tests for a live test site with the help of a recorder. Along the way, some testing techniques are introduced.

In part 3, I present an intuitive and maintainable automated test design: using reusable functions and page objects, followed by a case study showing the transforming of recorded test scripts to a maintainable way. Then I introduce an important concept: functional test refactoring, a process of testers applying refactorings to test scripts efficiently with refactoring support in testing tools such as [TestWise IDE](http://testwisely.com)⁵.

With a growing number of automated tests, so is the test execution time. Long feedback loops really slow down development. In part 4, I show how team collaboration and continuous integration can help to improve the feedback time greatly.

⁵<http://testwisely.com>

In Part 5, I switch the attention to several WebDriver backed variant frameworks: Watir, RWebSpec and Capybara and introduce another test syntax framework [Cucumber](http://cukes.info/)⁶. Then I will show how to apply the maintainable test design and techniques to them. Finally I share some strategies to apply test automation to your project.

Test scripts, Screencasts and other resources

To help readers learn test automation more effectively, the book has a dedicated site at: <http://zhimin.com/books/pwta>⁷, which contains the following resources:

- **Software.** Test automation is not necessarily expensive. All test frameworks featured in this book are free and open-sourced. Testing tools used for the exercises in this book are also free, and there are instructions to cater for other text-based testing tools.
- **Sample test scripts.** The sample test scripts for the exercises are ready-to-run. This book covers several popular test and syntax frameworks: Selenium-WebDriver, Watir, RWebSpec, RSpec and Cucumber. To help readers understand the differences, I have created 6 test projects with different combinations: <https://github.com/testwisely/agiletravel-ui-tests>⁸.
- **Sample web sites.** For readers who need web sites to try out automated test scripts, I have prepared two test sites for you:
 - *Agile Travel*: a simple flight booking site, which is used in the exercises.
 - *AdminWise*: a feature rich web 2.0 site with modules such as membership and library.
- **Tutorial screencasts.** There are screencasts for readers who will learn better with audio and video, so you will be able to see how it is done step by step.

For access code see the [Resources](#) section of this book.

Send me feedback

I will appreciate hearing from you. Comments, suggestions, errors in the book and test scripts are all welcome. You can submit your feedback on the book web site (<http://zhimin.com/books/pwta>).

⁶<http://cukes.info/>

⁷<http://zhimin.com/books/pwta>

⁸<https://github.com/testwisely/agiletravel-ui-tests>

Acknowledgements

I would like to thank everyone who sent feedback and suggestions, particularly Mingli Zhou, Darren James, Tim Wilson, Lloyd Blake, Hoang Uong and Lien Nguyen, for their time and wisdom.

I owe a huge ‘thank you’ to people behind great open-source testing frameworks such as Selenium-WebDriver and RSpec, and of course, the beautiful Ruby language.

Functional testing via User Interface is practical and light on theory, so is this book. I hope you find this book useful.

Zhimin Zhan

Brisbane, Australia

1. What is Web Test Automation?

Web Test Automation, or automated functional testing for web applications via the Graphical User Interface (GUI), is the use of automated test scripts to drive test executions to verify that the web application meets its requirements. During execution of an automated test for a web site, you see mouse and keyboard actions such as clicking a button and typing text in a text box in a browser, without human intervention. Web Test Automation sits under the category of black-box functional testing, where the majority of test efforts is in software projects.

Functional Testing vs Unit Testing vs Non-Functional Testing

Functional testing is to verify function requirements: **what** the system does. For example, “*User can request a password reset by providing a valid email*”. Functional testing is the focus of this book.

Unit testing is a type of white box testing performed by programmers at source code level. It is of no concerns to software testers. Unit test is a term that gets misused a lot. A more correct term would be “Programmer Test”. A product that passes comprehensive programmer tests can still fail on many functional tests. That’s because programmers tests are from a programmer’s perspective, and functional tests are from a user’s perspective. A programmer test is a kind of automated test too.

Non-functional testing is the testing of **how** the system works. For example, “*The response time of the home page must not exceed 5 seconds*.” Some type of non-functional testings, load testing in particular, utilize automated test tools as well.

1.1 Test automation benefits

The benefits of test automation are plenty. Below are four common ones:

- **Reliable.** Tests perform the same operations precisely each time they are run, therefore eliminating human errors.

- **Fast.** Test execution is faster than done manually.
- **Repeatable.** Once tests are created, they can be run repeatedly with little effort, even at lunch time or after working hours.
- **Regression Testing.** “The intent of regression testing is to ensure that a change, such as a bug fix, did not introduce new faults” [Myers, Glenford 04]. Comprehensive manual regression testing is almost impossible to conduct for two reasons: the time required and human errors. As Steve McConnell pointed out, “The only practical way to manage regression testing is to automate it.” [McConnell]



What do I like about test automation?

If you want me to use only one adjective to describe web test automation, it is **fun**. As a software engineer, I enjoy creating something that can do the work for me. I have a habit of triggering execution of a test suite before going out for lunch. I like the feeling of “still working” while enjoying a meal. When I come back, a test report is there.

As a product owner, I think it is essential to release software frequently without fear. To achieve that, comprehensive test automation via UI is a must.

1.2 Reality check

With more software projects adopting agile methodologies and more software application developments moving towards the Web, you would assume web test automation would be everywhere now. The answer is, sadly, no. In fact, functional testing in many projects is still executed in pretty much the same way: manually. For past decade, I have seen automated UI testing was done poorly or not at all in numerous “agile” projects, however, it has been talked a lot. Michael Feathers, a renowned agile mentor and the author of [Working Effectively with Legacy Code](http://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052)¹, summarized better than what I can in this blog article.

¹<http://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052>



UI Test Automation Tools are Snake Oil - Michael Feathers

It happens over and over again. I visit a team and I ask about their testing situation. We talk about unit tests, exploratory testing, the works. Then, I ask about automated end-to-end testing and they point at a machine in the corner. That poor machine has an installation of some highly-priced per seat testing tool (or an open source one, it doesn't matter), and the chair in front of it is empty. We walk over, sweep the dust away from the keyboard, and load up the tool. Then, we glance through their set of test scripts and try to run them. The system falls over a couple of times and then they give me that sheepish grin and say "we tried." I say, "don't worry, everyone does." [Feathers 10]

Michael Feathers was spot on summarizing the status of automated end-to-end testing. Over last 10 years, I pretty much experienced the same: software teams were either 'pretending automated UI testing' or 'not doing it at all'.

1.3 Reasons for test automation failures

The software testing survey conducted by Innovative Defense Technologies in 2007 [IDT07] shows *"73% of survey respondents believe Automated Testing is beneficial but few automate"*. The top reasons for survey participants not automating their software testing (while agreeing with the benefits) are:

- lack of time
- lack of budget
- lack of expertise

These reasons sound right to most people. However, saving time and money are two benefits of test automation, isn't that a contradiction (for lack of time and budget)? What are the real difficulties or challenges, apart from political or project management ones, that projects encounter during their adventures in automated testing?

To make it easy to understand, we can compare a project's test automation attempt with a person who is trying to climb over a standing two-hump camel from the front. Let's consider each of the following challenges he faces:

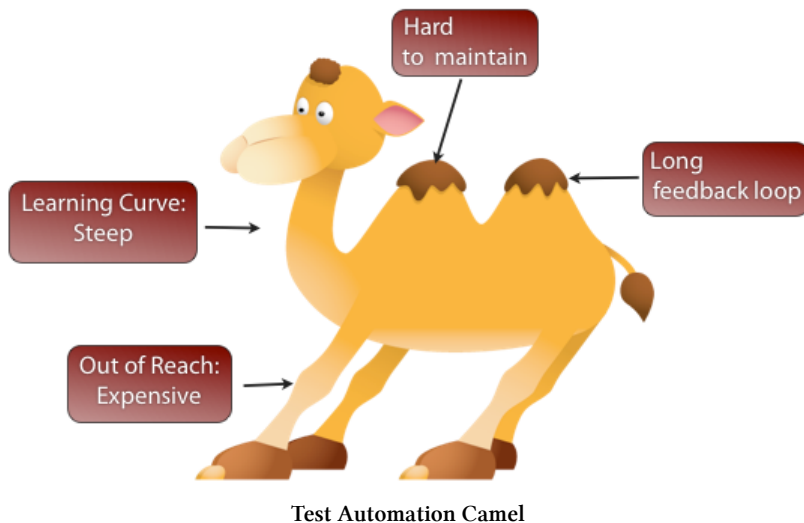


Figure 1-1 Test Automation Camel (graphics credit: www.freevectordownload.com)

1. Out of reach: Expensive

Commercial testing tools are usually quite expensive (I won't list prices here, in fact, I couldn't get prices for some so-called leading testing tools on their web sites, which is telling in itself). Automated testing is one of a few activities in software projects that the whole team can contribute to and benefit from. Besides testers, programmers may run automated tests for self verification and business analysts may utilize automated tests for customer demonstrations. However, high price of commercial testing tools makes the whole team's adoption of automated testing unfeasible.

There are free, open-source testing frameworks, such as Selenium WebDriver and Watir, both of which are featured in the classic book '[Agile Testing](http://www.agiletester.ca/)²' by Lisa Crispin and Janet Gregory. However the idea of free and open-source testing frameworks is still not appealing to many test managers. (Update: the previous statement was written in 2010, the situation has changed now as Selenium WebDriver is the dominant testing framework). Lack of skills, dedicated tools and support are their main concerns.

2. Steep Learning Curves: Difficult to learn

Traditional commercial tools are usually focused on a Record and Playback approach with test scripts in a vendor proprietary syntax. It looks easy when you watch the sales presentations. Unfortunately, it is a quite different story in real life (a programmer's minor

²<http://www.agiletester.ca/>

change to the application can ruin your hours of recording). When testers have to open the raw test scripts (generated by recorders) to edit, reality bites.

Open source test frameworks, on the other hand, require some degree of programming efforts, Selenium-WebDriver and Watir are among the popular ones. With programming, they provide flexibility needed for automated testing. However, the fact is that the majority of software testers do not possess programming skills, and many of them feel uncomfortable to learn it. Besides, there are few dedicated testing tools supporting these open-source test frameworks designed to suite testers. (Programming IDEs are designed for programmers, not for testers who may find them complicated and overwhelming).

3. Hump 1: Hard to maintain

Software under development changes frequently, and automated UI test scripts are vulnerable to application changes. Even a simplest change to the application could cause many existing test scripts fail. This, in my view, is the most fundamental reason for test automation failures.

4. Hump 2: Long feedback loop

Compared to programmer tests (which if written well, should have an execution time under 0.1 second), automated functional tests through UI are relatively slow. There is practically very little that testers can do to speed up execution of functional tests. With the number of test cases growing, so will be the test execution time. This leads to long feedback gap, from the time programmers committed the code to the time test execution completes. If programmers continue developing new features or fixes during the gap time, it can easily get into a tail-chasing problem. This will hurt team's productivity, not to mention team's morale.

New Challenges for testing Web applications

Specifically to web applications, with adoption of AJAX (Asynchronous JavaScript and XML) and increasing use of JavaScript, websites nowadays are more dynamic, therefore, bring new challenges to web test automation.

1.4 Successful web test automation

Having identified the reasons for test automation failures in projects, it becomes clear what it takes to succeed in web test automation:

1. Test scripts must be easy to read and maintain.
2. Testing framework/tools are easy to learn, affordable and support team collaboration.

3. Test execution must be fast.

Is that all possible? My answer is ‘Yes’. The purpose of this book is to show how you can achieve these.

1.5 Learning approach

This is not just another book on testing theories, as there are no shortage of them. In this book, we will walk through examples using test framework Selenium WebDriver and functional testing IDE TestWise. The best way to learn is to start doing it.

My father is a well respected high school mathematics teacher in our town. His teaching style is “teaching by examples”. He gets students to work on his carefully selected math exercises followed by concise instruction, then guide students who face challenges. By working with many testers, I found this is the most effective way for testers to master automated testing quickly.

For most web applications, regardless of technologies they are developed on, Microsoft Windows is often the target platform (at least for now). It will be the main platform for our exercises in this book:

Web Browser:	Chrome, Internet Explorer and Firefox
Test Framework:	Selenium WebDriver
Testing Tool:	TestWise IDE

If you are Mac user, like myself, the learning process is the same (majority of the test scripts run without change) except the screenshots shown in the book look different. All the techniques and test scripts are directly applicable for cross-browser testing.

On testing tools, I use TestWise, a testing IDE that supports Selenium WebDriver and Watir (*TestWise has free 30-day trial*), in this book. For readers who prefer their own favorite editors or IDEs, you can still use them, as all test scripts shown in this book are plain text. I will also provide instructions on how to execute tests from the command line.

Example test scripts for chapters in this book can be downloaded at the book site, and you can try them out by simply opening in TestWise and run. I have provided screencasts there as well, readers can watch how it is done.

In this book, we will focus on testing standard web sites (in HTML), excluding vendor specific and deprecated technologies such as Flash and SilverLight. The techniques shown in this book are applicable to general testing practices.

1.6 Next action

Enough theory for now. Let's roll up sleeves and write some automated tests.

2. First Automated Test

A journey of a thousand miles must begin with a single step.

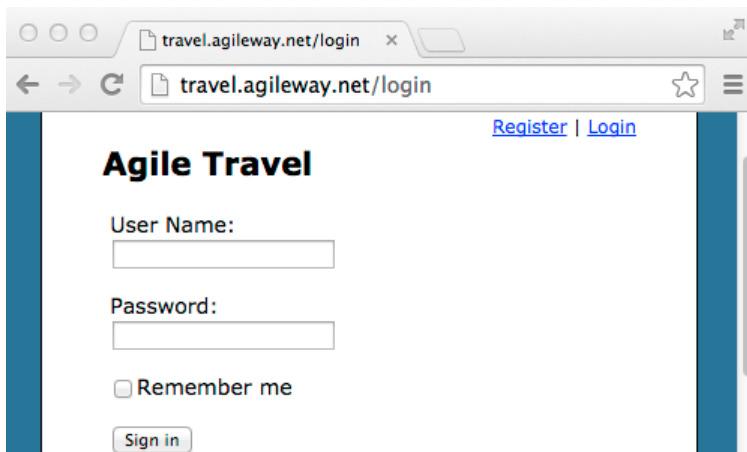
– Lao Tzu

Let's write an automated web test. If you are new to automated testing, don't feel intimidated. You are going to see your first automated test running in Internet Explorer in about 10 minutes, and that includes installing the test tool!

2.1 Test Design

A test starts with a requirement (called User Story in agile projects). Quite commonly, the first simple requirement to test is: User Authentication. We will use this requirement for our first test in this exercise.

By analyzing the requirement and the application (see the screenshot below),



Agile travel login

we can start to collect the test data:

Site URL: <http://travel.agileway.net>
User Login/Password: agileway/testwise

and design the test steps:

1. Enter username “agileway”
2. Enter password “testwise”
3. Click button “Sign In”
4. Verify: “Welcome agileway” appears

You might by now be saying “*there is no difference from manual testing*”. You are right. If you currently work as a manual tester, you probably feel a relief at knowing your test design skills can apply to automated testing. As a matter of fact, we usually perform the test steps manually as verification of test design before writing automated test scripts.

Now we are going to automate it. The main purpose of this exercise is to help you write an automated Selenium WebDriver test case and watch it running in a browser, in a matter of minutes. Don’t pay attention to details yet, as it will become clear as we move on. If you get stuck, follow the screencast for this exercise at <http://zhimin.com/books/pwta>¹.

2.2 Installing TestWise (about 2 minutes)

We will use TestWise, a Functional Testing IDE built for Selenium WebDriver, for this exercise.

Prerequisite

- A PC with MS Windows 7 or later
- TestWise recorder which requires Mozilla Firefox

Download

- *TestWise* from <http://testwisely.com/testwise/downloads>² (~20MB download).

Install

- **TestWise IDE.**

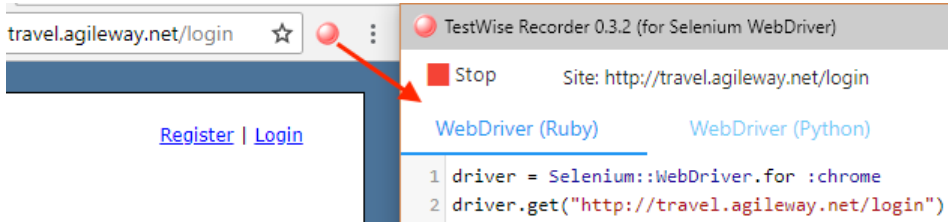
Double click *TestWise-x.x-setup.exe* to install, accept all default options. The default installation folder is *C:\agileway\TestWise*. Launch TestWise after the installation completes.

¹<http://zhimin.com/books/pwta>

²<http://testwisely.com/testwise/downloads>

- **TestWise Recorder.**

TestWise Recorder is a Chrome extension, which records your operations into executable Selenium WebDriver and Watir test scripts while you navigate through your web application in Chrome. To install, search for “TestWise Recorder” in [Chrome Web Store page](#)³, click “+ ADD TO CHROME” button. Click the recorder icon on toolbar to enable recording.



Enable recorder

You may use any other Selenium WebDriver recording tool

2.3 Create test

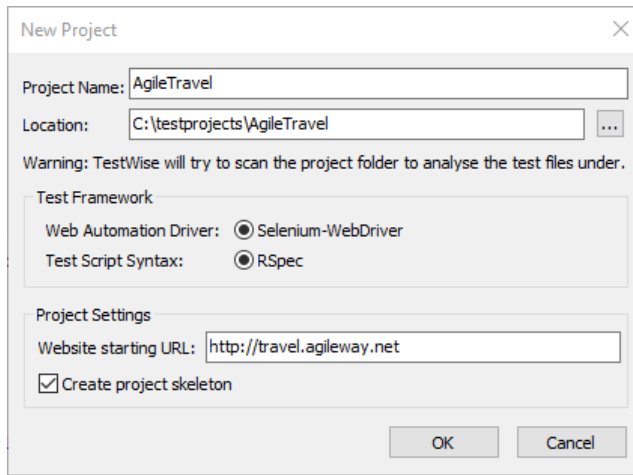
Now we are ready to create the test for our requirement: “User can login the site”. Hope you still remember the test design steps and test data.

Create new test project

TestWise has a project structure to organize test scripts. This structure is simply a folder containing all test related files such as test scripts and test data.

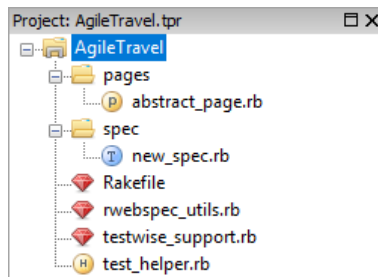
As we start from scratch, we need to create a new project first. If a sample project is already opened in TestWise, we need to close it. Select menu File → New Project, which will bring up the window shown below.

³<https://chrome.google.com/webstore/detail/testwise-recorder/febfgamlejngokejcainmklgcfphjbok>



Create Project

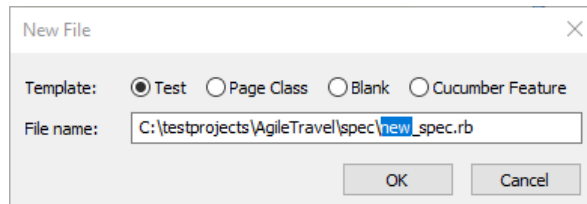
Enter project name, project folder and URL of web site to be tested. In this case, we enter “*Agile Travel*”, “*C:\testprojects\AgileTravel*” and “*http://travel.agileway.net*” respectively, then click ‘OK’ button. TestWise will create the project with skeleton files.



Project Skeleton

Create test script file

Now create the test script file for our test. Select ‘File’ → ‘New File’,



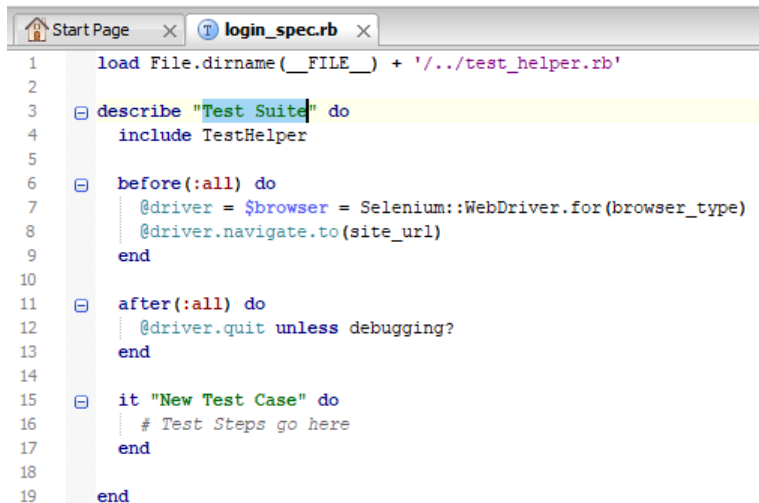
New test

Type text 'login' and press Enter to create new test script file: *login_spec.rb*



Try naming the test script file something related to the requirement, so you can find it easily later.

A new editor tab is created and opened with a test skeleton:



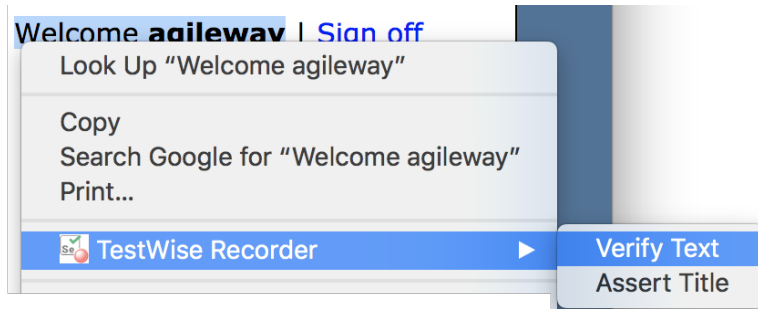
Login test

Recording

Open the site URL <http://travel.agileway.net> in Firefox and enable 'TestWise Recorder SideBar'. Perform the test steps below manually:

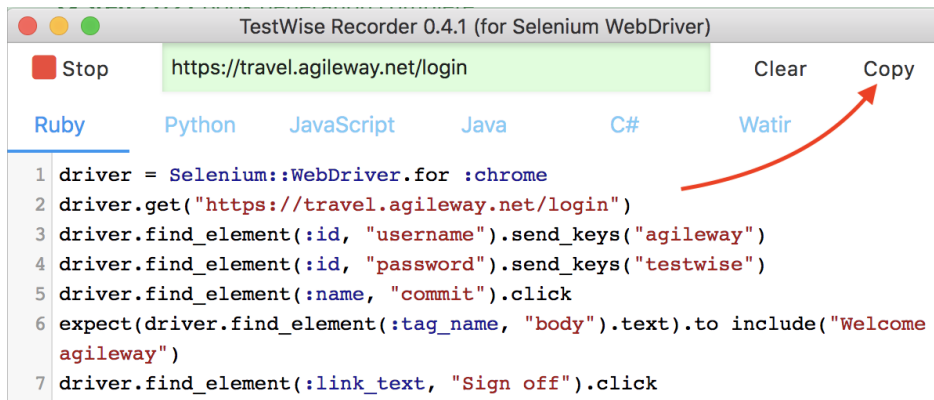
1. Enter username 'agileway'
2. Enter password 'testwise'

3. Click 'Sign In' button
4. To add verification for text 'Welcome agileway', highlight the text in browser, right click and select "Verify Text" in context menu.
5. Click Sign off link



Recording

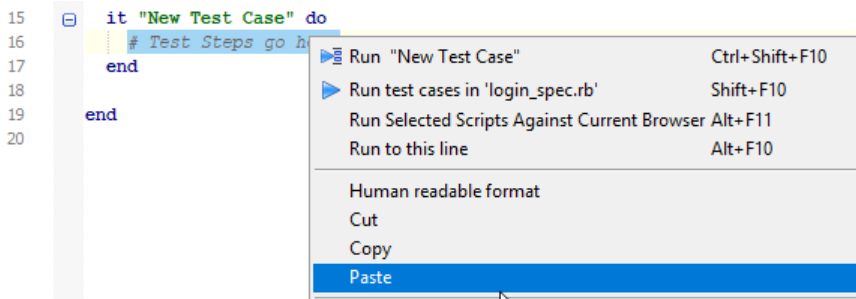
Test steps are recorded along the way. Once done, inside the TestWise Recorder window, click 'Copy' button on top to copy recorded test steps to clipboard.



Copy recorded test steps

2.4 Create test case from recorded test steps

Switch to the TestWise IDE (the *login_spec.rb* editor tab shall be still active), paste recorded test scripts into the test case.



Paste test steps

The test case is created. While we are here, update the test suite name (the string in describe "...") to "User Authentication" and the test case's name (the string in it "...") to "User can login with valid user name and password".

The first two copied steps:

```
driver = Selenium::WebDriver.for :chrome;
driver.get("http://travel.agileway.net/login")
```

start Chrome browser and navigate to our target server. You can delete them if using TestWise, as they are already included in before(:all) block (created by TestWise in a more generic format). The test scripts in the TestWise shall be like the below:

```
load File.dirname(__FILE__) + '/../test_helper.rb'

describe "User Authentication" do
  include TestHelper

  before(:all) do
    @driver = $browser = Selenium::WebDriver.for(browser_type)
    @driver.navigate.to(site_url)
  end

  after(:all) do
    @driver.quit unless debugging?
  end

  it "User can login with valid user name and password" do
    driver.find_element(:id, "username").send_keys("agileway")
    driver.find_element(:id, "password").send_keys("testwise")
    driver.find_element(:name, "commit").click
    expect(driver.find_element(:tag_name, "body").text).to include("Welcome agileway")
  end
end
```



```
driver.find_element(:link_text, "Sign off").click
end

end
```

2.5 Run test in browser

Select the Chrome browser icon and press ► on the tool bar (highlighted in the screenshot below) to run the test case, and you can watch the test execution in a Chrome window.

Set browser and target URL specifically

Some readers might ask *“I don’t see the target server URL and Chrome browser being set in the test script”*.

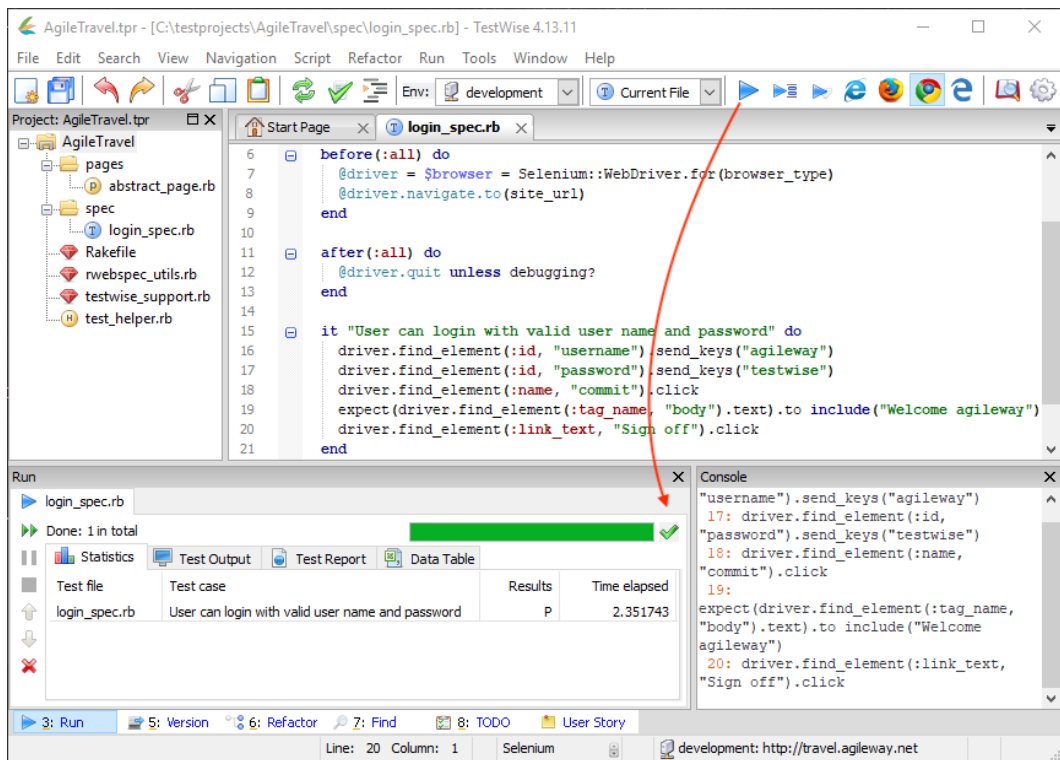
```
@driver = $browser = Selenium::WebDriver.for(browser_type)
@driver.navigate.to(site_url)
```

The `browser_type` and `site_url` are defined in `test_helper.rb`, which you can easily modify. More importantly, with IDE support, you can run tests against another target server (in *Project settings*) and browser quickly in IDE. Feel free to change the target browser to Firefox or IE (provided that the browser and its webdriver are installed correctly) and run the test again.

If you want to set the browser type and server URL specifically in each individual test script, you can.

```
@driver = $browser = Selenium::WebDriver.for(:firefox)
@driver.get("http://travel.agileway.net")
```

Note: `driver.navigate.to` is equivalent to `driver.get`, which navigates to a web address.



TestWise run

The green tick means the test passed.

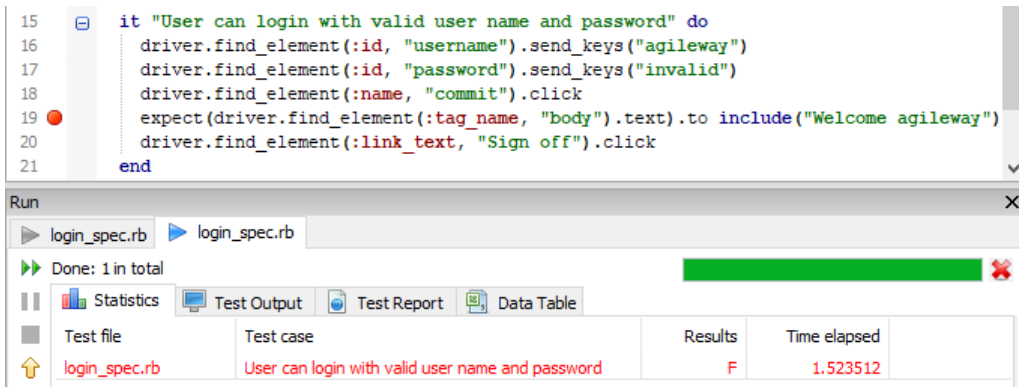
2.6 When a test failed...

We just saw a successful automated test. Naturally, you will ask what will happen when a test fails? As a matter of fact, during development of an automated test script, we are more likely to get errors or failures before we get it right. It is up to the technical testers to analyze the cause: is it a real application error or incorrect test scripts?

Next, we will make a simple change to the above test script to make it fail:

```
driver.find_element(:name, "password").send_keys("invalid") # will fail
```

Click ▶ to run the test. As expected, the test failed.



Test failed

In TestWise, the test execution is marked as “Failed” and ● is shown on line 18 of the test script indicating where the failure is.

We, as human, knew the reason for this failure: a wrong password was provided. From the test script’s “point of view”, it failed due to this assertion not met: finding the text “Welcome agileway” on the page.

If you want to find more details about the cause for test failure, check the text output of test execution including error trace under “Test Output” tab.



Failed test output

2.7 Wrap up

Let’s review what we have done in this chapter. Besides test design, we

- Installed TestWise IDE
- Installed TestWise Recorder
- Created a test project in TestWise IDE
- Recorded test scripts using TestWise Recorder in Firefox
- Created test script from pasted test steps
- Ran test case in browser (pass and failed)

Hopefully you were able to do all that within 10 minutes! You can view the screencast for this exercise online at the book's website at <http://zhimin.com/books/pwta>⁴.

⁴<http://zhimin.com/books/pwta>

3. How Automated Testing works

In the previous chapter, we created an automated functional test running in a web browser, Chrome. This was done by simulating a user interacting with the browser: typing texts and clicking buttons.

Before we move on, let us examine our test targets - web applications (or websites). Simply speaking, a web site consists of many web pages. Behind each web page there is an HTML (HyperText Markup Language) file. Browsers download the HTML files and render them.

HTML defines a set of standard web controls (aka elements) we are familiar with, such as text boxes, hyperlinks, buttons, check boxes, etc. For web application testing, we interact with these controls as well as the texts that get marked up in the HTML such as labels and headings.

Now let us review the test script we created in the last exercise:

```
it "[01] User can login" do
  driver = Selenium::WebDriver.for(:chrome)
  driver.navigate.to("http://travel.agileway.net")
  driver.find_element(:id, "username").send_keys("agileway")
  driver.find_element(:id, "password").send_keys("testwise")
  driver.find_element(:xpath, "//input[@value='Sign in']").click
  expect(driver.find_element(:tag_name, "body").text).to include("Welcome agileway")
end

# next test, comments start with '#'
```

Within a test case, test steps can be classified into the following two categories:

- **Operation** (also called step)

Performing some kind of keyboard or mouse action on a web page. The above example test has three operations:

```
driver.find_element(:id, "username").send_keys("agileway")
driver.find_element(:id, "password").send_keys("testwise")
driver.find_element(:xpath, "//input[@value='Sign in']").click
```

- **Check** (also called assertion)

Verifying the web page meets the requirement.

```
the_page_text = driver.find_element(:tag_name, "body").text
expect(the_page_text).to include("Welcome agileway")
```

3.1 Web test drivers

Web test drivers enable web controls to be driven by test scripts with a certain syntax, for testing purposes. All web test drivers covered in this book are free and open-source.

Selenium WebDriver

Selenium was originally created in 2004 by Jason Huggins, who was later joined by his other ThoughtWorks colleagues. Selenium supports all major browsers and tests can be written in many programming languages and run on Windows, Linux and Macintosh platforms.

Selenium 2 is merged with another test framework WebDriver led by Simon Stewart at Google (that's why you see 'selenium-webdriver'), Selenium 2.0 was released in July 2011.

Here is an example test in Selenium WebDriver:

```
require "selenium-webdriver"
driver = Selenium::WebDriver.for(:firefox) # or :ie, :chrome
driver.navigate.to "http://www.google.com"
driver.find_element(:name, "q").send_keys "WebDriver IDE"
driver.find_element(:name, "btnG").click # "btnG" is the 'Search' button
```

3.2 Automated testing rhythm

Regardless of which test framework you use, the 'testing rhythm' is the same:

1. Identify a web control
2. Perform operation on the control
3. Go to step 1 until reach a check point
4. Check
5. Go to step 1 until the test ends

Identify web controls

To drive controls on a web page, we need to identify them first.

Let's look at this sample web page:

User Name:

Password:

☐ Remember me

Its HTML source (you can view the HTML source of a web page by right clicking in the web page and selecting “View Page Source”):

```
User name: <input type="text" name="username" size="20"/>
Password: <input type="password" id="pwd_box" name="password" size="20"/>
<input type="submit" name="commit" value="Sign in"/>
```

Though the username and password appear the same (text box) on the browser, they are quite different in source. Some attributes in HTML tags tell web browsers how to render it, such as `size="20"` in user name text box. More importantly, application developers use attributes such as “name” (not exclusively) to determine user’s input is associated to which control.

We can identify web controls by using these attributes for testing. Here is one way to identify them in Selenium:

```
driver.find_element(:name, "username")
driver.find_element(:id, "pwd_box")
driver.find_element(:xpath, "//*[@value='Sign in']")
```

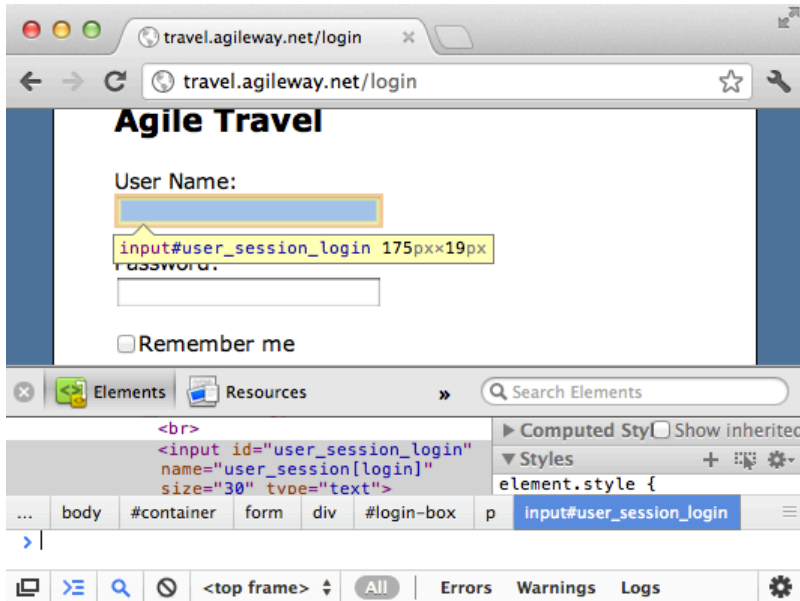
As you can see, these three test steps use three different attributes for three controls.

Obviously the easiest way to identify web controls is to use a recorder (a tool records user’s operation and generate test scripts), if you have one installed. However, in my opinion, it is essential for technical testers to master and be comfortable to do it manually. The reasons are:

- Some test frameworks don’t have recorders or have outdated ones
- Recorders might not work for certain circumstances

- Lack of freedom on choosing preferred attribute (for identifying controls)

In modern browsers, it is actually quite easy to identify element attributes (in HTML source) manually: just right-click on any page element and select Inspect Element.



Inspect in Chrome

Drive web controls

Once we have identified a web control, the next step is to perform required operation with it, such as typing text in a text field, clicking for a button, clearing a check box, and so on. Though different test frameworks have different syntax, the idea is the same.

Here are some examples:

```
driver.find_element(:name, "user[name]").send_keys "bob"
driver.find_element(:id, "next_btn").click
```

Check

The purpose of testing is to verify that a piece of function serves its purpose. After 'driving' the application to a certain point, we do checks (maybe that's why it is called 'checkpoint' in some testing tools).

In the context of web testing, typical checks are:

- verify certain texts are present
- verify certain HTML fragment are present (different from the above, this is to check raw page source)
- verify page title
- verify a link is present
- verify a web control is present or hidden

One key feature of Test frameworks (more in next section) is to provide syntax conventions to perform verifications as above. Here are some examples:

- xUnit (assertion style)

```
assert browser.html.include?("Payment Successful!")
assert browser.button(:text, "Choose Selenium").enabled?
assert browser.title == "User Registration"
```

- RSpec

```
expect(driver.page_source).to include("Payment Successful!")
expect(browser.find_element(:link_text, "Continue").displayed?).to be_truthy
expect(driver.title).to eq("User Registration")
```

3.3 Test frameworks

Web test drivers, such as Selenium WebDriver, drive browsers. However, to make effective use of them for testing, we need put them in a test framework which defines test structures and provides assertions (performing checks in test scripts).

xUnit

xUnit (JUnit and its cousins) test frameworks are widely used for unit testing by programmers. xUnit can be used in functional test scripts too, but it is not my preference, as it is not as expressive as the ones below.

RSpec

RSpec is a popular Behaviour Driven Development (BDD) framework in Ruby.

More expressive

Comparing to xUnit test frameworks, RSpec tests are easier to read. For example, for the JUnit test below:

```
class UserAuthenticationTest {  
  public void testCanLoginWithValidUsernameAndPassword {  
    // ...  
  }  
  public void testAccessDeniedForInvalidPassword() {  
    // ...  
  }  
}
```

Its RSpec version will be like this:

```
describe "User Authentication" do  
  it "User can login with valid login and password" do  
    # ...  
  end  
  
  it "Access denied for invalid password" do  
    #...  
  end  
end
```

Execution Hooks

Execution hooks are similar to `setUp()` and `tearDown()` functions in JUnit. Test steps inside a execution hook are run before or after test cases depending on the nature of the hook. The example below shows the order of execution in RSpec:

```
describe "Execution Order Demo" do  
  include RWebSpec::RSpecHelper  
  
  before(:all) do  
    puts "Calling before(:all)"  
  end  
  
  before(:each) do  
    puts "  Calling before(:each)"  
  end  
  
  after(:each) do  
    puts "  Calling after(:each)"  
  end  
  
  after(:all) do  
    puts "Calling after(:all)"  
  end  
end
```

```

end

it "First Test Case" do
  puts "    In First Test Case"
end

it "Second Test Case" do
  puts "    In Second Test Case"
end

end

```

Output

```

Calling before(:all)
  Calling before(:each)
    In First Test Case
  Calling after(:each)
  Calling before(:each)
    In Second Test Case
  Calling after(:each)
Calling after(:all)

```

What is the use of execution hooks? Let's look at the test script below (*the test script is in RWebSpec, an extension of Selenium WebDriver. please examine the structure of test scripts rather than test statement syntax, for now*). There are three login related test cases in a single test script file.

```

describe "User Login" do
  include TestHelper # defined functions such as open_browser, login_as

  it "Can login as Registered User" do
    open_browser
    login_as("james", "pass")
    expect(page_text).to include("Welcome James")
    logout
    close_browser
  end

  it "Can login as Guest" do
    open_browser
    login_as("guest", "changeme")
    expect(page_text).to include("Login OK")
  end
end

```

```
    logout
    close_browser
end

it "Can login as Administrator" do
  open_browser
  login_as("admin", "secret")
  assert_link_present_with_text("Settings")
  logout
  close_browser
end

end
```

By utilizing execution hooks, we can refine these test cases to:

```
describe "User Login" do
  include TestHelper

  before(:all) do
    open_browser
  end

  after(:each) do
    logout
  end

  after(:all) do
    close_browser
  end

  it "Can login as Registered User" do
    login_as("james", "pass")
    expect(page_text).to include("Welcome James")
  end

  it "Can login as Guest" do
    login_as("guest", "changeme")
    expect(page_text).to include("Login OK")
  end

  it "Can login as Administrator" do
    login_as("admin", "secret")
    assert_link_present_with_text("Settings")
  end
end
```

end

end

By utilizing RSpec's `before(:all)`, `after(:each)` and `after(:all)` hooks, this version is not only more concise, more importantly, every test case is now more focused (distinguished from each other). Using these hooks effectively will make test scripts more readable and easier to maintain. For readers who are new to RSpec, don't worry, I will cover it more in later chapters.

Cucumber

Cucumber, another relatively new BDD framework in Ruby, is gaining popularity rapidly. To avoid distractions, we will focus on test practices using Selenium-WebDriver + RSpec. There will be a dedicated chapter on Cucumber towards the end of this book.

3.4 Run tests from command line

In Chapter 2, we created an automated test script using a recorder and ran the test from TestWise.

One advantage of open-source test frameworks, such as Selenium WebDriver, is freedom. You can edit the test scripts in any text editor and run them from command line.

You need to install Ruby first, then install RSpec and preferred web test driver and library (called Gem in Ruby). Basic steps are:

- install Ruby interpreter
 - Windows installer: <http://rubyinstaller.org>
 - Mac: pre-installed with OS
 - Linux: get from package manager or compile from source
- install RSpec
 - > *gem install rspec*
- install test framework gem(s)
 - > *gem install selenium-webdriver*

For windows users, especially the ones who have difficulty installing gems behind a corporate proxy, you may simply download and install free pre-packaged RubyShell (based on Ruby Windows Installer) at <http://testwisely.com/testwise/downloads>.

Once the installation (takes about 1 minute) is complete, we can run a RSpec test from command line. you need to have some knowledge on typing commands in console (called Command on Windows).

To run test cases in a test script file, enter command

```
> rspec google_spec.rb
```

Run multiple test script files in one go:

```
> rspec first_spec.rb second_spec.rb
```

Run individual test case in a test script file, supply a line number in chosen test case range.

```
> rspec google_spec.rb:30
```

To generate a test report (HTML) after test execution:

```
> rspec -fh google_spec.rb > test_report.html
```

The command syntax is the same for Mac OS X and Linux platforms.

4. TestWise - Functional Testing IDE

In Chapter 2, we wrote a simple automated test case using TestWise, a functional testing Integration Development Environment (IDE). Selenium WebDriver test scripts can be developed in any text-based editors or IDEs. You can safely skip this chapter if you had decided the tool. Readers, who want to be more productive with TestWise, might find this chapter useful.

4.1 Philosophy of TestWise

The Philosophy of TestWise:

- **“The Power of Text”**
(inspired from the classic book Pragmatic Programmers)
- **“Convention over Configuration”**
(inspired from popular Ruby on Rails framework)
- **Simplicity**

The Power of Text

Unlike some testing tools, the main window of TestWise is a text-based editor, with various testing functions such as test execution, test refactoring, test navigation, etc. The benefits of using plain text (test scripts):

- Use of Source Control system to track revision and compare differences
- Powerful text editing, such as Snippets
- Search and replace, even across multiple files in project scope
- Refactoring (we will cover this in later chapter)
- Easy view or edit using any text editors without dependency on proprietary tool

Convention over Configuration

The principle of “Convention over Configuration” is gaining more acceptance with the success of Ruby on Rails framework. It makes sense for writing automated tests as well. In the context of testing, with conventions in place, when a tester opens a new test project, she/he should feel comfortable and can get to work straight way.

TestWise defines simple conventions for the test project structure, test file naming and page classes, as you will see later in this chapter. This helps communication among team members or seeking help externally when necessary.

Simplicity

TestWise is designed from the ground up to suit testers, without compromises often found in testing tools that are based on programming IDEs (which are designed for programmers). Every feature in TestWise has one single purpose: a better testing experience.

To make new-to-automation testers more willing to adopt, TestWise is designed to be easy to install, launch quickly and get you started in minutes.

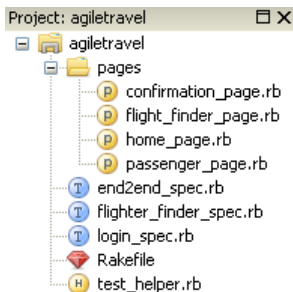
Next-Generation Functional Testing Tool

In October 2007, The Agile Alliance held a Functional Testing Tools Visioning Workshop to envision the next-generation of functional testing tools: “We are lacking integrated development environments that facilitate things like: refactoring test elements, command completion, incremental syntax validation (based on the domain specific test language), keyboard navigation into the supporting framework code, debugging, etc.” [AAFTTVW 07]




TestWise was designed and implemented before the workshop, but shares the same vision.

4.2 TestWise project structure

The project structure in TestWise is simple.



There are several file types distinguished by icons in TestWise:

-  **Test script files** (xxx_spec.rb)
One test script file may contain one or more test cases (the extension '.rb' means it is a Ruby script file).
-  **Page class files** (xxx_page.rb under /pages folder)
Page class are for reusable Ruby class representing a web page, we will cover it in detail in the next chapter.
-  **Test Helper** (test_helper.rb)
Common reusable functions are defined in Test Helper. It is included at the beginning of all test script files and the functions are available for all test scripts.
- **Project file** (xxx.tpr)
Store project settings. To open a TestWise project, look for a xxx.tpr file
- **Rakefile**
Configuration file for Rake build language (equivalent to build.xml for Ant), which can be used to execute all or a custom suite of test cases.
- **Test data** (under /testdata folder, optional)
The place to put your test data.

4.3 Test execution

Test execution, obviously, is the most important feature for testing tools. TestWise offers several ways to run tests.

Run test cases in a test script file (F10)

A test script file may contain one or more test cases which commonly form a logic group.

Run individual test case (Shift+F10)

When developing or debugging (trying to find out what went wrong) a new test case, you can just run this single test case and leave the web browser at the state when an error occurred for analyse. And yes, this is the most frequently used method for executing tests.

Run All Tests in folder

Also you can run all tests under a folder.

Run selected tests: Test Suite

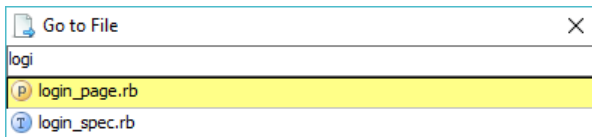
A Test Suite is a group of selected test script files to allow a custom set of test cases to be executed together.

4.4 Keyboard navigation

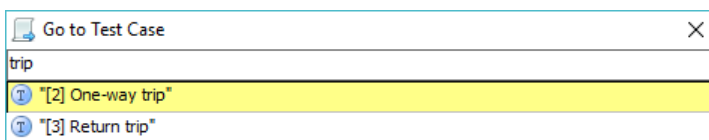
One criteria identified by Agile Alliance work for Next-Gen Functional Testing tools is “keyboard navigation into the supporting framework code”. Those who are used to operating with a mouse all the time might find ‘keyboard navigation’ is just a matter of personal preference, and wonder how it is made into the list?

For any projects that are doing serious automated testing, there will be a large number of test scripts. When the number is big, being able to find the test case quickly (which at the ‘fast end’ of spectrum, means via the keyboard), keyboard navigation becomes more than just a convenience.

Go to Test Script File (Ctrl+T)



Go to Test Case (Ctrl+Shift+T)



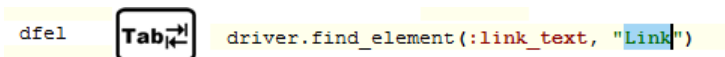
Rocky's mouse

Once I worked with a tester nicknamed Rocky who was in his fifties. Despite many doubts, he fell in love with automated testing quickly. He developed RSI (Repetitive Strain Injury, a potentially disabling illness caused by prolonged repetitive hand movements) with his mouse hand. Certainly years of the using computer mice had contributed to that. When

we worked together on test cases, I moved the mouse to the far right side and sometimes even put a piece of paper between him and the mouse. Changing a habit is never easy, but Rocky was doing admirably well. Weeks later, Rocky used the keyboard more than the mouse and felt more productive as a result. Months later after I left the project, I met one of his colleagues, who told me: he saw Rocky once snapped the mouse on his desk, and said to himself: *“Zhimin said not to use it”*.

4.5 Snippets

Snippets in TestWise are small bits of text that expand into full test script statements. The use of snippets helps to create test steps more effectively when crafted manually. For example, type ‘cl’ then press Tab key in a script editor, TestWise will expand it into the test statement below (clicking a hyperlink):

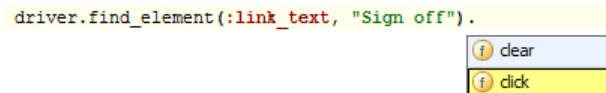


```
dfel driver.find_element(:link_text, "Link")
```

There are two ways to insert a snippet:

- Enter snippet abbreviation and press Tab key, or
- Press ‘Ctrl+J’ and select from the list, or type to narrow down the selection.

After a snippet is expanded, you may type over the highlighted text and press Tab to move to next one if there is any. For example, type “Sign off” then press Tab key, the cursor will move to the end of line. Type . and select click to complete this test statement.



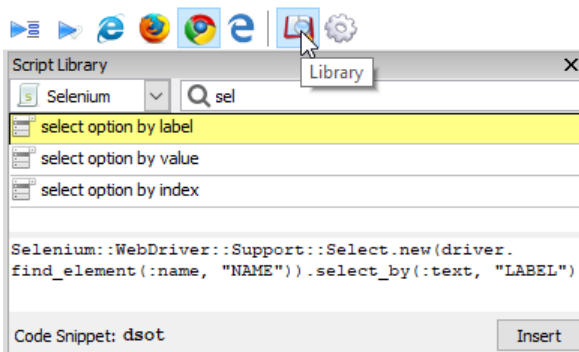
```
driver.find_element(:link_text, "Sign off").
```

clear

click

4.6 Script library

For testers who are new to the test framework and do not know the script syntax, may have many ‘how-to’ questions such as: What is the test script syntax for clicking a button?, How to assert the check box is checked?, etc. TestWise’s built-in script library can provide the answers.



4.7 Test refactoring

Test Refactoring is a process of refining test scripts to make it easier to read, and more importantly, easier to maintain. One unique feature of TestWise is its refactoring support, performing test refactoring efficiently and reliably.

We will cover this important topic in later chapters.

4.8 Wrap up

We have quickly introduced some features of TestWise to help you develop test scripts more efficiently. For more details, please check TestWise online documentation and screencasts.

5. Case Study

In this chapter, we will write six automated tests for a test web site.

5.1 Test site

In this over-simplified flight booking system: Agile Travel (<http://travel.agileway.net>¹), there are 4 high level functions on 4 web pages:

Sign in → Select flights → Enter passenger details → Pay by credit card (then check confirmation)

Some may think the confirmation should have its own page. That's correct, however, I combined the payment page and the confirmation page for testing AJAX.

We are going to write several test cases to verify core functions below:

- Sign in
- Select flights
- Enter passenger details
- Pay by credit card

We will create four test script files, inside which are test cases that are dedicated to testing each core function.

I suggest you spend a few minutes playing with this web site to get familiar to it, as you do for your work.

5.2 Preparation

The automated test framework used in this case study are Selenium WebDriver + RSpec, and automated tests will be executed in Chrome.

¹<http://travel.agileway.net>

Web Site: <http://travel.agileway.net>
Test user login: [agileway/testwise](#)
Platform: Chrome on Windows 7 or later, or Mac
Software to be installed: TestWise IDE (any edition), Chrome with TestWise Recorder plug-in

5.3 Create test project

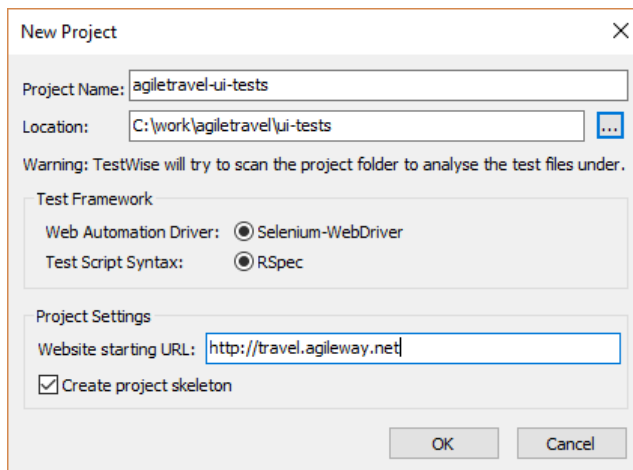
Objective

- Create a test project in TestWise

Assume there is an existing folder `c:\work\agiletravel`, we can add a folder `ui-tests` under it to store our automated test scripts.

In TestWise, select menu 'File' → 'New Project' (close the existing project first if there is one), specify

- name of test project
- test project folder
- select “**Selenium-Webdriver**” for web automation driver and “**RSpec**” for syntax.
- web site URL



If you want to open this project in TestWise later, select menu 'File' → 'Open Project', locate the project file `agiletravel-ui-tests.tpr` under `c:\work\agiletravel\ui-tests` folder.

5.4 Test Suite: Sign in

Objective

- Create test cases using a recorder
- Multiple test cases in same test script file
- Analyse test error
- Understand test execution interference

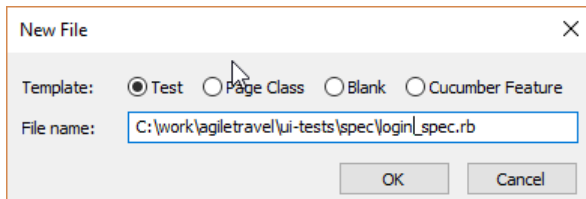
Test Design

We start with two simple and common test cases: one positive and one negative

- Sign in OK
- Sign in failed with invalid password

Positive Case: User can sign in OK

Select menu 'File' → 'New File', enter file name as 'login_spec.rb'



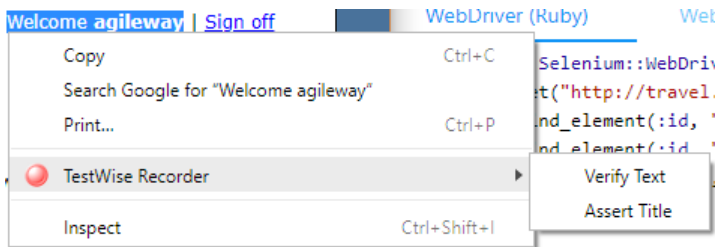
A test case skeleton is created in newly created test script file *login_spec.rb*:

```
it "New Test Case" do
  # Test Steps go here
end
```

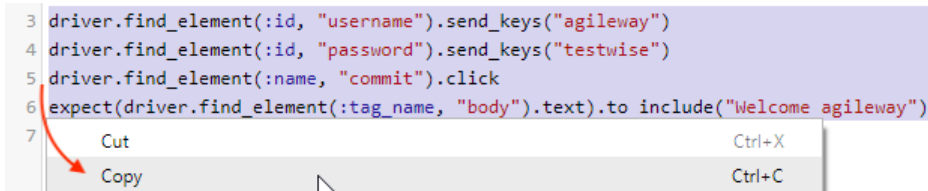
Set test case name by changing the text “New Test Case” to “User can sign in OK”.

Start Chrome browser, navigate to our test site URL: <http://travel.agileway.net>, and enable TestWise Recorder by clicking its icon on the toolbar. In Chrome, enter user name and password (*agileway/testwise*), and click ‘Sign in’ button.

A test case is not complete without checks. We could use the presence of the text ‘Welcome (username)’ as the determination of a user is signed in successfully. To create this assertion step, highlight “Welcome XXX” text, right click and select ‘Add verifyText for ...’.



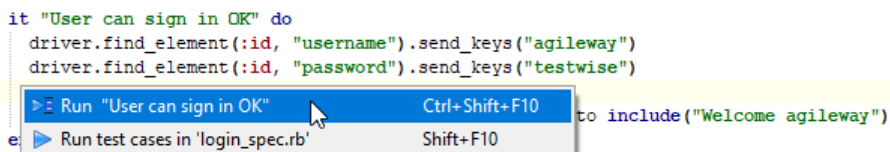
Now right click in the recorder window and 'Copy all' recorded test steps:



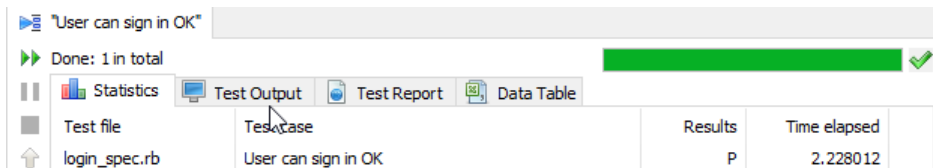
Paste recorded test steps in the test case in TestWise. Now we get:

```
it "User can sign in OK" do
  driver.find_element(:id, "username").send_keys("agileway")
  driver.find_element(:id, "password").send_keys("testwise")
  driver.find_element(:name, "commit").click
  expect(driver.find_element(:tag_name, "body").text).to include("Signed in!")
end
```

Run the test (right click any line within the test case and select *Run "User can sign in OK"*)



It passed! (indicated by the green tick)



Negative Case: User failed to sign in due to invalid password

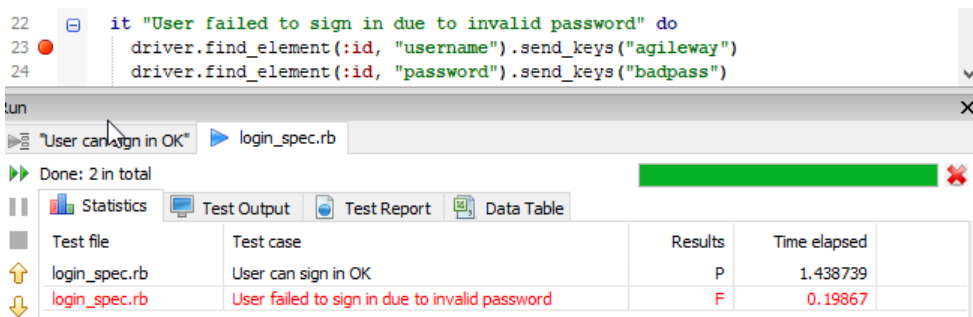
Now we continue to add another login related test case in *login_spec.rb*: user failed to sign in with invalid password. By using TestWise Recorder, we can quickly create this negative test case as below:

```
it "User failed to sign in due to invalid password" do
  driver.find_element(:id, "username").send_keys("agileway")
  driver.find_element(:id, "password").send_keys("badpass")
  driver.find_element(:name, "commit").click
  the_page_text = driver.find_element(:tag_name, "body").text
  expect(the_page_text).to include("Invalid email or password")
end
```

Run this test case, it shall pass.

Run all test cases in the login_spec.rb

Now click ▶ on the tool bar to run the two test cases in *login_spec.rb*. The second test case failed, but it runs fine by itself. Let's examine. In TestWise, the error occurred on line 23.



The screenshot shows the TestWise interface with the following components:

- Code Editor:** Lines 22-24 of *login_spec.rb* are visible. Line 23 has a red dot indicating an error.
- Test Run Summary:** Shows "Done: 2 in total".
- Test Results Table:**

Test file	Test case	Results	Time elapsed
login_spec.rb	User can sign in OK	P	1.438739
login_spec.rb	User failed to sign in due to invalid password	F	0.19867

Clicking the 'Test Output' tab, error trace tells us that the element with id "username" could not be located:

The screenshot shows the TestWise Test Output window. At the top, it says "Done: 2 in total" with a green progress bar. Below the title bar are tabs for "Statistics", "Test Output", "Test Report", and "Data Table". The "Test Output" tab is active, showing a failure summary and details. On the left side of the window, there are icons for "Up", "Down", and "Close". On the right side, there are buttons for "Styled" and "Raw".

```
.F
Failures:

1) Test Suite User failed to sign in due to invalid password
Failure/Error: bridge.find_element_by by, what.to_s, ref

Selenium::WebDriver::Error::NoSuchElementException:
no such element: Unable to locate element: {"method":"id","selector":"username"}
(Session info: chrome=67.0.3396.99)
(Driver info: chromedriver=2.37.544315 (730aa6a5fdb59ac9f4c1e8cbc59bfb5ce12b7), plat
# C:/work/agiletravel/ui-tests/spec/login_spec.rb:23
```

To debugging test scripts for Web applications, the number one rule is to keep the browser open and inspect after test execution. TestWise does this automatically when running one test case. For running all test cases in a test script file, by default, the test script closes the browser it started. This is necessary as we don't want to see many browser windows when running a suite test scripts. Back to our problem, we can simply (and temporarily) comment out the test statement of closing browser (in `after(:all)`).

```
after(:all) do
  # @driver.quit unless debugging?
end
```

Run the test script (both test cases) again. This time, we see the page showing in the browser is the one after signing in successfully, as the result of executing the first test case. Our second test case was expecting the home page to enter a user name in a text box. Well, since the current page is not the home page, the test failed.

← → ↻ travel.agileway.net/flights/start ☆

Signed in! Welcome **agileway** | [Sign off](#)

Select Flight

Trip type: ☒ Return ☐ One way

From:

To:

Departing:

Returning:

How can you prevent execution of the first test case from affecting the second one? One solution is to add a 'sign off' step: `driver.find_element(:link_text, "Sign off").click` at the end of the first test case.

```
it "User can sign in OK" do
  driver.find_element(:id, "username").send_keys("agileway")
  driver.find_element(:id, "password").send_keys("testwise")
  driver.find_element(:name, "commit").click
  expect(driver.find_element(:tag_name, "body").text).to include("Signed in!")
  driver.find_element(:link_text, "Sign off").click
end
```

```
it "User failed to sign in due to invalid password" do
  driver.find_element(:id, "username").send_keys("agileway")
  driver.find_element(:id, "password").send_keys("badpass")
  driver.find_element(:name, "commit").click
  the_page_text = driver.find_element(:tag_name, "body").text
  expect(the_page_text).to include("Invalid email or password")
end
```

Both test cases should pass now. Don't forget to uncomment the line `# @driver.quit` unless debugging? to close the browser after the test execution completes.

5.5 Test Suite: Select Flights

Objective

- Verify text across pages
- Check dynamic page

Test Case Design

There are quite a few scenarios we could write tests for on this page. In this exercise, we will write two:

- A one-way trip
- A return trip

Case 1: One-way trip

Create a new test script file: *flight_spec.rb*.

```
it "One-way trip" do
  driver.find_element(:id, "username").send_keys("agileway")
  driver.find_element(:id, "password").send_keys("testwise")
  driver.find_element(:name, "commit").click
  driver.find_element(:xpath, "//input[@type='radio' and @name='tripType' and @value='one\
way']").click
  elem_from = driver.find_element(:name, "fromPort")
  Selenium::WebDriver::Support::Select.new(elem_from).select_by(:text, "Sydney")
  elem_to = driver.find_element(:name, "toPort")
  Selenium::WebDriver::Support::Select.new(elem_to).select_by(:text, "New York")
  elem_depart_day = driver.find_element(:id, "departDay")
  Selenium::WebDriver::Support::Select.new(elem_depart_day).select_by(:text, "02")
  elem_depart_month = driver.find_element(:id, "departMonth")
  Selenium::WebDriver::Support::Select.new(elem_depart_month).select_by(:text, "May 2016")
  driver.find_element(:xpath, "//input[@value='Continue']").click

  # page_text defined in helper: driver.find_element(:tag_name, "body").text
  expect(page_text).to include("2016-05-02 Sydney to New York")
end
```

Case 2: Return trip

Still in *flight_spec.rb*, add the second test case: return trip.

```

it "Return trip" do
  driver.find_element(:id, "username").send_keys("agileway")
  driver.find_element(:id, "password").send_keys("testwise")
  driver.find_element(:name, "commit").click
  driver.find_element(:xpath, "//*[@name='tripType' and @value='return']").click
  elem_from = driver.find_element(:name, "fromPort")
  Selenium::WebDriver::Support::Select.new(elem_from).select_by(:text, "Sydney")
  elem_to = driver.find_element(:name, "toPort")
  Selenium::WebDriver::Support::Select.new(elem_to).select_by(:text, "New York")
  elem_depart_day = driver.find_element(:id, "departDay")
  Selenium::WebDriver::Support::Select.new(elem_depart_day).select_by(:text, "02")
  elem_depart_month = driver.find_element(:id, "departMonth")
  Selenium::WebDriver::Support::Select.new(elem_depart_month).select_by(:text, "May 2016")
  elem_return_day = driver.find_element(:id, "returnDay")
  Selenium::WebDriver::Support::Select.new(elem_return_day).select_by(:text, "04")
  elem_ret_month = driver.find_element(:id, "returnMonth")
  Selenium::WebDriver::Support::Select.new(elem_ret_month).select_by(:text, "June 2016")
  driver.find_element(:xpath, "//*[@value='Continue']").click

  expect(page_text).to include("2016-05-02 Sydney to New York")
  expect(page_text).to include("2016-06-04 New York to Sydney")
end

```



For avoid wrapping in text, I changed long dropdown selection test step like the one below:

```

Selenium::WebDriver::Support::Select.new(driver.find_element(:name, "fromPort")).\
select_by(:text, "Sydney")

to

elem_from = driver.find_element(:name, "fromPort")
Selenium::WebDriver::Support::Select.new(elem_from).select_by(:text, "Sydney")

```

These two versions are equivalent. Personally, I prefer the first one, as I generally like one test step line for a user operation.

You might notice the step below wasn't included in the recorded test steps.

```

driver.find_element(:xpath, "//*[@type='radio' and @name='tripType' and @value='return']").click

```

This is because this radio button was already pre-selected. You may skip this step. I added this step as I want to make sure this radio button is selected. To record this step, you

- click 'One way' radio button
- right click in the recorder to clear test steps
- click 'Return' radio button

Or you could try inspecting the HTML source manually (see 'Identify Web Controls' section in Chapter 3).

You may want to add 'sign off' steps to make both the test cases work. But there is another easier and cleaner way.

Technique: use execution hooks

You might have noticed that both test cases start with same 3 sign-in test steps and end with a sign off test step. If we think about it, we don't have to test the functionality of signing in and signing off for each test case. In fact, our focus is to test the different scenarios after signed in.

With the knowledge of RSpec, we can move these 3 test steps into a '*before(:all)*' execution hook. This way, we only need to sign in once regardless of how many test cases in this test script file.

```
before(:all) do
  @driver = Selenium::WebDriver.for(:chrome)
  driver.navigate.to(site_url)
  driver.find_element(:id, "username").send_keys("agileway")
  driver.find_element(:id, "password").send_keys("testwise")
  driver.find_element(:name, "commit").click
end

after(:all) do
  @driver.quit unless debugging?
end

it "One-way trip" do
  driver.find_element(:xpath, "//input[@type='radio' and @name='tripType' and @value='one\
way']").click
  # ...
end

it "Return trip" do
  driver.find_element(:xpath, "//input[@type='radio' and @name='tripType' and @value='ret\
urn']").click
```


The HTML fragment `<div id="returnTrip">` is the section that will be hidden when the 'One way' radio button is clicked.

Add the assertion to the test script.

```
expect(driver.find_element(:id, "returnTrip").displayed?).to be_falsey
```

The complete test case:

```
it "One-way trip" do
  driver.find_element(:xpath, "///input[@name='tripType' and @value='oneway']").click
  expect(driver.find_element(:id, "returnTrip").displayed?).to be_falsey
  elem_from = driver.find_element(:name, "fromPort")
  Selenium::WebDriver::Support::Select.new(elem_from).select_by(:text, "Sydney")
  elem_to = driver.find_element(:name, "toPort")
  Selenium::WebDriver::Support::Select.new(elem_to).select_by(:text, "New York")
  elem_depart_day = driver.find_element(:id, "departDay")
  Selenium::WebDriver::Support::Select.new(elem_depart_day).select_by(:text, "02")
  elem_depart_month = driver.find_element(:id, "departMonth")
  Selenium::WebDriver::Support::Select.new(elem_depart_month).select_by(:text, "May 2016")
  driver.find_element(:xpath, "///input[@value='Continue']").click

  expect(page_text).to include("2016-05-02 Sydney to New York")
end
```

5.6 Enter passenger details

Objective

- Validation
- Assert value in a text field

Test Design

For the passenger page, a business rule states that a last name must be provided. We could create a separate test case for each validation, however, this will be an overkill. We can simply add the validation within the main stream test case. That is,

- submit the form without entering last name
- verify the validation error message
- enter first name and last name
- submit the form

- verify the passenger's name is saved

Must provide last name

Passenger Details

Flights (oneway trip)

2012-01-01 **New York** to **Sydney**

Passenger details

First name:

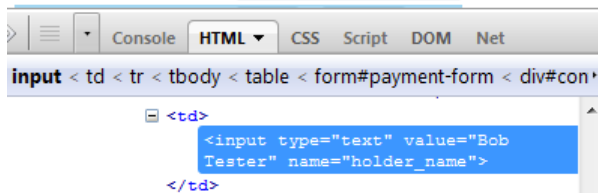
Last name:

If the passenger's details are saved properly, the full name is pre-populated as card holder name on the credit card page. We could use this as our check, i.e. getting value of text box with name "holder_name".

Card type: ☐ Visa ☐ Master

Card holder's name:

Card number:



Now we get the test scripts from the recorder.

it "Can enter passenger details" do

```
driver.find_element(:xpath, "//*[@name='tripType' and @value='oneway']").click
```

```
elem_from = driver.find_element(:name, "fromPort")
```

```
Selenium::WebDriver::Support::Select.new(elem_from).select_by(:text, "Sydney")
```

```
elem_to = driver.find_element(:name, "toPort")
```

```
Selenium::WebDriver::Support::Select.new(elem_to).select_by(:text, "New York")
```

```
elem_depart_day = driver.find_element(:id, "departDay")
```

```
Selenium::WebDriver::Support::Select.new(elem_depart_day).select_by(:text, "02")
```

```
elem_depart_month = driver.find_element(:id, "departMonth")
```

```
Selenium::WebDriver::Support::Select.new(elem_depart_month).select_by(:text, "May 2016")
```

```
driver.find_element(:xpath, "//*[@value='Continue']").click
```

```
# now on passenger page
driver.find_element(:xpath,"//input[@value='Next']").click
expect(driver.page_source).to include("Must provide last name")
driver.find_element(:name, "passengerFirstName").send_keys("Bob")
driver.find_element(:name, "passengerLastName").send_keys("Tester")
driver.find_element(:xpath,"//input[@value='Next']").click

expect(driver.find_element(:name, "holder_name")["value"]).to eq("Bob Tester")
end
```

The last assertion step is not from the recorder, you need type it in.


5.7 Book confirmation after payment

Objective

- AJAX Testing
- Retrieving text or value from specific element

Test Design

We navigate our way to the payment page. After filling in the credit card details and clicking on the 'Pay now' button, an animated loading image (see below) shows, indicating that the payment is being processed.

Card type:	<input type="radio"/> Visa <input checked="" type="radio"/> Master
Card holder's name:	<input type="text" value="Bob the Tester"/>
Card number:	<input type="text" value="4242424242424242"/>
Expiry in :	<input type="text" value="04"/> / <input type="text" value="2012"/>
<input type="button" value="Pay now"/> 	

After a few seconds, the flight book confirmation is displayed containing a booking number and flight details. The animated loading image disappears.



Confirmation

Booking number: 9

Flights (oneway Trip)

2012-05-02 **Sydney** to **New York**

Passenger Details: Bob Tester

Technique: Testing AJAX

I am sure that you are now quite familiar to this kind of user experience - the web page processes information and shows the results without having to refresh the whole page. The term used to describe the technology responsible for this enhanced user experience is 'AJAX'. From the testing perspective, an AJAX operation immediately 'completes' after the mouse/keyboard action (such as clicking the 'Pay now' button), no page reload is observed. After the server finished processing the request, seconds or even minutes later, some part of web page may be updated.

One simple solution for testing an AJAX operation is to wait enough time for the AJAX operation to fully complete, then perform assertions like below:

```
driver.find_element(:xpath,"//input[@type='submit' and @value='Pay now']").click
sleep 10 # wait 10 seconds
expect(driver.page_source).to include("Booking number")
```

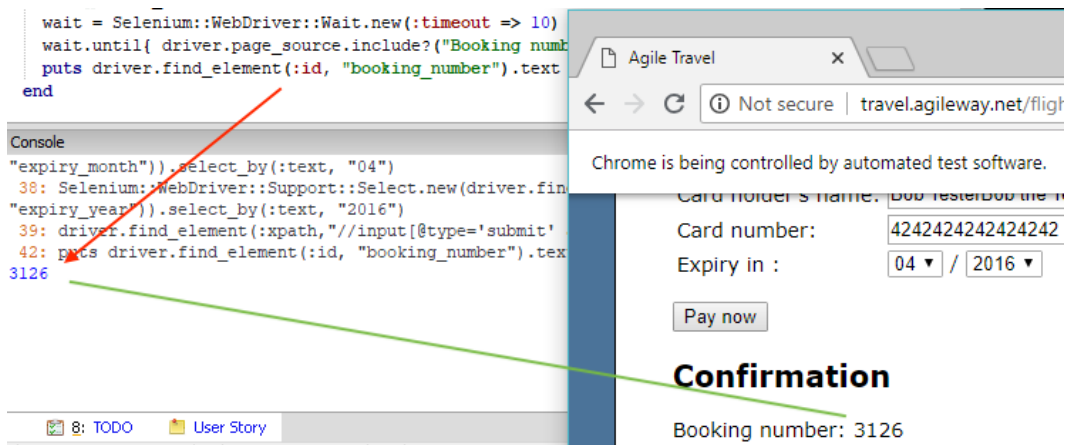
The above approach works, but is not efficient. If the AJAX operation finishes early, the test execution will still pause there and wait unnecessarily. RWebSpec introduces a convenient function *try_for(seconds) { test steps }* to keep trying next test steps every 1 second up to a specified time. If the operation was performed successfully within the given time, it moves on to the next test step. If the operation still cannot be performed after that time, an error is thrown.

```
wait = Selenium::WebDriver::Wait.new(:timeout => 10) # seconds
wait.until{ driver.page_source.include?("Booking number") }
```

Technique: Displaying value from specific HTML element in console

Sometimes it may be useful to get a value or text from a specific element on the page. For example, if the booking number in this website is in some number pattern (such as 20120228-123), we can further verify the booking number against the pattern.

When developing a test case, we often want to seek confirmation by displaying certain data (also known as printing out). For instance, a tester may want to print to the console the confirmation number from the test output. In TestWise, you can use the `puts` function to display text in the console window, as shown below:



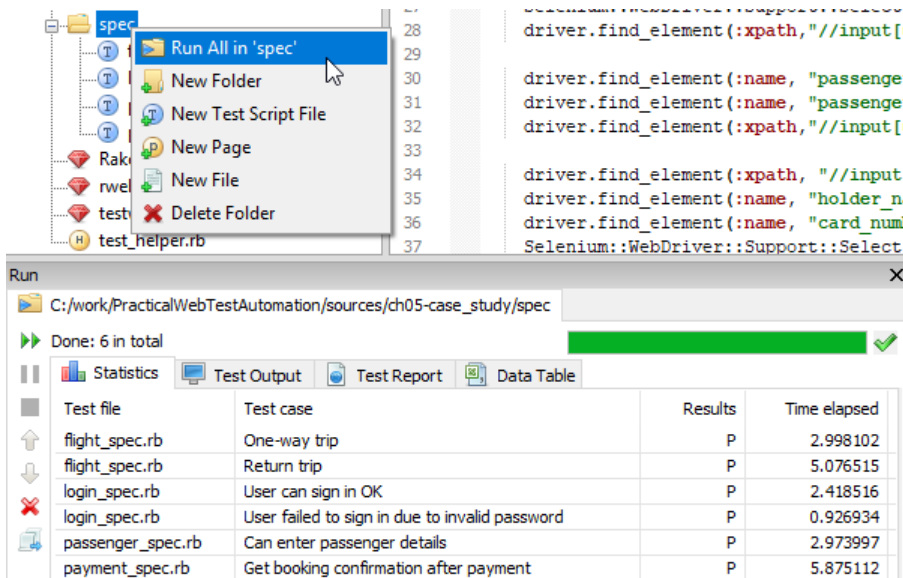
The test scripts:

```
it "Get booking confirmation after payment" do
  # ... up to payment page
  # ... enter credit card details

  driver.find_element(:xpath, "///input[@type='submit' and @value='Pay now']").click
  wait = Selenium::WebDriver::Wait.new(:timeout => 10) # seconds
  wait.until{ driver.page_source.include?("Booking number")
  puts driver.find_element(:id, "booking_number").text
end
```

5.8 Run all tests

We can run all these 6 test cases (in 4 test script files) together. Right click the project folder and select “Run All in ...”. The screenshot below is taken after a successful execution of all 6 test cases.



Run all tests

A more detailed test report can be found under ‘Test Report’ tab.

```
Test Results 6 test cases, 0 failures
              Finished in 46.19342

File: flight_spec.rb
  Specification: "Select Flights"
    Test Case: "One-way trip"
    Test Case: "Return trip"
File: login_spec.rb
  Specification: "User Login"
    Test Case: "User can sign in OK"
    Test Case: "User failed to sign in due to invalid password"
File: passenger_spec.rb
  Specification: "Passenger"
    Test Case: "Can enter passenger details"
File: payment_spec.rb
  Specification: "Payment"
    Test Case: "Get booking confirmation after payment "
```

5.9 Wrap up

We have created several automated test cases in Selenium WebDriver. Along the way, some techniques were introduced. After this exercise, you should be ready to write real tests for your project. I expect that some of you might be very excited, especially after seeing execution of a couple of real tests for your project, and think that test automation is easy.

Here I want to remind you of the test automation camel . After writing dozens of test scripts, you will face the first hump: **Hard to Maintain**. But that's OK. In Chapter 7 and 8, I will show you how to overcome that hump!