Getting Started With the IDE

Contents

Section 1- Getting Started with Selenium IDE

Test automation has grown in popularity over the years because developers do not have the time or money to invest in large test teams to make sure that applications work as they are expected. Developers also want to make sure that the code they have created works as they expect it to.

Jason Huggins saw this issue too and wanted to make sure that any system he was working on would work on multiple operating systems and browsers. So he created Selenium.

Selenium IDE is a fully-featured **IDE** (**Integrated Development Environment**) that installs as a plugin in Mozilla Firefox and enables developers to test their web applications through Selenium. With the Selenium IDE, you can record user interactions with the web browser and play them back to test for errors. It's a powerful, robust IDE that radically simplifies and automates the QA testing process.

Selenium is one of the most well-known testing frameworks in the world that is in use. It is an open source project that allows testers and developers alike to develop functional tests to drive the browser. It can be used to record workflows so that it reduces the time in regression testing. Selenium can work on any browser that supports JavaScript, since Selenium has been built using JavaScript.

In this chapter, you will learn the basics of the Selenium IDE, how to use it, and how to locate a web element on a web page. We shall cover the following topics:

- What is Selenium IDE
- Recording our first test
- Updating tests to work with AJAX sites
- Using variables in our tests
- Debugging tests
- Saving tests to be used later
- Creating and saving test suites

Understanding Selenium IDE

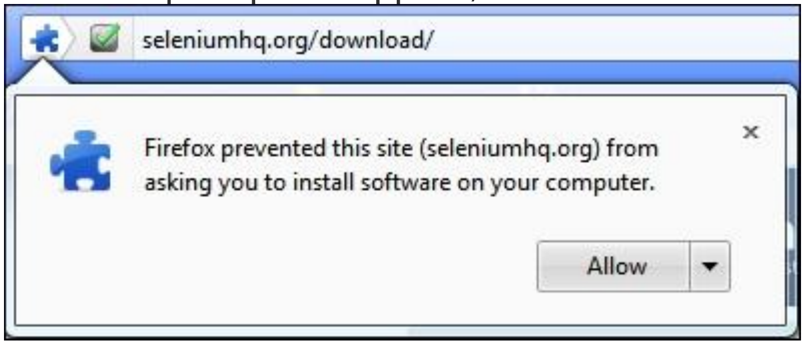
Selenium IDE is a Firefox add-on developed originally by Shinya Kasatani as a way to use the original Selenium Core code without having to copy Selenium Core onto the server. **Selenium Core** is the key JavaScript module that allows Selenium to drive the browser. It has been developed using JavaScript so that it can interact
with **DOM** (**Document Object Model**) using native JavaScript calls.

Selenium IDE was developed to allow testers and developers to record their actions as they follow the workflow that they need to test.

Installing Selenium IDE

Now that we understand what Selenium IDE is, it is a good time to install it. At the end of these steps, you will have successfully installed Selenium IDE onto your computer:

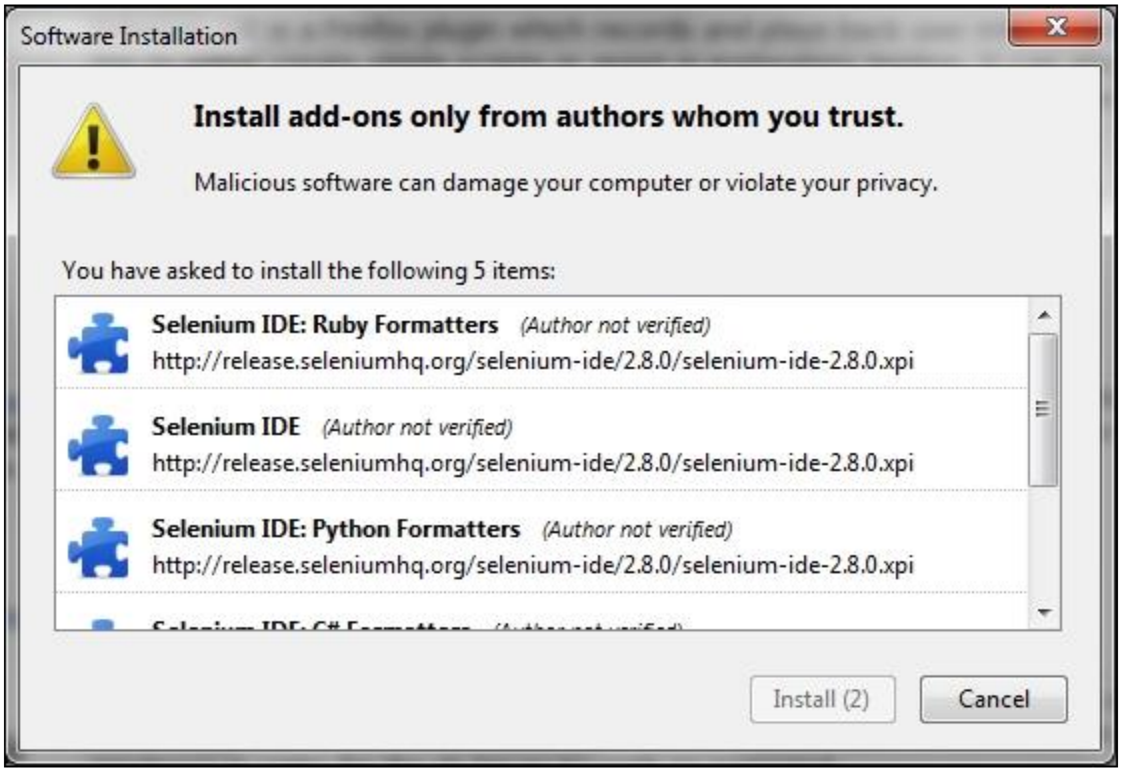
1. Go to http://seleniumhq.org/download/.
2. Click on the download link for Selenium IDE. You may see a message appear saying **Firefox prevented this site (seleniumhq.org) from asking you to install software on your computer**. If you do, click the **Allow** button.
3. A Firefox prompt will appear, as shown in the following screenshot:



seleniumhq.org/download/

Firefox prevented this site (seleniumhq.org) from asking you to install software on your computer.

4. You will then be asked if you would like to install Selenium IDE and the exporter add-ons. These have been made pluggable to the IDE by the work that Adam Goucher did. You will see a screenshot similar to the following one:


Software Installation

Install add-ons only from authors whom you trust.

Malicious software can damage your computer or violate your privacy.

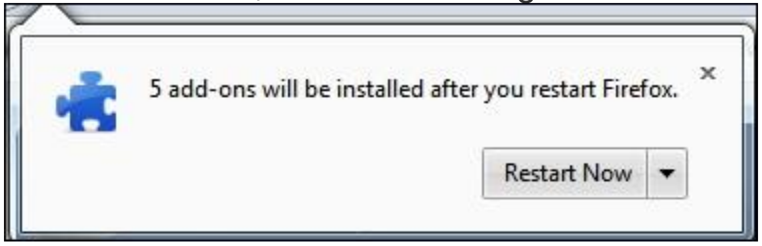
You have asked to install the following 5 items:

Selenium IDE: Ruby Formatters *(Author not verified)*
http://release.seleniumhq.org/selenium-ide/2.8.0/selenium-ide-2.8.0.xpi

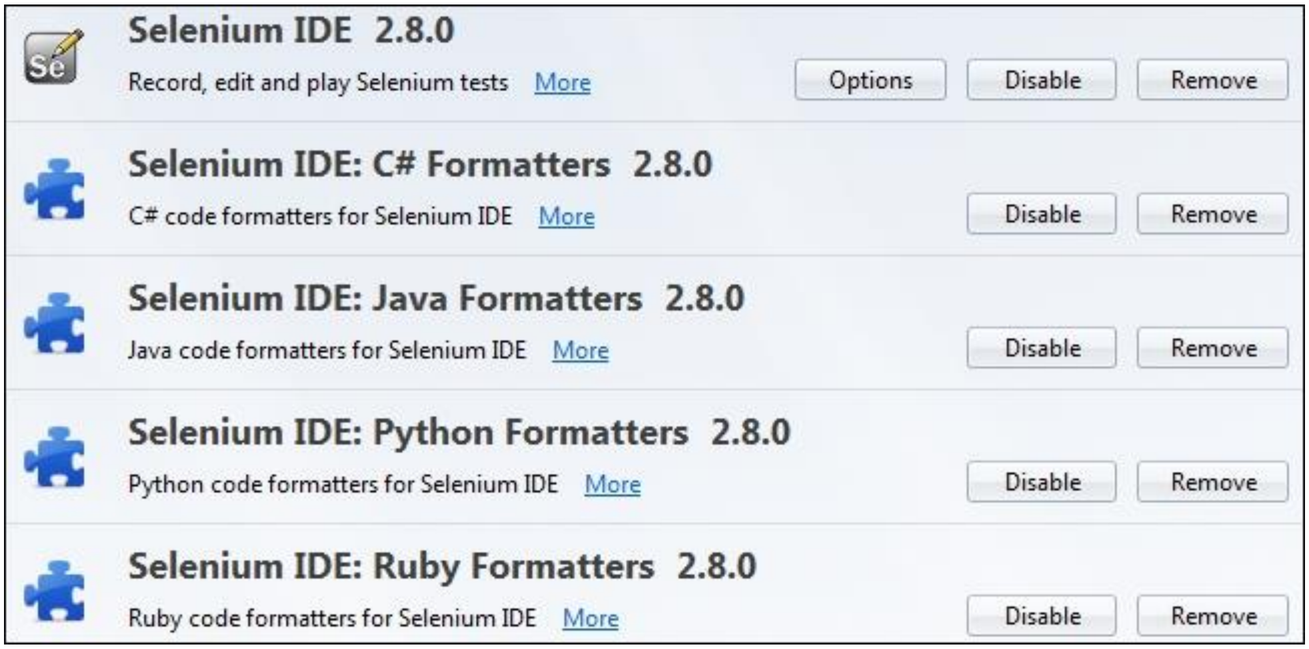
Selenium IDE *(Author not verified)*
http://release.seleniumhq.org/selenium-ide/2.8.0/selenium-ide-2.8.0.xpi

Selenium IDE: Python Formatters *(Author not verified)*
http://release.seleniumhq.org/selenium-ide/2.8.0/selenium-ide-2.8.0.xpi

5. Click on **Install** button. This will now install Selenium IDE and formatters as Firefox add-ons.
6. Once the installation process is complete, it will ask you to restart Firefox. Click the **Restart Now**button. Firefox will close and then reopen. If you have anything open in another browser, it might be worth saving your work as Firefox will try to go back to its original state. However, this cannot be guaranteed.


5 add-ons will be installed after you restart Firefox.

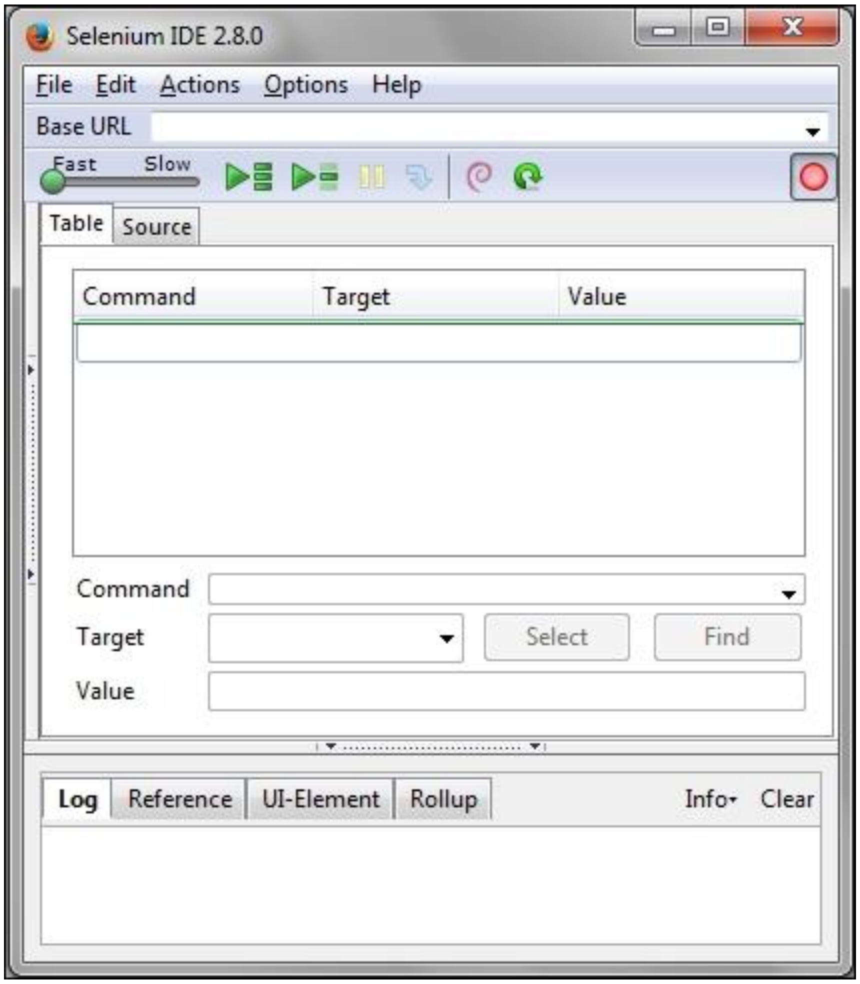
7. Once the installation is complete, the add-ons window will show the Selenium IDE and its current version:


Selenium IDE 2.8.0
Record, edit and play Selenium tests More
Selenium IDE: C# Formatters 2.8.0
C# code formatters for Selenium IDE More
Selenium IDE: Java Formatters 2.8.0
Java code formatters for Selenium IDE More
Selenium IDE: Python Formatters 2.8.0
Python code formatters for Selenium IDE More
Selenium IDE: Ruby Formatters 2.8.0
Ruby code formatters for Selenium IDE More

You have successfully installed Selenium IDE and we can start thinking about writing our first test.

Getting acquainted with the Selenium IDE tool

Now that Selenium IDE has been installed, let's take some time to familiarize ourselves with it. This will give us the foundation that we can use in later chapters. Open up Selenium IDE by going through the tools menu in Mozilla Firefox. Navigate to **Tools | Selenium IDE**. A window will appear. If the menu bar is not available, which is now the default in Firefox, you can launch Selenium IDE via **Firefox | Web Developer | Selenium IDE**.



Starting from the top, I will explain what each of the items are:

- **Base URL**: This is the URL that the test will start at. All open commands will be relative to **Base URL** unless a full path is inserted in the open command.
- **Speed Slider**: This is the slider under the **Fast** and **Slow** labels in the screenshot.
- : This **play entire test suite** icon runs all the tests in the IDE.
- : This **play current test case** icon runs a single test in the IDE.
- : This **pause/resume** icon pauses a test that is currently running.
- : This **step** icon steps through the test once it has paused.
- : This is the **record** button. This will be engaged when the test is recording.

- The **Command** drop-down list has a list of all the commands that are needed to create a test. You can type into it to use the autocomplete functionality or use it as a dropdown.
- The **Target** textbox allows you to input the location of the element that you want to work against.
- The **Find** button, once the target box is populated, can be clicked to highlight the element on the page.
- The **Value** textbox is where you place the value that needs to change. For example, if you want your test to type in an input box on the web page, you will put what you want it to type in the value box.
- The **Table** tab will keep track of all your commands, targets, and values. It has been structured this way because the original version of Selenium was styled on FIT tests. **FIT** (**Framework for Integrated Testing**) was created by Ward Cunningham. The tests were originally designed to be run from HTML files and the IDE keeps this idea for its tests.
- If you click the **Source** tab, you will be able to see the HTML that will store the test. Each of the rows will look like this:

```
<tr>
  <td>open</td>
  <td>/chapter1</td>
  <td></td>
</tr>
```

- The area below the **Value** textbox will show the Selenium log while the tests are running. If an item fails, then it will have an `[error]` entry. This area will also show help on Selenium commands when you are working in the **Command** drop-down list. This can be extremely useful when typing commands into Selenium IDE instead of using the record feature.
- The **Log** tab will show a log of what is happening during the test. The **Reference** tab gives you documentation on the command that you have highlighted, and is also useful if you forgot some command; users can just start writing command in the **Command** field, then select **some like searched** and read the reference.

Rules in creating tests with Selenium IDE

Now that we have installed Selenium IDE and understand what it is, we can think about working through our first tests. There are a few things that you need to consider when creating your first test. These rules apply to any form of test automation but need to be adhered to, especially when creating tests against a user interface:

- Tests should always have a known starting point. In the context of Selenium, this can mean opening a certain page to start a workflow.
- Tests should not have to rely on any other tests to run. If a test is going to add something, do not have a separate test to delete it. This is to ensure that if something goes wrong in one test, it will not mean you have a lot of unnecessary failures to check.
- Tests should only test one thing at a time.
- Tests should clean up after themselves.

These rules, like most rules, can be broken. However, breaking them can mean that you may run into issues later on, and when you have hundreds, or even thousands of tests, these small issues can mean that large parts of a test suite are failing.

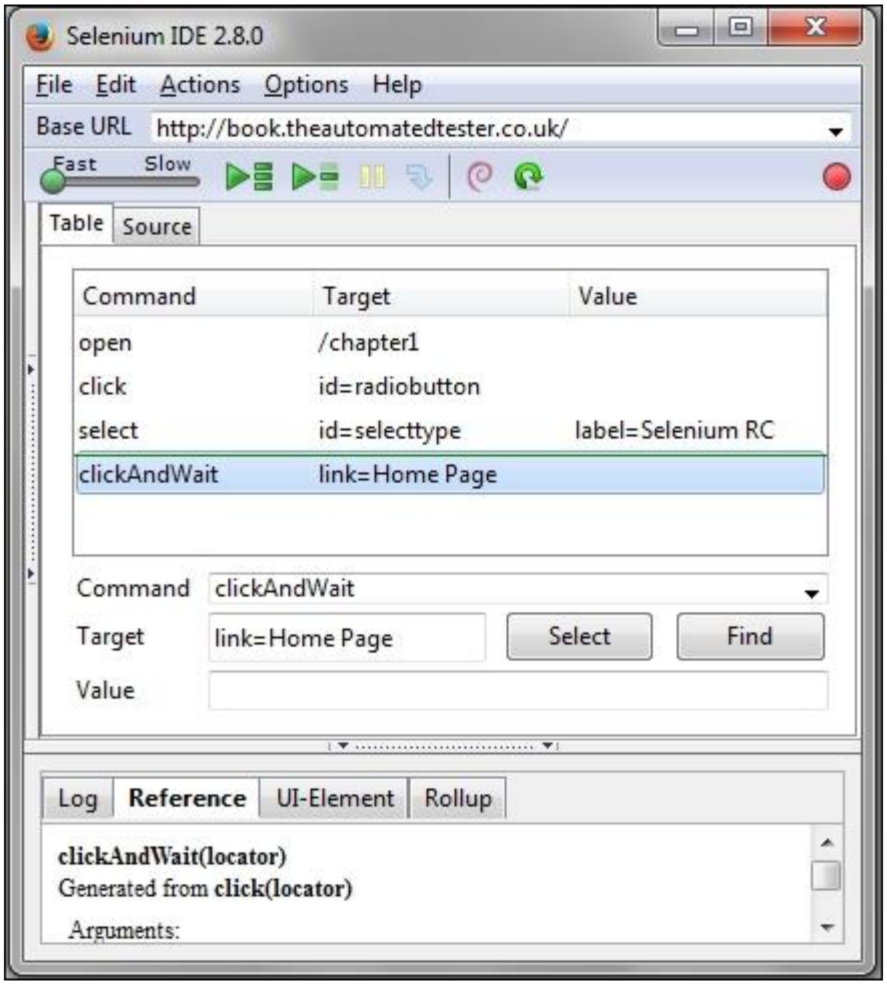
With these rules in mind, let's create our first Selenium IDE test.

Recording your first test with Selenium IDE

We will record our first test using Selenium IDE. To start recording the tests, we will need to start Mozilla Firefox. Once it has been loaded, you will need to start Selenium IDE. You will find it under the **Tools** drop-down menu in Mozilla Firefox or in the Web Developer drop-down menu. Once loaded, it will look like the next screenshot. Note that the record button is engaged when you first load the IDE.

To start recording your tests, let's do the following:

1. When in the record mode, navigate to http://book.theautomatedtester.co.uk/chapter1.
2. On the web application, do the following:
    1. Click on the record button (red-colored radio button).
    2. Select another value from the drop-down box, for example, **Selenium RC**.
    3. Click on the **Home Page** link.


Selenium IDE 2.8.0
File Edit Actions Options Help
Fast Slow
Table Source
Select
Find
Log Reference UI-Element Rollup

Base URL http://book.theautomatedtester.co.uk/

Command	Target	Value
open	/chapter1	
click	id=radiobutton	
select	id=selecttype	label=Selenium RC
clickAndWait	link=Home Page	

Command: clickAndWait

Target: link=Home Page

Value:

clickAndWait(locator)
Generated from **click(locator)**
Arguments:

3. Your test has now been recorded and should look like the preceding screenshot.
4. Click the play button that looks like this icon:
5. Once your test has completed, it will look like the above screenshot:

We have successfully recorded our first test and played it back. As we can see, Selenium IDE tried to apply the first rule of test automation by specifying the `open` command. It set the starting point of the test, in this case, **/chapter1**, and then it began stepping through the workflow that we want to record.

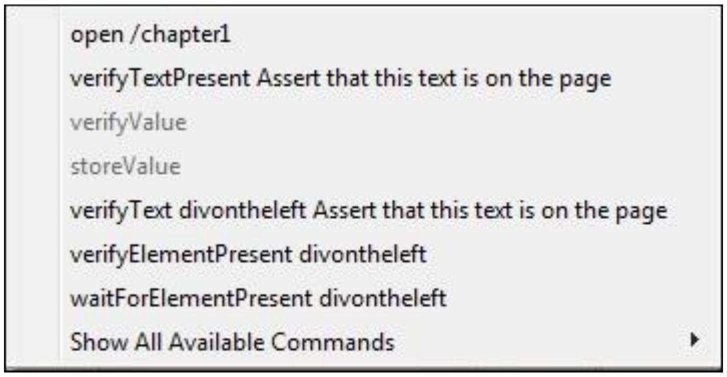
Once the actions have all been completed, you will see that all of the actions have a green background. This shows that they have completed successfully. On the left, you will see that it has completed one successful test, or run, within Selenium IDE. If you were to write a test that failed, the **Failure** label will have a **1** next to it.

Validating a test with assert and verify

In the last few steps, we were able to record a workflow that we would expect the user to perform. It will test that the relevant bit of functionality is there, such as buttons and links to work against. Unfortunately, we are not checking whether the other items on the page are there or if they are visible when they should be hidden. We will now work against the same page as before, but we shall make sure that different items are on the page.

There are two mechanisms for validating elements available on the application under test. The first is **assert**: this allows the test to check whether the element is on the page. If it is not available, then the test will stop on the step that failed. The second is **verify**: this allows the test to check if the element is on the page, but if it isn't, then the test will carry on execution. To add the assert or verify commands to the tests, we need to use the context menu that Selenium IDE adds to Firefox. All that one needs to do is right-click on the element if on Windows or Linux. If you have a Mac, then you will need to do the two-finger click to show the context menu.

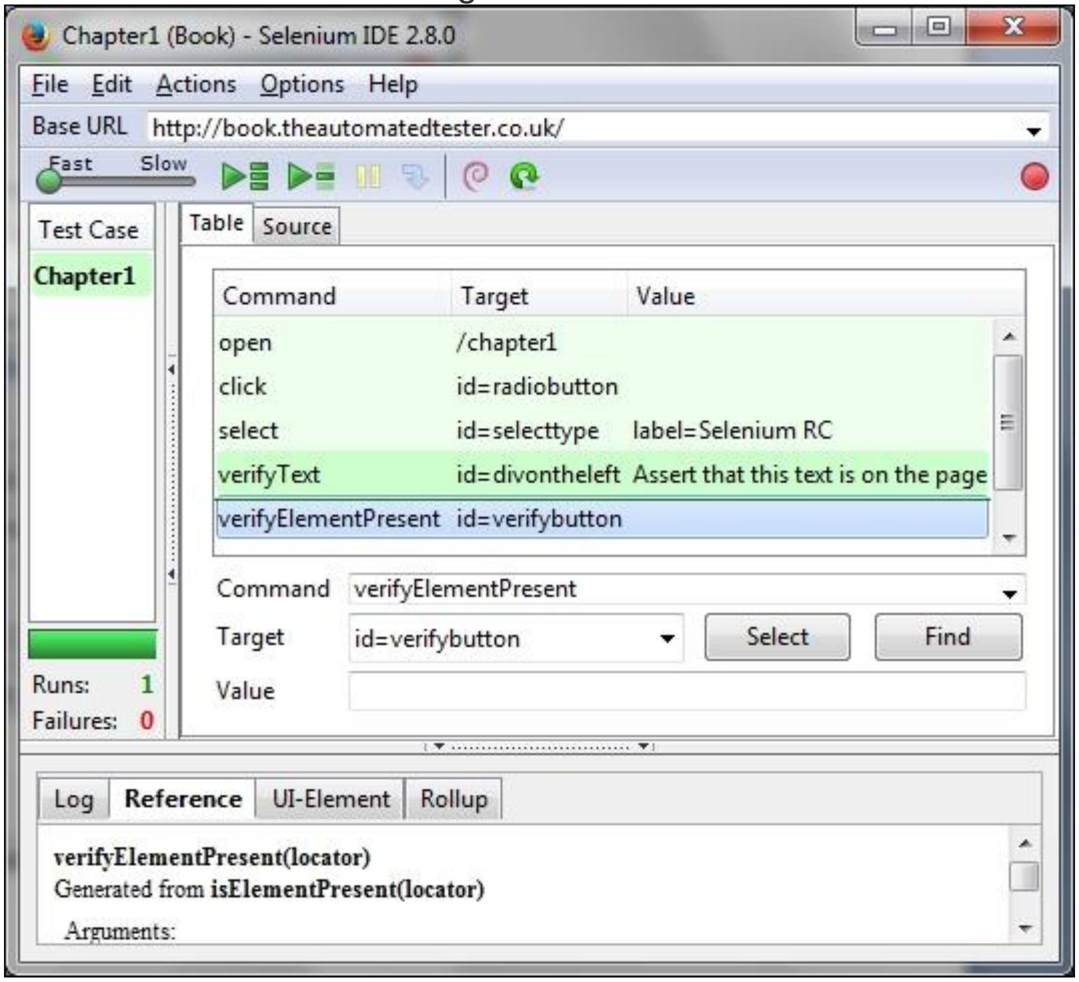
When the context menu appears, it will look roughly like the following screenshot with the normal Firefox functions above it:


open /chapter1
verifyTextPresent Assert that this text is on the page
verifyValue
storeValue
verifyText divontheleft Assert that this text is on the page
verifyElementPresent divontheleft
waitForElementPresent divontheleft
Show All Available Commands

We will record the test and learn how to use/verify some commands as follows:

1. Open the IDE so that we can start recording.
2. Navigate to http://book.theautomatedtester.co.uk/chapter1.

3. Select **Selenium Grid** from the drop-down box.
4. Change the selection to **Selenium Grid**.
5. Verify that the **Assert that this text is on the page** text is mentioned on the right-hand side of the drop-down box, by right-clicking on the text and selecting **verifyText id=diveontheleft Assert that this text is on the page**. You can see the command in the previous screenshot.
6. Verify that the button is on the page. You will need to add a new command for `verifyElementPresent` with the `verifybutton` target in Selenium IDE.
7. Now that you have completed the previous steps, your Selenium IDE should look like the following screenshot:


Chapter1 (Book) - Selenium IDE 2.8.0
File Edit Actions Options Help
Fast Slow
Test Case
Chapter1
Table Source
Select
Find
Runs: 1
Failures: 0
Log Reference UI-Element Rollup

Base URL http://book.theautomatedtester.co.uk/

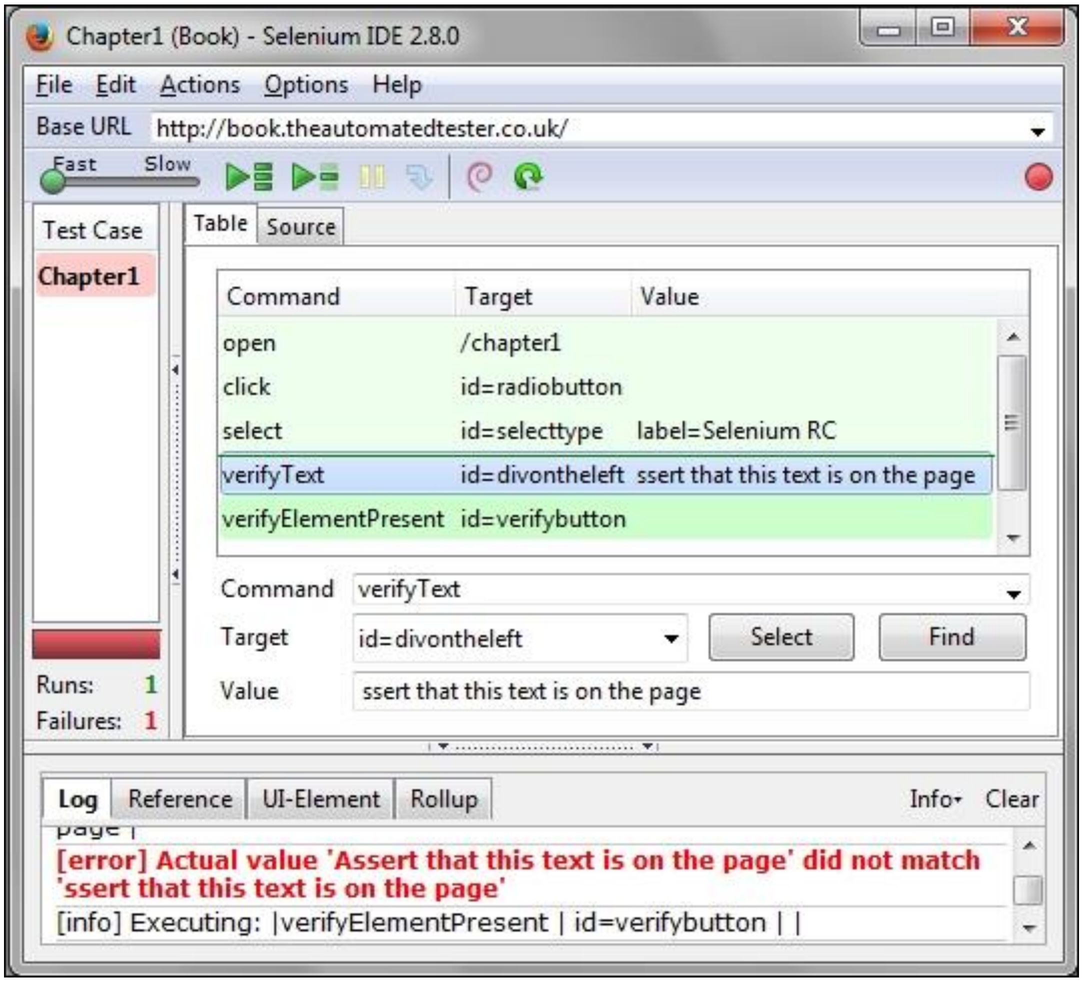
Command	Target	Value
open	/chapter1	
click	id=radiobutton	
select	id=selecttype	label=Selenium RC
verifyText	id=divontheleft	Assert that this text is on the page
verifyElementPresent	id=verifybutton	

Command verifyElementPresent
Target id=verifybutton
Value

verifyElementPresent(locator)
Generated from **isElementPresent(locator)**
Arguments:

If you now run the test, you will see it has verified that what you are expecting to see on the page has appeared. Notice that the verify commands have a darker green color. This is to show that they are more important to the test than moving through the steps. The test has now checked that the text we required is on the page and that the button is there too.

What will happen if the verify command did not find what it was expecting? The IDE would have thrown an error stating what was expected was not there, but then carried on with the rest of the test. We can see an example of this in the following screenshot:


Chapter1 (Book) - Selenium IDE 2.8.0
File Edit Actions Options Help
Base URL http://book.theautomatedtester.co.uk/
Fast Slow
Test Case
Chapter1
Table Source
Select
Find
Runs: 1
Failures: 1
Log Reference UI-Element Rollup
Info Clear

Command	Target	Value
open	/chapter1	
click	id=radiobutton	
select	id=selecttype	label=Selenium RC
verifyText	id=divontheleft	ssert that this text is on the page
verifyElementPresent	id=verifybutton	

Command: verifyText
Target: id=divontheleft
Value: ssert that this text is on the page

```
page |
[error] Actual value 'Assert that this text is on the page' did not match
'ssert that this text is on the page'
[info] Executing: |verifyElementPresent | id=verifybutton | |
```

The test would not have carried on if it was using assert as the mechanism for validating that the elements and text were loaded with the page.

We have just seen that we can add asserts or verification to the page. Selenium IDE does not do this when recording, so it will always be a manual step. The assert command will stop the running tests, unlike the verify command in which the tests will continue running even after failure. In both the cases, Selenium IDE will log an error. Each of these have their own merits.

Some of the verify and assert methods are as follows:

Selenium Command	**Command Description**
`verifyElementPresent`	This verifies an expected element is present on the page.
`assertElementPresent`	This asserts an expected element is present on the page.
`verifyElementNotPresent`	This verifies an expected element is not present on the page.
`assertElementNotPresent`	This asserts an expected element is not present on the page.
`verifyText`	This verifies expected text and its corresponding HTML tags are present on the page.
`assertText`	This asserts expected text and its corresponding HTML tags are present on the page.
`verifyAttribute`	This verifies the value of an attribute of any HTML tag on the page.
`assertAttribute`	This asserts the value of an attribute of any HTML tag on the page.
`verifyChecked`	This verifies whether the condition of the checkbox is checked or not on the page.
`assertChecked`	This asserts whether the condition of the checkbox is checked or not on the page.
`verifyAlert`	This verifies the alert present on the page.
`assertAlert`	This asserts the alert present on the page.
`verifyTitle`	This verifies an expected page title.
`assertTitle`	This asserts an expected page title.

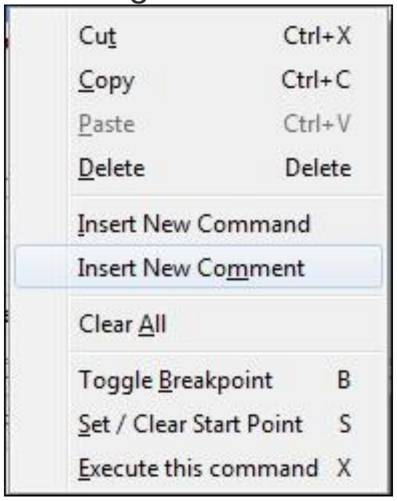
Creating comments in your tests

Before we carry on further with Selenium, this is a good time to mention how to create comments in your tests. As all good software developers know, having readable code and having comments can make maintenance in the future much easier. Unlike in software development, it is extremely hard, almost impossible, to write self-documenting code. To combat this, it is good practice to make sure that your tests have comments that future software testers can use.

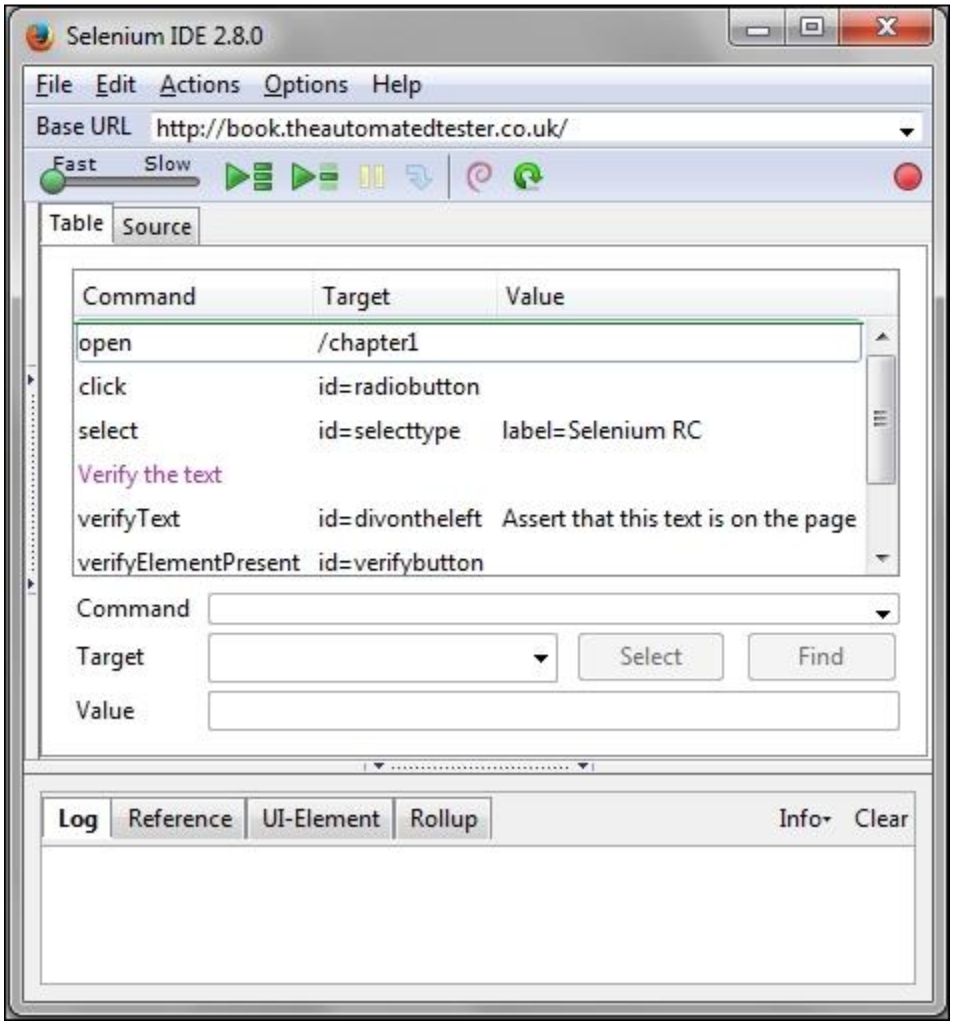
Adding Selenium IDE comments

To add comments to your tests, perform the following steps:

1. In the test that was created earlier, right-click on a step. For example, the verify step.
2. The Selenium IDE context menu will be visible as shown in the following screenshot:



3. Click on **Insert New Comment**. A space will appear between the Selenium commands.
4. Click on the **Command** textbox and enter in a comment so that you can use it for future maintenance. It will look like the following screenshot:


Selenium IDE 2.8.0
File Edit Actions Options Help
Fast Slow
Table Source
Command
Target
Select
Find
Value
Log Reference UI-Element Rollup Info Clear

Base URL http://book.theautomatedtester.co.uk/

Command	Target	Value
open	/chapter1	
click	id=radiobutton	
select	id=selecttype	label=Selenium RC
Verify the text		
verifyText	id=divontheleft	Assert that this text is on the page
verifyElementPresent	id=verifybutton	

We just had a look at how to create comments. Comments will always appear as purple text in the IDE. This, like in most IDEs, is to help you spot comments quicker when looking through your test cases. Now that we know how to keep our tests maintainable with comments, let's carry on working with Selenium IDE to record/tweak/replay our scripts.

Multiplying windows

Web applications, unfortunately, do not live in one window of your browser. An example of this can be a site that shows reports. Most reports will have their own window so that people can easily move between them.

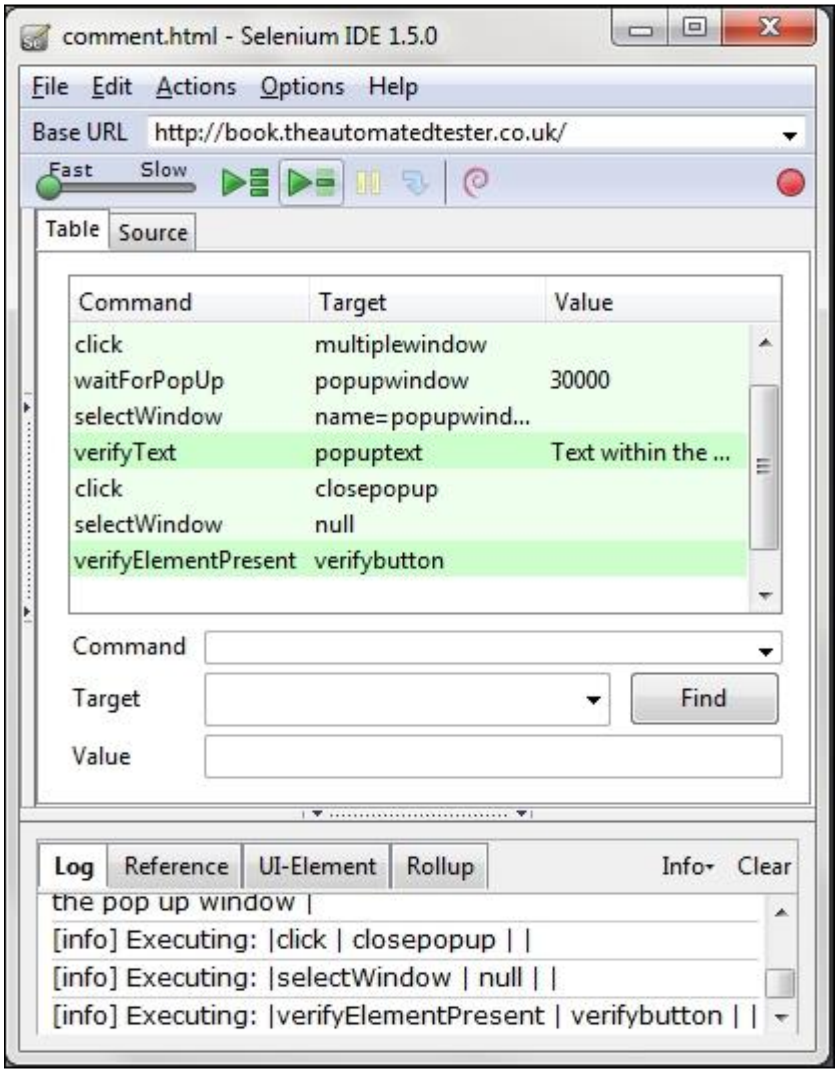
Unfortunately, in testing terms, this can be quite difficult to do, but in this section, we will have a look at creating a test that can move between windows.

Working with multiple windows

Working with multiple browser windows can be one of the most difficult things to do within a Selenium test. This is down to the fact that the browser needs to allow Selenium to programmatically know how many child browser processes have been spawned.

In the following examples, we will see the tests click on an element on the page that will cause a new window to appear. If you have a pop-up blocker running, it's a good idea to disable it for this site while you work through these examples. Open up Selenium IDE and go to the **Chapter 1** page on the site and refer to the following steps:

1. Click on one of the elements on the page that has the text **Click this link to launch another window**. This will cause a small window to appear.
2. Verify the text in the popup by right-clicking and selecting **VerifyText id=popup text within the popup window**.
3. Once the window has loaded, click on the **Close the Window** text inside it.
4. Add a verify command for an element on the page. Your test should now look like the following screenshot:


comment.html - Selenium IDE 1.5.0
File Edit Actions Options Help
Base URL http://book.theautomatedtester.co.uk/
Fast Slow
Table Source
Find
Log Reference UI-Element Rollup Info Clear

Command	Target	Value
click	multiplewindow	
waitForPopUp	popupwindow	30000
selectWindow	name=popupwind...	
verifyText	popuptext	Text within the ...
click	closepopup	
selectWindow	null	
verifyElementPresent	verifybutton	

```
the pop up window |
[info] Executing: |click | closepopup | |
[info] Executing: |selectWindow | null | |
[info] Executing: |verifyElementPresent | verifybutton | |
```

Sometimes, Selenium IDE will add a `clickAndWait` command instead of a `click` command. This is because it notices that the page has to unload. If this happens, just change the `clickAndWait`command to a click so that it does not cause a timeout in the test.

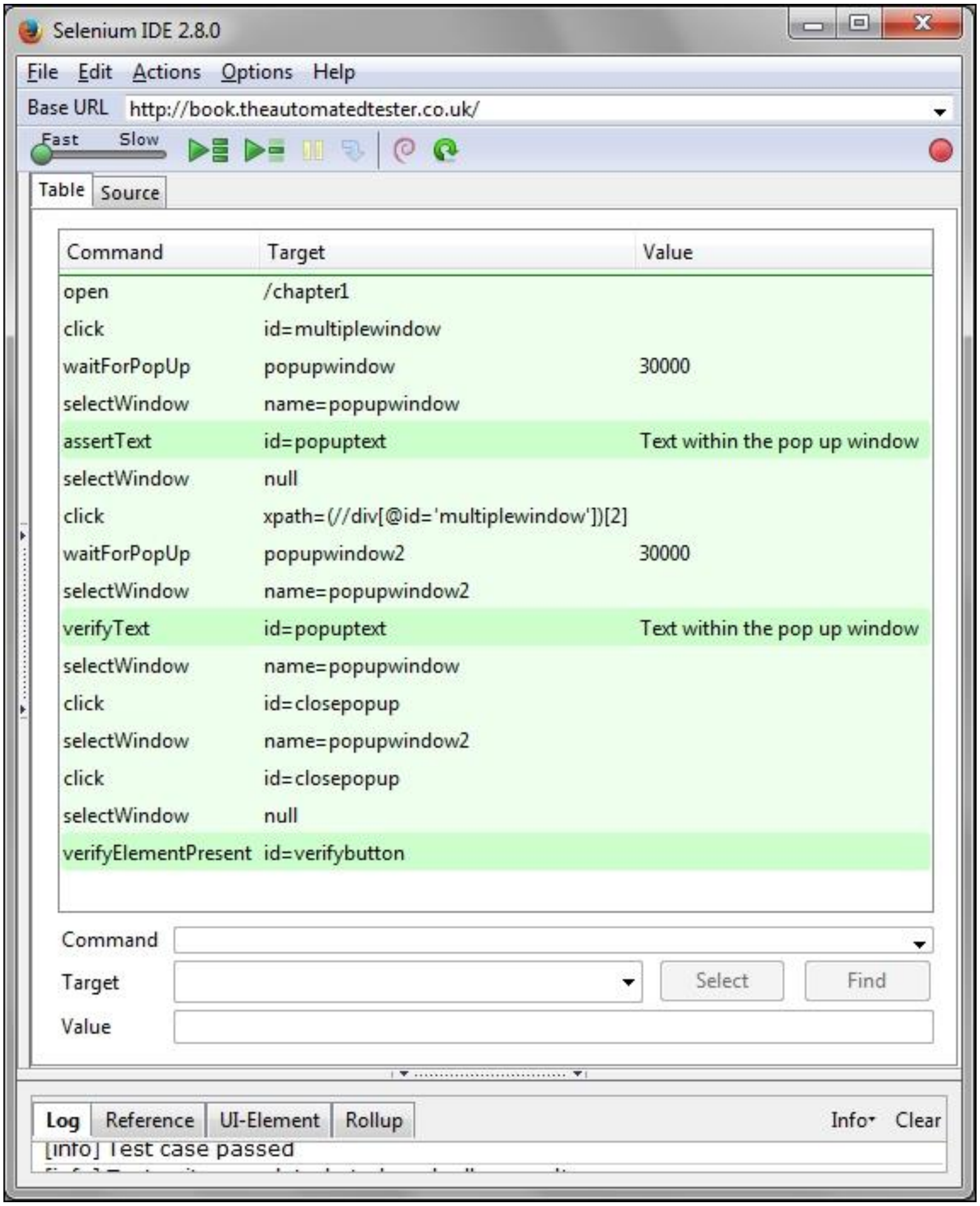
In the test script, we can see that it has clicked on the item to load the new window and then has inserted a `waitForPopUp` command. This is so that your test knows that it has to wait for a web server to handle the request and the browser to render the page. Any commands that require a page to load from a web server will have a `waitFor` command. The next command is the `selectWindow`command. This command tells Selenium IDE that it will need to switch context to the window, called `popupwindow`, and will execute all the commands that follow in that window unless told otherwise by a later command.

Once the test has finished with the pop-up window, it will need to return to the parent window from where it started. To do this, we need to specify `null` as the window. This will force the `selectWindow` command to move the context of the test back to its parent window.

Complex working with multiple windows

In this example, we will open two pop-up windows and move between them and the parent window as it completes its steps:

1. Start Selenium IDE and go to **Chapter 1** on the website.
2. Click on the **Click this link to launch another window** link. This will launch a pop-up window.
3. Assert the text on the page. We do this by right-clicking and selecting **assertText**.
4. Go back to the parent window and click on the link to launch the second pop-up window.
5. Verify the text on the page.
6. Move to the first pop-up window and close it using the close link. As before, be aware of `clickAndWait` instead of click.
7. Move to the second pop-up window and close it using the close link.
8. Move back to the parent window and verify an element on that page.
9. Run your test and watch how it moves between the windows. When complete, it should look like the following screenshot:


Selenium IDE 2.8.0
File Edit Actions Options Help
Fast Slow
Table Source
Select Find
Log Reference UI-Element Rollup Info Clear

Base URL: http://book.theautomatedtester.co.uk/

Command	Target	Value
open	/chapter1	
click	id=multiplewindow	
waitForPopUp	popupwindow	30000
selectWindow	name=popupwindow	
assertText	id=popuptext	Text within the pop up window
selectWindow	null	
click	xpath=(//div[@id='multiplewindow'])[2]	
waitForPopUp	popupwindow2	30000
selectWindow	name=popupwindow2	
verifyText	id=popuptext	Text within the pop up window
selectWindow	name=popupwindow	
click	id=closepopup	
selectWindow	name=popupwindow2	
click	id=closepopup	
selectWindow	null	
verifyElementPresent	id=verifybutton	

```
[info] Test case passed
```

We just had a look at creating a test that can move between multiple windows. We saw how we can move between the child windows and its parent window as though we were a user.

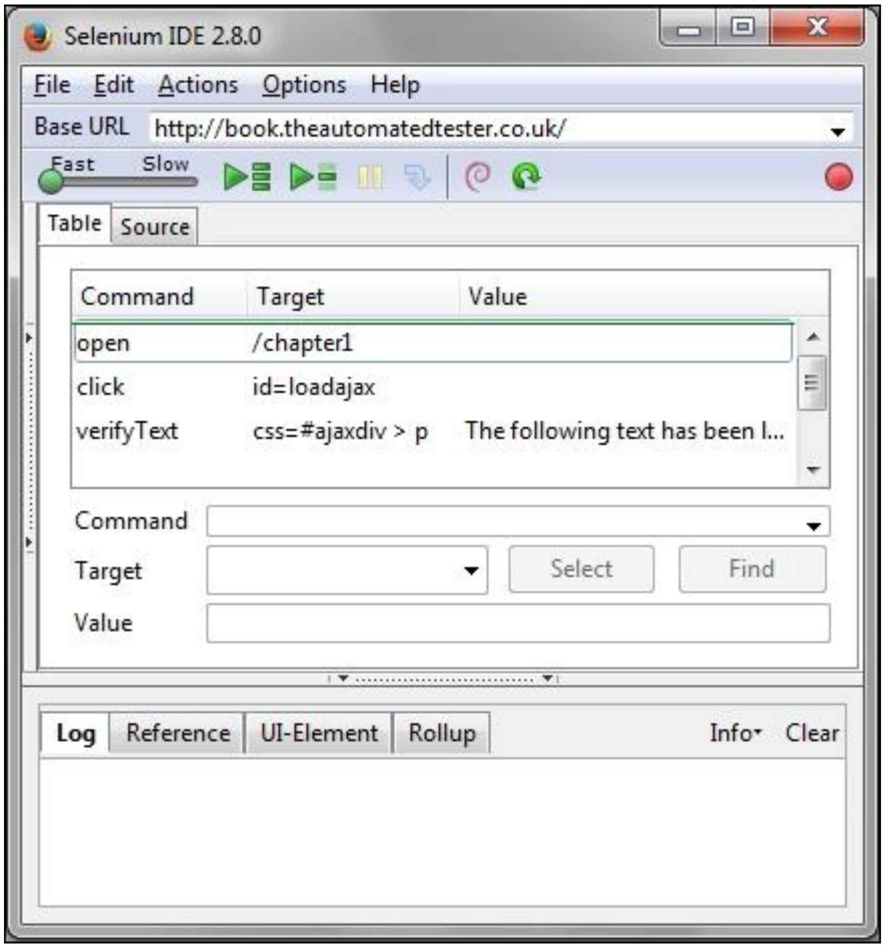
Selenium tests against AJAX applications

Web applications today are being designed in such a way that they appear the same as desktop applications. Web developers are accomplishing this by using AJAX within their web applications. **AJAX** (**Asynchronous JavaScript And XML**) relies on JavaScript creating asynchronous calls and then returning XML with the data that the user or application requires to carry on. AJAX does not rely on XML anymore, as more and more people move over to **JSON** (**JavaScript Object Notation**), which is more lightweight in the way it transfers the data. It does not rely on the extra overhead of opening and closing tags that is needed to create valid XML.

Working on pages with AJAX

In our example, we will click on a link and then assert that some text is visible on the screen:

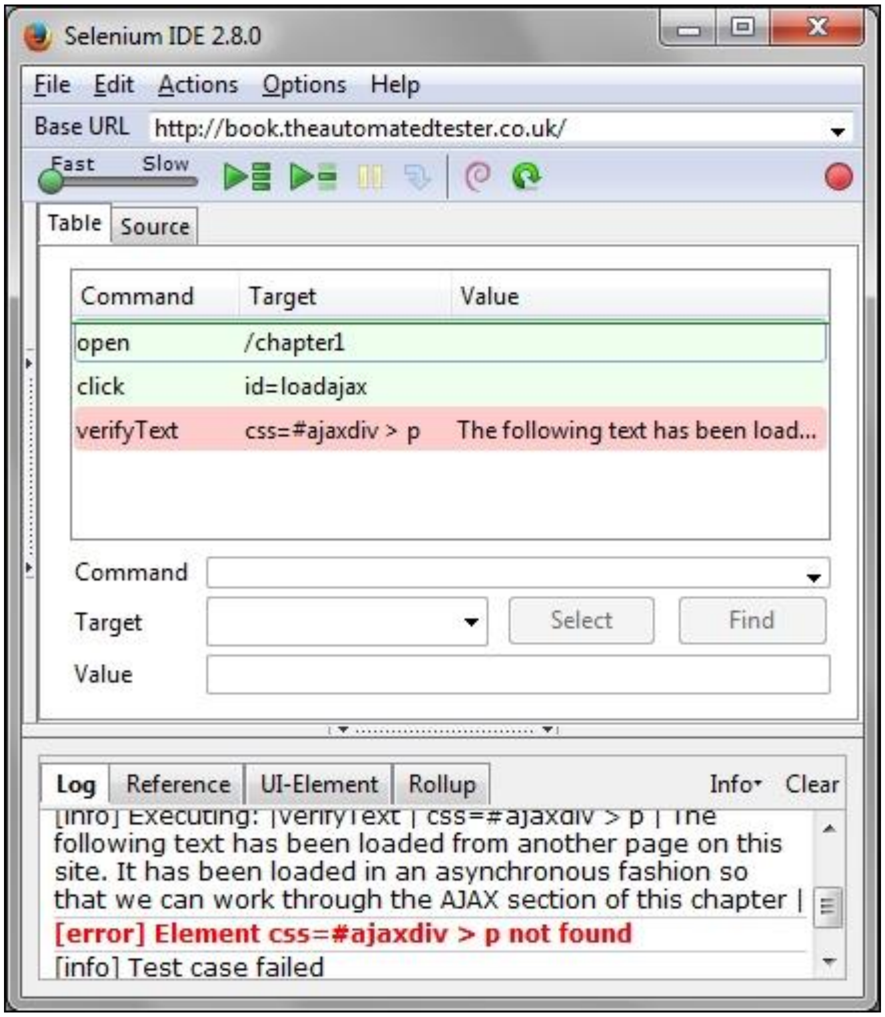
1. Start up Selenium IDE and make sure that the record button is pressed.
2. Navigate to http://book.theautomatedtester.co.uk/chapter1.
3. Click on the text that says **Click this link to load a page with AJAX**.
4. Verify the text that appears on your screen. Your test should look like the following screenshot:


Selenium IDE 2.8.0
File Edit Actions Options Help
Fast Slow
Table Source
Select Find
Log Reference UI-Element Rollup Info Clear

Base URL http://book.theautomatedtester.co.uk/

Command	Target	Value
open	/chapter1	
click	id=loadajax	
verifyText	css=#ajaxdiv > p	The following text has been l...

5. Run the test that you have created. When it has finished running, it should look like the following screenshot:


Selenium IDE 2.8.0
File Edit Actions Options Help
Base URL http://book.theautomatedtester.co.uk/
Fast Slow
Table Source
Command
Target
Value
Select
Find
Log Reference UI-Element Rollup Info Clear

Command	Target	Value
open	/chapter1	
click	id=loadajax	
verifyText	css=#ajaxdiv > p	The following text has been load...

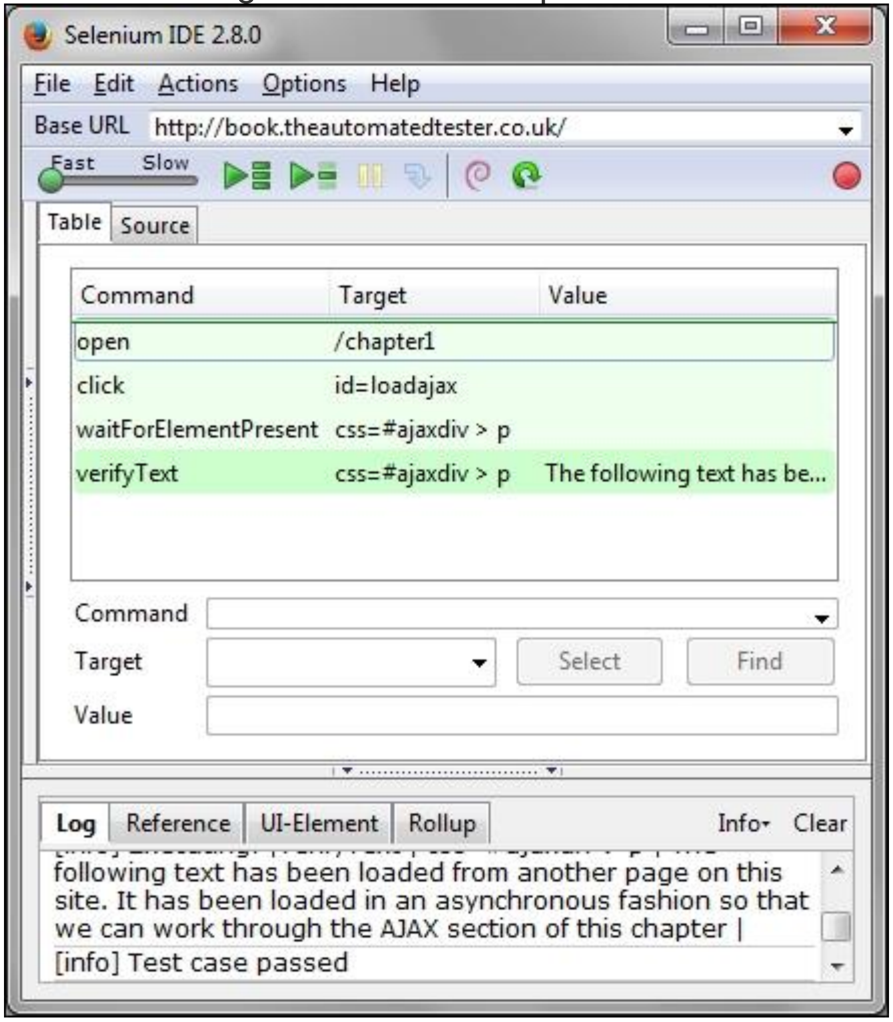
```
[info] Executing: |verifyText | css=#ajaxdiv > p | The following text has been loaded from another page on this site. It has been loaded in an asynchronous fashion so that we can work through the AJAX section of this chapter |
[error] Element css=#ajaxdiv > p not found
[info] Test case failed
```

Have a look at the page that you are working against. Can you see the text that the test is expecting? You should see it, so why has this test failed? The test has failed because when the test reached that point, the element containing the text was not loaded into the DOM. This is because it was being requested and rendered from the web server into the browser.

To remedy this issue, we will need to add a new command to our test so that our tests pass in the future:

1. Right-click on the step that failed so that the Selenium IDE context menu appears.
2. Click on **Insert New Command**.
3. In the **Command** select box, type `waitForElementPresent` or select it from the drop-down menu.
4. In the **Target** box, add the target that is used in the `verifyText` command.

5. Run the test again and it should pass this time:


Selenium IDE 2.8.0
File Edit Actions Options Help
Base URL http://book.theautomatedtester.co.uk/
Fast Slow
Table Source
Select Find
Log Reference UI-Element Rollup Info Clear

Command	Target	Value
open	/chapter1	
click	id=loadajax	
waitForElementPresent	css=#ajaxdiv > p	
verifyText	css=#ajaxdiv > p	The following text has be...

```
following text has been loaded from another page on this
site. It has been loaded in an asynchronous fashion so that
we can work through the AJAX section of this chapter |
[info] Test case passed
```

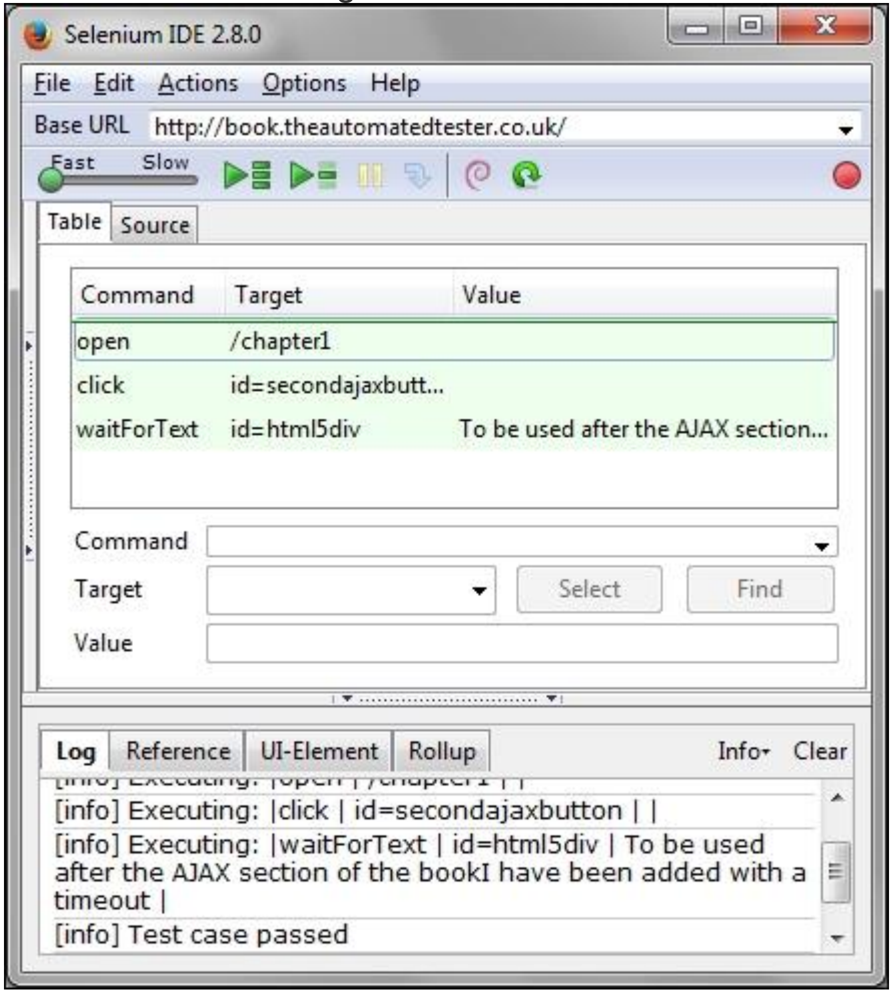
Selenium does not implicitly wait for the item that it needs to interact with, so it is seen as good practice to wait for the item you need to work with and then interact with it. The `waitFor` commands will timeout after 30 seconds by default, but if you need it to wait longer, you can specify the tests by using the `setTimeout` command. This will set the timeout value that the tests will use in future commands.

If need be, you can change the default wait if you go to **Options | Options** and then on the **General**tab, and under **Default timeout value of recorded command in milliseconds** (*30s = 30,000ms*) change it to what you want. Remember, *1,000 milliseconds = 1 second*.

Working with AJAX applications

As more and more applications try to act like desktop applications, we need to be able to handle synchronization steps between our test and our application. In this section, we will see how to handle AJAX and what to synchronize:

1. Navigate to http://book.theautomatedtester.co.uk/chapter1.
2. Click on the **load text to the page** button.
3. Wait for the text **I have been added with a timeout**. Your test will look like the following screenshot:


Selenium IDE 2.8.0
File Edit Actions Options Help
Base URL http://book.theautomatedtester.co.uk/
Fast Slow
Table Source
Command
Target
Value
Select
Find
Log Reference UI-Element Rollup Info Clear

Command	Target	Value
open	/chapter1	
click	id=secondajaxbutt...	
waitForText	id=html5div	To be used after the AJAX section...

```
[info] Executing: |click | id=secondajaxbutton | |
[info] Executing: |waitForText | id=html5div | To be used
after the AJAX section of the bookI have been added with a
timeout |
[info] Test case passed
```

In the previous examples, we waited for an element to appear on the page; there are a number of different commands that we can use to wait. Also, remember that we can take advantage of waiting for something not to be on the page, for example, `waitForElementNotPresent`. This can be just as effective as waiting for it to be there.

The following commands make up the `waitFor` set of commands, but this is not an exhaustive list:

Selenium Command	**Command Description**
`waitForAlertNotPresent`	This waits for an alert to disappear from the page.
`waitForAlertPresent`	This waits for an alert to appear on the page.
`waitForElementPresent`	This waits for an expected element to appear on the page.
`waitForElementNotPresent`	This waits for an expected element to disappear from the page.
`waitForTextPresent`	This waits for expected text and its corresponding HTML tags to appear on the page.
`waitForTextNotPresent`	This waits for expected text and its corresponding HTML tags to disappear from the page.
`waitForPageToLoad`	This waits for all elements to appear on the expected page.
`waitForFrameToLoad`	This waits for an expected frame and its corresponding HTML tags to appear on the page.

A number of these commands are run implicitly when other commands are being run. An example of this is the `clickAndWait` command. This will fire off a `click` command and then fire off a `waitForPageToLoad` command. Another example is the `open` command, which only completes when the page has fully loaded.

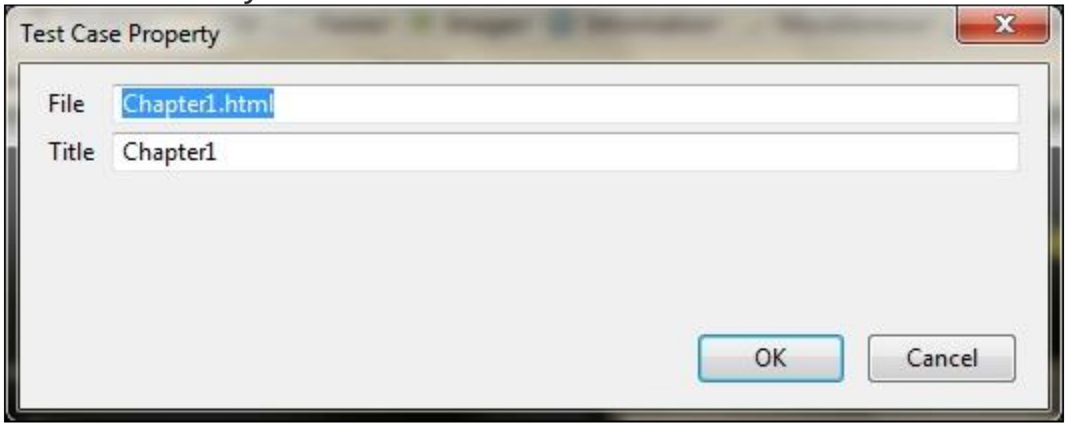
If you are feeling confident, then it's a good time to try different `waitFor` command techniques.

Storing information from the page in the test

Sometimes, there is a need to store elements that are on the page to be used later in a test. It could be that your test needs to pick a date that is on the page and use it later so that you do not need to hardcode values into your test.

Once the element has been stored, you will be able to use it again by requesting it from a JavaScript dictionary that Selenium keeps track of. To use the variable, it will take one of the following two formats: it can look like `${variableName}` or `storedVars['variableName']`. I prefer the `storedVars` format as it follows the same format as within Selenium internals. To see how this works, let's work through the following example:

1. Open up Selenium IDE and switch off the record button.
2. Navigate to http://book.theautomatedtester.co.uk/chapter1.
3. Right-click on the text **Assert that this text is on the page** and go to the `storeText` command in the context menu and click on it.
4. A dialog will appear as shown in the following screenshot. Enter the name of a variable that you want to use. I have used `textOnThePage` as the name of my variable.

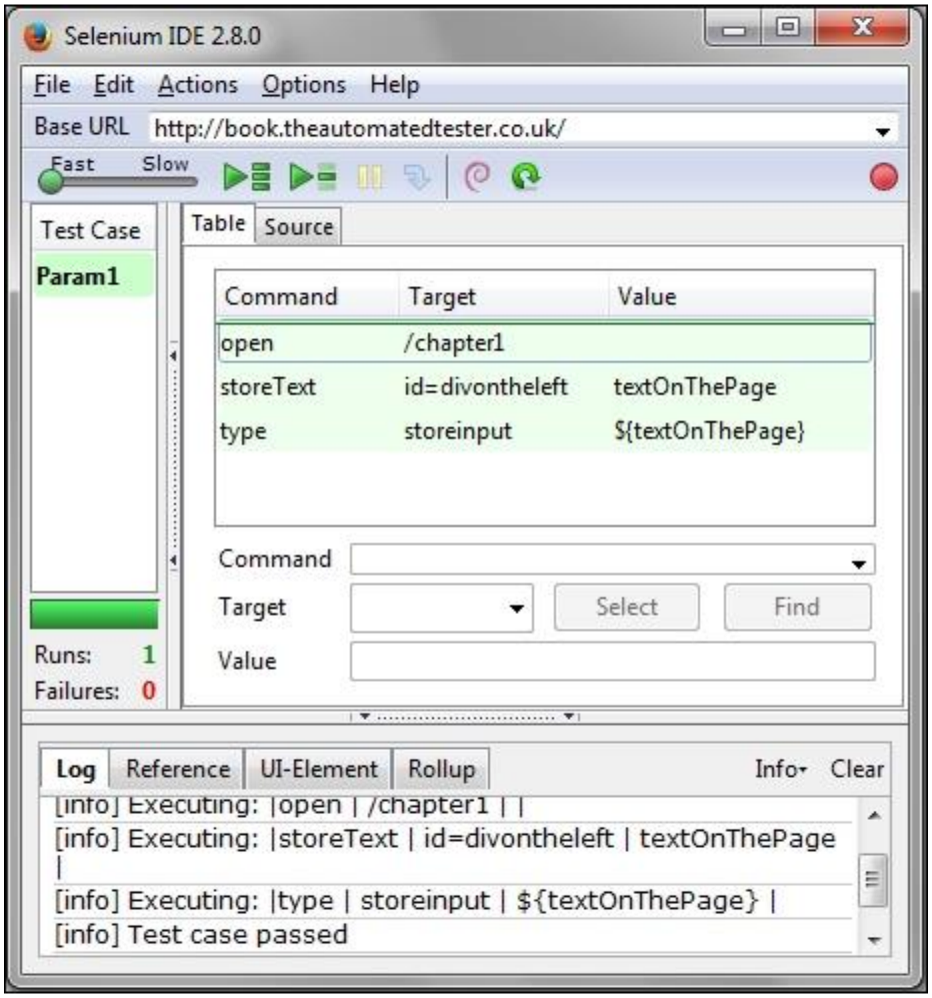


Test Case Property

File: Chapter1.html

Title: Chapter1

5. Click on the row below the `storeText` command in Selenium IDE.
6. Type `type` into the **Command** textbox.
7. Type `storeinput` into the **Target** box.
8. Type `${textOnThePage}` into the **Value** box.
9. Run the test. It should look like the following screenshot:



Command	Target	Value
open	/chapter1	
storeText	id=divontheleft	textOnThePage
type	storeinput	${textOnThePage}

```
[info] Executing: |open | /chapter1 | |
[info] Executing: |storeText | id=divontheleft | textOnThePage
|
[info] Executing: |type | storeinput | ${textOnThePage} |
[info] Test case passed
```

Once your test has completed running, you will see that it has placed **Assert that this text is on the page** into the textbox.

Debugging tests

We have successfully created a number of tests and have seen how we can work against AJAX applications, but unfortunately, creating tests that run perfectly the first time can be difficult. Sometimes, as a test automator, you will need to debug your tests to see what is wrong.

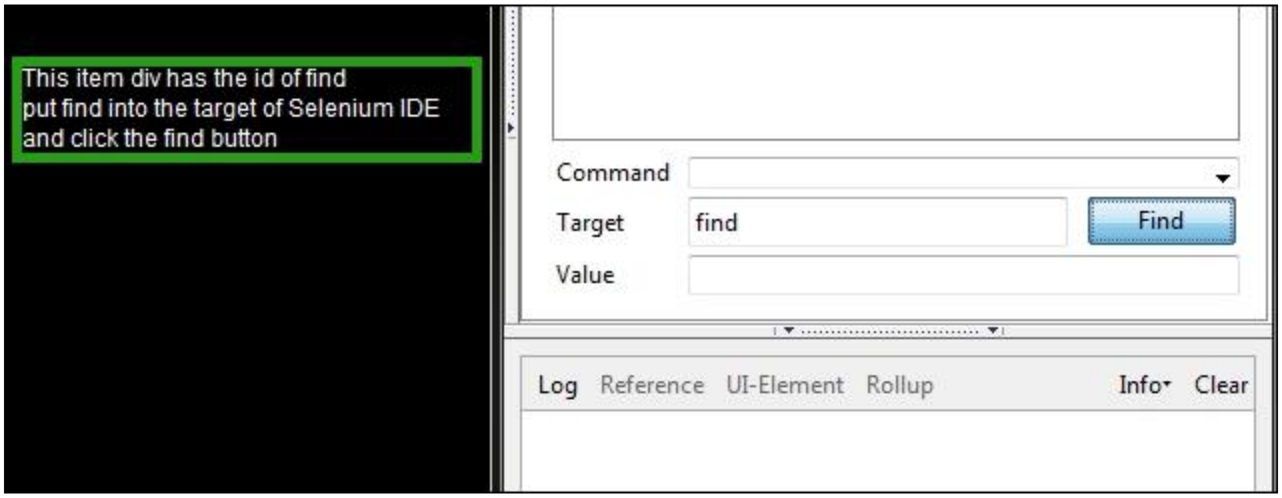
To work through this part of the chapter, you will need to have a test open in Selenium IDE.

These two steps are quite useful when your tests are not running and you want to execute a specific command. They are:

1. Highlight a command.
2. Press the *X* key. This will make the command execute in Selenium IDE.

When a test is running, you can press the pause button to pause the test after the step that is currently being run. Once the test has been paused, the step button is no longer disabled and you can press it to step through the test as if you were stepping through an application.

If you are having issues with elements on the page, you can type in their location and then click on the **Find** button. This will surround the element that you are looking for with a green border that flashes for a few seconds. It should look like the following screenshot:



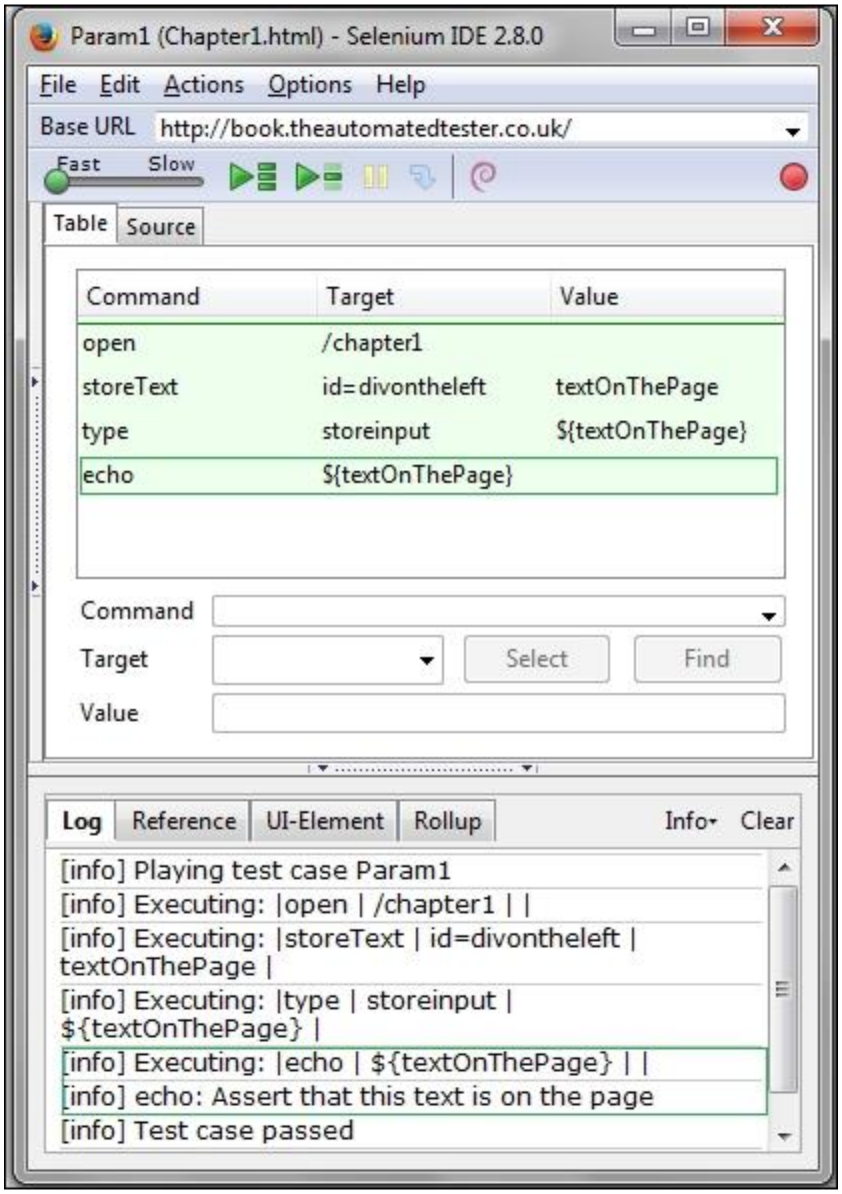
This item div has the id of find
put find into the target of Selenium IDE
and click the find button

Command

Target: find

Value

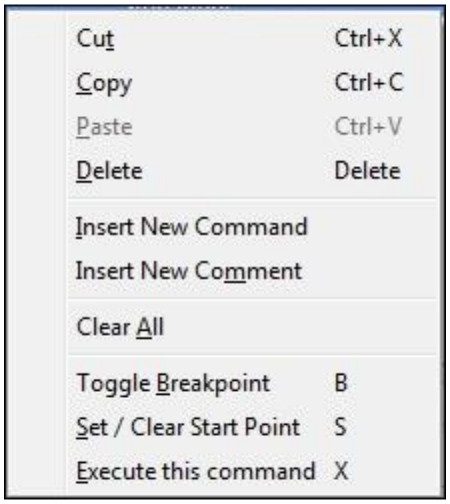
The echo command is also a good way to write something from your test to the log. This is equivalent to the `Console.log` code in JavaScript, for example, `echo | ${textOnThePage}`, as shown in the following screenshot:


Param1 (Chapter1.html) - Selenium IDE 2.8.0
File Edit Actions Options Help
Base URL http://book.theautomatedtester.co.uk/
Fast Slow
Table Source
Command
Target
Value
Select
Find
Log Reference UI-Element Rollup Info Clear

Command	Target	Value
open	/chapter1	
storeText	id=divontheleft	textOnThePage
type	storeinput	${textOnThePage}
echo	${textOnThePage}	

```
[info] Playing test case Param1
[info] Executing: |open | /chapter1 | |
[info] Executing: |storeText | id=divontheleft |
textOnThePage |
[info] Executing: |type | storeinput |
${textOnThePage} |
[info] Executing: |echo | ${textOnThePage} | |
[info] echo: Assert that this text is on the page
[info] Test case passed
```

Also, remember that if you are trying to debug a test script that you have created with Selenium IDE, you can set breakpoints in your test. You simply right-click on the line and select breakpoint from the list. It will be similar to the following screenshot:

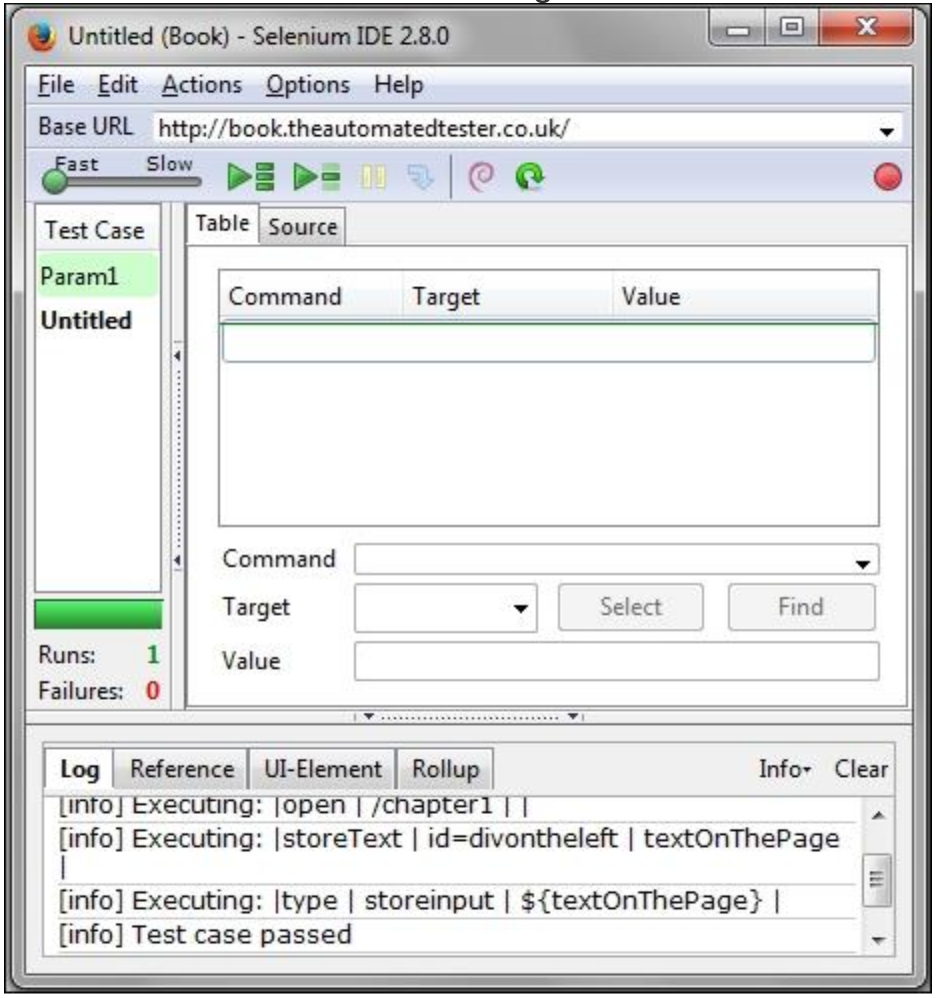


You can also use the keyboard shortcut of **B** to allow you to do it quicker.

Creating test suites

We managed to create a number of tests using Selenium IDE and have managed to run them successfully. The next thing to have a look at is how to create a test suite, so that we can open the test suite and then have it run a number of tests that we have created. If you have Selenium IDE open from the last steps, click on the **File** menu:

1. Click **New Test Case**.
2. You will see that Selenium IDE has opened a new area on the left of the IDE as shown in the following screenshot:


Untitled (Book) - Selenium IDE 2.8.0
File Edit Actions Options Help
Base URL
Fast Slow
Test Case
Table Source
Command Target Value
Select Find
Log Reference UI-Element Rollup Info Clear

Base URL: http://book.theautomatedtester.co.uk/

Test Case: Param1, Untitled

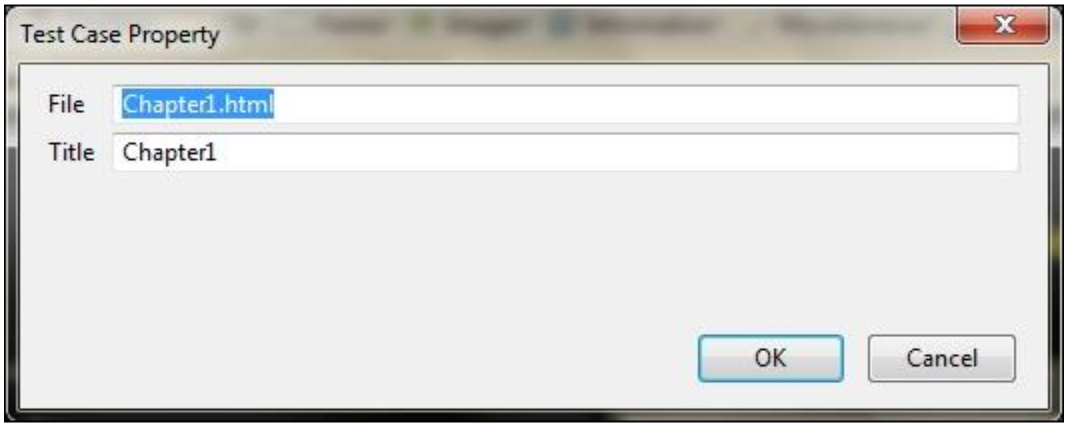
Runs: 1
Failures: 0

```
[info] Executing: |open | /chapter1 | |
[info] Executing: |storeText | id=divontheleft | textOnThePage
|
[info] Executing: |type | storeinput | ${textOnThePage} |
[info] Test case passed
```

You can do this as many times as you want, and when the **Play entire test suite** button is clicked, it will run all the tests in the test suite. It will log all the passes and failures at the bottom of the **Test Case**box.

To save this, click on the **File** menu and then click **Save Test Suite** and save the test suite file to a place where you can get to it again. One thing to note is that saving a test suite does not save the test case. Make sure that you save the test case every time you make a change and not just the test suite.

To change the name of the test case to something a lot more meaningful, you can do this by right-clicking on the test and clicking on the **Properties** item in the context menu:



File: Chapter1.html
Title: Chapter1

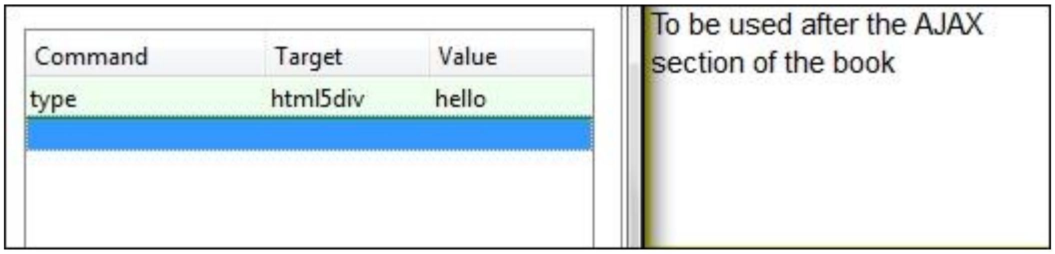
You can now add meaningful names to your tests and they will appear in Selenium IDE instead of falling back to their filenames.

We have managed to create our first test suite. This can be used to group tests together to be used later. If your tests have been saved, you can update the test suite properties to give the tests a name that is easier to read.

What you cannot record

We have seen our tests work really well by recording them and then playing them back. Unfortunately, there are a number of things that Selenium cannot do. Since Selenium was developed in JavaScript, it tries to synthesize what the user does with JavaScript events. This means that it is bound by the same rules that JavaScript has in any browsers by operating within the sandbox.

- Silverlight and Flex/Flash applications, at the time of writing, cannot be recorded with Selenium IDE. Both these technologies operate in their own sandbox and do not operate with the DOM to do their work. HTML5, at the time of writing, is not fully supported with Selenium IDE. A good example of this is elements that have the `contentEditable=true` attribute. If you want to see this, you can use the `type` command to type something into the `html5div` element. The test will tell you that it has completed the command, but the UI will not have changed, as shown in the following screenshot:



Command	Target	Value
type	html5div	hello

To be used after the AJAX section of the book

- Selenium IDE does not work with Canvas elements on the page either, so you will not be able to make your tests move items around on a page.
- Selenium cannot do file uploads. This is due to the JavaScript sandbox not allowing JavaScript to interact with `<input type=file>` elements on a page. While you might be able to send the text to the box, it will not always do what you expect, so I would recommend not doing it.

We will be able to automate a number of these elements with **Selenium WebDriver** in later chapters of this book.

Summary

We learned a lot in this section about Selenium IDE, learning how to create your first test using the record and replay functions and we now understand some of the basic concepts such as moving between multiple windows that can appear in a test, and saving our tests for future use.

Specifically, we covered the following topics:

- **How to install Selenium IDE**: We started by downloading Selenium IDE from http://seleniumhq.org.
- **What Selenium IDE is made up of**: The breakup of Selenium IDE allowed us to see what makes up Selenium IDE. It allowed us to understand the different parts that make up a command that will be executed in a test as well as its basic format. We had a look at how to load Selenium IDE and how to get started with recording tests. We saw that a Selenium IDE command is made up of three sections: the command, the target, and the value that might be used.
- **Recording and replaying tests**: We used Selenium IDE to record a workflow that a user will need in their tests. We also had a look at verifying and asserting that elements are on the page and that the text we are expecting is also on the page.
- **How to add comments to tests**: In this section of the chapter, we saw how to add comments to the tests so that they are more maintainable.
- **Working with multiple windows**: In this section, we saw how applications today can have pop-up windows that tests need to be able to move between.
- **Working with AJAX applications**: AJAX applications do not have the items needed for the tests when the tests get to commands. To get around this, we had a look at adding `waitFor` commands to the tests. This is due to the fact that Selenium does not implicitly wait for elements to appear on the page.
- **Storing information in variables**: There is always something that is on the page that needs to be used later, but unfortunately, you will not know what the value is before the test runs. This section showed us how we can record items into a variable and use it later in a test. This can be something that has happened on a page and needs to be checked that it is still there on later pages.

- **Debugging tests**: Creating tests does not always go according to the plan, so in this section, we saw some of the different ways to debug your tests.
- **Parameterization**: Parameterizing the data in Selenium IDE.
- **Saving test suites**: Finally, we saw how we can save tests for future use and how we can save them into different groups by saving them into test suites.

We also discussed what cannot be tested using Selenium IDE. We saw that Silverlight and Flex/Flash applications cannot be tested, and that when working with a number of HTML5 elements, the tests say that they have completed the tasks even though the UI has not changed. In later chapters, we will discuss different mechanisms that we can use within our tests that might be useful against HTML5 elements on the page.

Now that we've learnt about Selenium IDE, we're ready to look at all the different techniques we can use to find elements on the page.