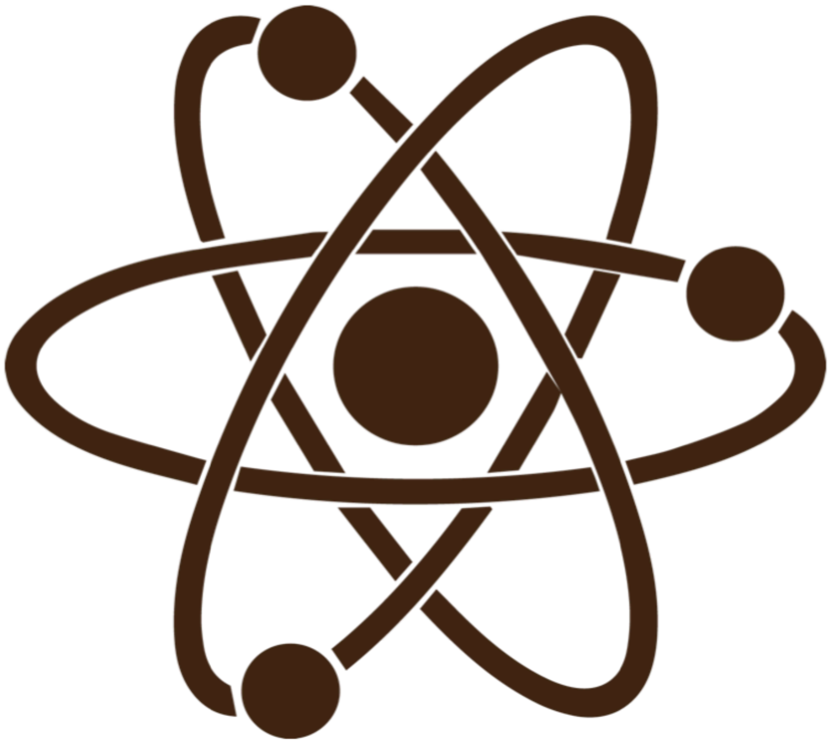
Elemental Selenium



JavaScript Tips

By Dave Haeffner

Version 4.0.0

Preface

This book is a compendium of tips from my weekly Selenium tip newsletter (Elemental Selenium). It's aim is to give you a glimpse into the things you'll see in the wild, and a reference for how to deal with them.

The tips differ from The Selenium Guidebook in that they do not build upon previous examples. Instead, they serve as standalone works that can be consumed individually. So feel free to read them in order, or jump around.

Enjoy!

Acknowledgements

When I started my weekly tip newsletter for Selenium it was just in Ruby. I eventually branched out to Java, and later on to C# and Python. I saved JavaScript for last, and unfortunately, never got around to making tips available for the language -- until now.

Somewhere along the way I open-sourced my tip code examples and started receiving contributions from the community.

For JavaScript, Manoj Kumar stepped up and submitted the first examples using ES6 and `async` / `await`. It was a welcome contribution that served as a helpful nudge to get me started on this path. Thank you!

It's been a long time coming for me to offer these tips in JavaScript, and I'm excited to finally do so.

Table of Contents

Chapter 1
How To Upload a File

The Problem

Uploading a file is a common piece of functionality found on the web. But when trying to automate it you get prompted with a with a dialog box that is just out of reach for Selenium.

In these cases people often look to a third-party tool to manipulate this window (e.g., AutoIt). While this can help solve your short-term need, it sets you up for failure later by chaining you to a specific platform (e.g., AutoIt only works on Windows), effectively limiting your ability to test this functionality on different browser & operating system combinations.

A Solution

A work-around for this problem is to side-step the system dialog box entirely. We can do this by using Selenium to insert the full path of the file we want to upload (as text) into the form and then submit the form.

Let's step through an example.

An Example

NOTE: We are using a file upload example found on the-internet.

First let's pull in our requisite libraries for asertions, constructing a path to a file, and driving the browser with Selenium.

```
// filename: test/upload.spec.js
const assert = require("assert");
const { Builder, By } = require("selenium-webdriver");
const path = require("path");
// ...
```

Now to create a new test class and add in test setup and teardown for it.

```
// filename: test/upload.spec.js
describe("Upload Test", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("firefox").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

After declaring the class we create two methods. The first method, `beforeEach`, will execute before each test in this class. In it we are launching a new instance of Firefox with Selenium and storing it in a class variable that we'll use throughout this class. After our test executes the second method, `afterEach`, will execute. This calls `driver.quit()` which ends the session by closing the browser instance and the destroying the Selenium object in-memory.

Now to wire up our test.

```
// filename: test/upload.spec.js
// ...
  it("upload a file", async function() {
    let filename = "some-file.txt";
    let filePath = path.join(process.cwd(), filename);

    await driver.get("http://the-internet.herokuapp.com/upload");
    await driver.findElement(By.id("file-upload")).sendKeys(filePath);
    await driver.findElement(By.id("file-submit")).click();

    let text = await driver.findElement(By.id("uploaded-files")).getText();
    assert.equal(text, filename);
  });
});
```

After declaring the test method (e.g., `it("upload a file", async function() {`) we specify the name of file we'd like to upload. It's a text file called `some-file.txt` in the current working directory. We determine the path to this file and store it in a variable called `filePath`.

Next we visit the page with the upload form, input the string value of `filePath`, and click the submit button on the form. After the file is uploaded to the page it will display the filename it just processed. We use this text on the page to perform our assertion (making sure the uploaded file is what we expect).

Expected Behavior

When we save this file and run it (e.g., `mocha` from the command-line) this is what will happen:

- Open the browser
- Visit the upload form page
- Inject the file path into the form and submit it
- Page displays the uploaded filename
- Grab the text from the page and assert it's what we expect
- Close the browser

Outro

This approach will work across all browsers. If you want to use it with a remote instance (e.g., on Selenium Grid or Sauce Labs) then you'll want to have a look at `file_detector`.

Happy Testing!

Chapter 2
How To Download a File

The Problem

Just like with uploading files we hit the same issue with downloading them. A dialog box just out of Selenium's reach.

A Solution

With some additional configuration when setting up Selenium we can easily side-step the dialog box. This is done by instructing the browser to download files to a specific location without triggering the dialog box.

After the file is downloaded we can perform some simple checks to make sure the file is what we expect.

Let's dig in with an example.

An Example

Let's start by pulling in our requisite libraries.

```
// filename: test/download.spec.js
const assert = require("assert");
const { Builder, By, Key } = require("selenium-webdriver");
const firefox = require("selenium-webdriver/firefox");
const fs = require("fs");
const path = require("path");
// ...
```

Now to create a setup method for our test.

```
// filename: test/download.spec.js
// ...
describe("File download", function() {
  let driver;
  const tmpDir = path.join(__dirname, "tmp");

  beforeEach(async function() {
    if (!fs.existsSync(tmpDir)) fs.mkdirSync(tmpDir);
    let options = new firefox.Options()
      .setPreference("browser.download.dir", tmpDir)
      .setPreference("browser.download.folderList", 2)
      .setPreference(
        "browser.helperApps.neverAsk.saveToDisk",
        "images/jpeg, application/pdf, application/octet-stream"
      )
      .setPreference("pdfjs.disabled", true);

    driver = await new Builder()
      .forBrowser("firefox")
      .setFirefoxOptions(options)
      .build();
  });
// ...
```

After declaring our test suite, we declare two variables. One for the Selenium instance (e.g., `driver`) and the other for the temporary directory where we'll want to automatically download files to (e.g., `tmpDir`). In it we're storing the absolute path to the current working directory, plus the name `tmp`.

In the setup method (e.g., `beforeEach`) we create this directory if it's not already there, and then create a new browser options object (for Firefox in this case), specifying the necessary configuration parameters to make it automatically download the file where we want (e.g., in the newly created temp directory).

Here's a breakdown of each of the browser preferences being set:

- `browser.download.dir` accepts a string. This is how we set the custom download path. It needs to be an absolute path.
- `browser.download.folderList` takes a number. It tells Firefox which download directory to use. `2` tells it to use a custom download path, wheras `1` would use the browser's default path, and `0` would place them on the Desktop.
- `browser.helperApps.neverAsk.saveToDisk` tells Firefox when not to prompt for a file download. It accepts a string of the file's MIME type. If you want to specify more than one, you do it with a comma-separated string (which we've done).
- `pdfjs.disabled` is for when downloading PDFs. This overrides the sensible default in Firefox

that previews PDFs in the browser. It accepts a boolean.

We then hand the options object on to Selenium as part of the incantation to create a new browser instance.

Now let's take care of our test's teardown.

```
// filename: test/download.spec.js
// ...
  function cleanupTmpDir() {
    if (fs.existsSync(tmpDir)) {
      const files = fs.readdirSync(tmpDir).map(file => path.join(tmpDir, file));
      files.forEach(file => fs.unlinkSync(file));
      fs.rmdirSync(tmpDir);
    }
  }

  afterEach(async function() {
    await driver.quit();
    cleanupTmpDir();
  });
// ...
```

In the teardown (e.g., `beforeEach`) we close the browser instance and then clean up the temp directory by deleting its contents, and then the directory itself.

Now to wire up our test.

```
// filename: test/download.spec.js
// ...
  it("should automatically download to local disk", async function() {
    await driver.get("http://the-internet.herokuapp.com/download");
    await driver.findElement(By.css(".example a")).click();
    const files = fs.readdirSync(tmpDir).map(file => path.join(tmpDir, file));
    assert(files.length);
    assert(fs.statSync(files[0]).size);
  });
});
```

After visiting the page we find the first download link and click it. The click triggers an automatic download to the temp directory. After the file downloads, we perform some rudimentary checks to make sure the temp directory contains files and the first file in the directory is not empty.

Expected Behavior

When we save this file and run it (e.g., `mocha` from the command-line) this is what will happen:

- Create a uniquely named temp directory in the present working directory
- Open the browser
- Visit the page
- Find and click the first download link on the page
- Automatically download the file to the temp directory without prompting
- Check that the temp directory is not empty
- Check that the downloaded file is not empty
- Close the browser
- Delete the temp directory

Outro

A similar approach can be applied to some other browsers with varying configurations.

Happy Testing!

Chapter 3
How To Download a File Without a Browser

The Problem

In a previous chapter we stepped through how to download files with Selenium by configuring the browser to download them locally and verifying their file size when done.

While this works it requires a custom configuration that is inconsistent from browser to browser.

A Solution

Ultimately we shouldn't care if a file was downloaded or not. Instead, we should care that a file can be downloaded. And we can do that by using an HTTP client alongside Selenium in our test.

With an HTTP library we can perform a header (or `HEAD`) request for the file. Instead of downloading the file we'll receive header information for the file which contains information like the content type and content length (amongst other things). With this information we can easily confirm the file is what we expect without onerous configuration, local disk usage, or lengthy download times (depending on the file size).

Let's dig with an example.

An Example

To start things off let's pull in our requisite libraries and wire up some test setup and teardown methods.

```
// filename: test/download-v2.spec.js
const assert = require("assert");
const { Builder, By } = require("selenium-webdriver");
const http = require("http");
const url = require("url");

describe("File download v2", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("chrome").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Next we'll want to write a helper function to create the options we'll use to perform the HTTP request.

```
// filename: test/download-v2.spec.js
  function getHttpOptions(target) {
    const _url = url.parse(target);
    return {
      method: "HEAD",
      protocol: _url.protocol,
      hostname: _url.hostname,
      path: _url.path
    };
  }
// ...
```

The helper function (e.g., `getHttpOptions`) receives a URL, parses it, and use it to create and return an object with the necessary values.

Now we're ready to wire up our test.

It's just a simple matter of visiting the page with download links, grabbing a URL from one of them, and performing a `HEAD` request with it.

```
// filename: test/download-v2.spec.js
// ...
  it("verify a file download by inspecting its head request", async function() {
    await driver.get("http://the-internet.herokuapp.com/download");
    const download_url = await driver
      .findElement(By.css(".example a"))
      .getAttribute("href");
    const request = http.request(getHttpOptions(download_url), response => {
      assert(response.headers["content-type"] === "application/octet-stream");
      assert(Number(response.headers["content-length"]) > 0);
    });
    request.end();
  });
});
```

Once we receive the response we check its header for the `content-type` and `content-length` to make sure the file is the correct type and not empty.

Expected Behavior

When you save this and run it (e.g., `mocha` from the command-line) here is what will will happen:

- Open the browser
- Load the page
- Grab the URL of the first download link
- Perform a `HEAD` request against it with an HTTP library
- Store the response
- Check the response headers to see that the file type is correct
- Check the response headers to see that the file is not empty

Outro

Compared to the browser specific configuration with Selenium this is hands down a leaner, faster, and more maintainable approach. But unfortunately it only works with files served up from a dedicated URL. So if you're trying to test file downloads that are generated in-memory as part of the browser session (a.k.a. not accessible from a URL) then you'll need to reach for the browser specific Selenium configuration.

Happy Testing!

Chapter 4
How To Take A Screenshot on Failure

The Problem

With browser tests it can often be challenging to track down the issue that caused a failure. By itself a failure message along with a stack trace is hardly enough to go on. Especially when you run the test again and it passes.

A Solution

A simple way to gain insight into your test failures is to capture screenshots at the moment of failure. And it's a quick and easy thing to add to your tests.

Let's dig in with an example.

An Example

Let's start by importing our requisite libraries and wire up some setup and teardown methods.

```
// filename: test/screenshot.spec.js
const assert = require("assert");
const { Builder, By } = require("selenium-webdriver");
const fs = require("fs");
const path = require("path");

describe("Screenshot", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("chrome").build();
  });

  afterEach(async function() {
    if (this.currentTest.state !== "passed") {
      const testName = this.currentTest.fullTitle().replace(/\s/g, "-");
      const fileName = path.join(__dirname, `screenshot_${testName}.jpg`);
      fs.writeFileSync(fileName, await driver.takeScreenshot(), "base64");
    }
    await driver.quit();
  });
// ...
```

In `afterEach` we check to see if the test was unsuccessful (e.g., `this.currenTest.state !== "passed"`). If not, then the test has either failed or errored and we capture a screenshot through the help of Selenium's `.takeScreenshot()` function. To save it to disk, we use `fs` and its `writeFileSync` function. It takes the full path to save to, the payload (e.g., the screenshot), and the encoding (which for the image is `"base64"`).

To make the filename unique we use the test name after cleaning it up (by replacing spaces with `-`). When this command executes it will save an image file (e.g., `screenshot_Screenshot-on-failure.jpg`) to the local system in the current working directory.

Now to wire up a test which will fail.

```
// filename: test/screenshot.spec.js
// ...
  it("on failure", async function() {
    await driver.get("http://the-internet.herokuapp.com");
    assert(true === false);
  });
});
```

Expected Behavior

When we save this file and run it (`mocha` from the command-line) here is what will happen:

- Open the browser
- Load the homepage of the-internet
- Fail on the assertion
- Capture a screenshot in the current working directory
- Close the browser

Outro

Happy Testing!

Chapter 5
How To Use Selenium Grid

The Problem

If you're looking to run your tests on different browser and operating system combinations but you're unable to justify using a third-party solution like Sauce Labs then what do you do?

A Solution

With Selenium Grid you can stand up a simple infrastructure of various browsers on different operating systems to not only distribute test load, but also give you a diversity of browsers to work with.

A brief Selenium Grid primer

Selenium Grid is part of the Selenium project. It lets you distribute test execution across several machines. You can connect to it with Selenium Remote by specifying the browser, browser version, and operating system you want. You specify these values through Selenium Remote's `Capabilities`.

There are two main elements to Selenium Grid -- a hub, and nodes. First you need to stand up a hub. Then you can connect (or "register") nodes to that hub. Nodes are where your tests will run, and the hub is responsible for making sure your tests end up on the right one (e.g., the machine with the operating system and browser you specified in your test).

Let's step through an example.

An Example

Part 1: Grid Setup

Selenium Grid comes built into the Selenium Standalone Server. So to get started we'll need to download the latest version of it from here.

Then we need to start the hub.

```
> java -jar selenium-server-standalone.jar -role hub
19:05:12.718 INFO - Launching Selenium Grid hub
...
```

After that we can register nodes to it.

```
> java -jar selenium-server-standalone.jar -role node -hub
http://localhost:4444/grid/register
19:05:57.880 INFO - Launching a Selenium Grid node
...
```

NOTE: This example only demonstrates a single node on the same machine as the hub. To span nodes across multiple machines you will need to place the standalone server on each machine and launch it with the same registration command (replacing **`http://localhost`** with the location of your hub, and specifying additional parameters as needed).

Now that the grid is running we can view the available browsers by visiting our Grid's console at `http://localhost:4444/grid/console`.

To refine the list of available browsers, we can specify an additional `-browser` parameter when registering the node. For instance, if we wanted to only offer Safari on a node, we could specify it with `-browser browserName=safari`, which would look like this:

```
java -jar selenium-server-standalone.jar -role node -hub
http://localhost:4444/grid/register -browser browserName=safari
```

We could also repeat this parameter again if we wanted to explicitly specify more than one browser.

```
java -jar selenium-server-standalone.jar -role node -hub
http://localhost:4444/grid/register -browser browserName=safari -browser browserName=
chrome -browser browserName=firefox
```

There are numerous parameters that we can use at run time. You can see a full list here.

Part 2: Test Setup

Now let's wire up a simple test script to use our new Grid.

```
// filename: test/grid.spec.js
const assert = require("assert");
const { Builder, By } = require("selenium-webdriver");

describe("Grid", function() {
  let driver;

  beforeEach(async function() {
    const url = "http://localhost:4444/wd/hub";
    const caps = {
      browserName: "chrome"
    };
    driver = await new Builder()
      .usingServer(url)
      .withCapabilities(caps)
      .build();
  });

  afterEach(async function() {
    await driver.quit();
  });

  it("hello world", async function() {
    await driver.get("http://the-internet.herokuapp.com/");
    assert((await driver.getTitle()) === "The Internet");
  });
});
```

Notice in `beforeEach` we're using a URL to connect to the grid (e.g., `usingServer(url)`). And we are telling the grid which browser we want to use with a `caps` object (caps is short for capabilities).

You can see a full list of the available Selenium `Capabilities` options here.

Expected Behavior

When we save this file and run it (e.g., `mocha` from the command-line) here is what will happen:

- test connects to the grid hub
- hub determines which node has the necessary browser/platform combination
- hub opens an instance of the browser on the found node
- test runs on the new browser instance
- test completes and the browser closes on the node

Outro

If you're looking to set up Selenium Grid to work with Internet Explorer or Chrome, be sure to read up on how to set them up since there is additional configuration required for each. And if you run into issues, be sure to check out the browser driver documentation for the browser you're working with:

- ChromeDriver
- EdgeDriver
- geckodriver (Firefox)
- InternetExplorerDriver
- SafariDriver

Also, it's worth noting that while Selenium Grid is a great option for scaling your test infrastructure, it by itself will NOT give you parallelization. That is to say, it can handle as many connections as you throw at it (within reason), but you will still need to find a way to execute your tests in parallel.

Happy Testing!

Chapter 6

How To Test Checkboxes

The Problem

Checkboxes are an often used element in web applications. But how do you work with them in your Selenium tests? Intuitively you may reach for a method that has the word 'checked' in it -- like `.checked` or `.is_checked`. But this doesn't exist in Selenium. So how do you do it?

A Solution

There are two ways to approach this -- by seeing if an element has a `checked` attribute (a.k.a. performing an attribute lookup), or by asking an element if it has been selected.

Let's step through each approach to see their pros and cons.

An Example

For reference, here is the markup from the page we will be testing against (an example from the-internet).

```
<form>
  <input type="checkbox"> unchecked<br>
  <input type="checkbox" checked=""> checked
</form>
```

Let's kick things off by requiring our dependent libraries, declare our test class, and wire up some test setup and teardown methods.

```
// filename: test/checkboxes.spec.js
const assert = require("assert");
const { Builder, By } = require("selenium-webdriver");

describe("Checkboxes", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("chrome").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Before we dig into writing a test to verify the state of the page, let's walk through both checkbox approaches to see what Selenium gives us.

To do that we'll want to grab all of the checkboxes on the page and iterate through them. Once using an attribute lookup and again asking if the element is selected. For each we'll output the return values we get from Selenium.

```
// filename: test/checkboxes.spec.js
// ...
  it("should list values for different approaches", async function() {
    await driver.get("http://the-internet.herokuapp.com/checkboxes");
    const checkboxes = await driver.findElements(
      By.css('input[type="checkbox"]')
    );

    console.log("With .getAttribute('checked')");
    for (let checkbox in checkboxes)
      console.log(await checkboxes[checkbox].getAttribute("checked"));

    console.log("\nWith .is_selected");
    for (let checkbox in checkboxes)
      console.log(await checkboxes[checkbox].isSelected());
  });
});
// ...
```

When we save our file and run it (e.g., `mocha` from the command-line), here is the output we'll see.

```
With .attribute('checked')
null
true

With .is_selected
false
true
```

With the attribute lookup, depending on the state of the checkbox, we receive either a `null` or a `true` boolean value. Whereas with `.isSelected` we get a boolean value either way.

Let's see what these approaches look like when put to use in our test.

```
// filename: test/checkboxes.spec.js
// ...
  it("should list values for different approaches", async function() {
    // ...
    assert(checkboxes[checkboxes.length - 1].getAttribute("checked"));
    assert(checkboxes[checkboxes.length - 1].isSelected());
    assert(checkboxes[0].getAttribute("checked"));
    assert(checkboxes[0].isSelected());
  });
});
// ...
```

With either approach we can simply assert on the return value (even if it's a value of `null` -- since in JavaScript `null` evaluates to `false`) and have things work as expected.

Expected Behavior

When we save and run the file (e.g., `mocha` from the command-line) here is what will happen:

- Open the browser
- Visit the page
- Find all of the checkboxes on the page
- Assert that the last checkbox (the one that is supposed to be checked on initial page load) is checked
- Close the browser

Outro

Attribute lookups are generally meant for pulling information out of the page for review. However in this case they lend themselves to seeing if a checkbox is checked. But there is a method which was built for this use case that is more readable and has a predictable set of return values. So `isSelected` should be the thing you reach for, knowing that an attribute lookup is there as a solid backup if you find you need it.

Happy Testing!

Chapter 7

How To Test For Disabled Elements

The Problem

On occasion you may have the need to check if an element on a page is disabled or enabled. Sounds simple enough, but how do you do it? It's not a well known function of Selenium. So doing a trivial action like this can quickly become a pain.

A Solution

If we look at the API documentation for Selenium's Element class we can see there is an available method called `isEnabled` that can help us accomplish what we want.

Let's take a look at how to use it.

An Example

For this example we will use a dropdown list from the-internet. In this list there a few options to select, one which should be disabled. Let's find this element and assert that it is disabled.

First we'll require our dependent libraries, declare our test class, and wire up some setup and teardown methods for our test.

```
// filename: test/disabledElement.spec.js
const assert = require("assert");
const { Builder, By } = require("selenium-webdriver");

describe("Disabled Element", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("chrome").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now let's wire up our test.

```
// filename: test/disabledElement.spec.js
// ...
  it("dropdown list should contain disabled and enabled elements", async function() {
    driver.get("http://the-internet.herokuapp.com/dropdown");
    const dropdownList = await driver.findElements(By.css("option"));
    assert((await dropdownList[0].isEnabled()) === false);
  });
});
```

After loading the page, we find all of the elements that have an option tag (which are all of the items in the dropdown list). This returns a list of elements, so we use the first one (which is the one with the text of `'Please select an option'`).

Once we have the element we want we see if it's enabled (with `.isEnabled`) and assert based on the response.

Also, since we grabbed all of the dropdown list options, we can easily test the opposite case by checking the second or third option in the list.

```
// filename: test/disabledElement.spec.js
// ...
  it("dropdown list should contain disabled and enabled elements", async function() {
        // ...
    assert(await dropdownList[1].isEnabled());
  });
});
```

Expected Behavior

When we save this file and run it (e.g., `mocha` from the command-line) here is what will happen:

- Open a browser
- Visit the page
- Grab all dropdown list elements
- Assert that the first element in the list is not enabled
- Assert that the second element in the list is enabled
- Close the browser

Outro

Hopefully this tip has helped make the simple task of seeing if an element is enabled or disabled more approachable.

Happy Testing!

Chapter 8
How To Select From a Dropdown List

The Problem

Selecting from a dropdown list seems like one of those simple things. Just grab the list by it's element and select an item within it based on the text you want.

While it sounds pretty straightforward, there is a bit more finesse to it.

Let's take a look at a couple of different approaches.

An Example

First let's pull in our requisite libraries, declare the test class, and wire up some simple setup and teardown methods.

```
// filename: spec/dropdown.spec.js
const assert = require("assert");
const { Builder, By } = require("selenium-webdriver");

describe("Dropdown", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("chrome").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now lets' wire up our test.

```
// filename: spec/dropdown.spec.js
// ...
  it("select the hard way", async function() {
    await driver.get("http://the-internet.herokuapp.com/dropdown");
    const options = await driver.findElements(By.css("#dropdown option"));
    let selection;
    for (const option in options) {
      if ((await options[option].getText()) === "Option 1") {
        await options[option].click();
        break;
      }
    }
    for (const option in options) {
      if (await options[option].isSelected()) {
        selection = await options[option].getText();
        break;
      }
    }
    assert(selection === "Option 1");
  });
// ...
```

After visiting the example application we find the dropdown list by its ID and store it in a variable. We then find each clickable element in the dropdown list (e.g., each `option`) with `findElements`.

Grabbing all of the options returns a collection that we iterate over and when the text matches what we want it will click on it.

We finish the test by performing a check to see that our selection was made correctly. This is done by looping over the dropdown options collection one more time. This time we're getting the text of the item that was selected, storing it in a variable, and then making an assertion against it.

While this works, there is a simpler way to do this.

Another Example

```
// filename: spec/dropdown.spec.js
// ...
  it("select the simpler way", async function() {
    await driver.get("http://localhost:9292/dropdown");
    await driver
      .findElement(
        By.xpath("//*[@id='dropdown']/option[contains(text(),'Option 1')]")
      )
      .click();
    const selection = await driver
      .findElement(By.css('#dropdown option[selected="selected"]'))
      .getText();
    assert(selection === "Option 1");
  });
});
```

Similar to the first example, we are selecting a dropdown list option by its text. But instead of first finding the dropdown list, we instead employ a more robust locator (e.g., XPath) to find an element within the dropdown list that contains our target text -- and click it.

We then determine what has been selected in the dropdown list by using one more powerful locator (e.g., a CSS selector). This time, looking to see which option has the `selected` attribute, and grabbing its text. We use this found text to assert that the correct item was chosen.

Expected Behavior

If you save this file with either of these examples and run it (e.g., `mocha` from the command-line) here is what will happen:

- Open the browser
- Visit the example application
- Find the dropdown list
- Select the requested item from the dropdown list
- Assert that the selected option is the one you expect
- Close the browser

Outro

Hopefully this tip will help you breeze through selecting items from a dropdown list.

Happy Testing!

Chapter 9

How To Work With Hovers

The Problem

If you need to work with mouse hovers in your tests it may not be obvious how to do this with Selenium. And a quick search through the documentation can easily leave you befuddled forcing you to go spelunking through StackOverflow for the solution.

A Solution

By leveraging Selenium's Actions we can handle more complex user interactions like hovers. This is done by telling Selenium which element we want to move the mouse to, and then performing what we need to after.

Let's dig in with an example.

An Example

Our example application is available here on the-internet. It has a few avatars displayed in a grid layout. When you hover over each of them, they display additional user information and a link to view a full profile.

Let's write a test that will hover over the first avatar and make sure that this additional information appears.

First we'll include our requisite libraries, declare the test class, and wire up some simple setup and teardown methods.

```
// filename: test/hovers.spec.js
const assert = require("assert");
const { Builder, By, Key } = require("selenium-webdriver");

describe("Hovers", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("firefox").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now let's write our test.

```
// filename: test/hovers.spec.js
// ...
  it("displays caption on hover", async function() {
    await driver.get("http://the-internet.herokuapp.com/hovers");
    const avatar = await driver.findElement(By.css(".figure"));
    await driver
      .actions({ bridge: true })
      .move({ origin: avatar })
      .perform();
    const caption = await driver.findElement(By.css(".figcaption"));
    assert(caption.isDisplayed());
  });
});
```

After loading the page we find the first avatar and store it in a variable (`avatar`). We then use the `.move` function and feed it the avatar variable (which triggers the hover).

We then check to see if the additional user information is displayed with `.isDisplayed` in our assertion.

Expected Behavior

When we save this file and run it (e.g., `mocha` from the command-line) here is what will happen:

- Open the browser
- Visit the page
- Hover over the first avatar

- Assert that the caption appeared on the page
- Close the browser

Outro

Happy Testing!

Chapter 10
How To Work With JavaScript Alerts

The Problem

If your application triggers any JavaScript pop-ups (a.k.a. alerts, dialogs, etc.) then you need to know how to handle them in your Selenium tests.

A Solution

Built into Selenium is the ability to switch to an alert window and either accept or dismiss it. This way your tests can continue unencumbered by dialog boxes that may feel just out of reach.

Let's dig in with an example.

An Example

Our example application is available here on the-internet. It has various JavaScript Alerts available (e.g., an alert, a confirmation, and a prompt). Let's demonstrate testing a confirmation dialog (e.g., a prompt which asks the user to click `Ok` or `Cancel`).

First, we'll include our requisite libraries, declare the test class, and wire up some simple setup and teardown methods.

```
// filename: test/js-alerts.spec.js
const assert = require("assert");
const { Builder, By, Key } = require("selenium-webdriver");

describe("JS Alerts", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("chrome").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now let's write our test.

```
// filename: test/js-alerts.spec.js
// ...
  it("general use", async function() {
    await driver.get("http://the-internet.herokuapp.com/javascript_alerts");
    await driver.findElement(By.css("ul > li:nth-child(2) > button")).click();
    const popup = await driver.switchTo().alert();
    popup.accept();
    const result = await driver.findElement(By.id("result")).getText();
    assert(result == "You clicked: Ok");
  });
});
```

A quick glance at the page's markup shows that there are no unique IDs on the buttons. So to click on the second button (to trigger the JavaScript confirmation dialog) we need a mildly clever locator which targets the second item in the unordered list.

After we click the button to trigger the JavaScript Alert we use Selenium's `switchTo().alert()` to focus on the JavaScript pop-up and use `.accept()` to click `Ok` (if we wanted to click `Cancel` we would have used `.dismiss()`).

After accepting the alert, our main browser window will automatically regain focus and the page will display the result that we chose. This text is what we use for our assertion, making sure that the words `You clicked: Ok` are displayed on the page.

Expected Behavior

When we save this file and run it (e.g., `mocha` from the command-line) here is what will happen:

- Open the browser
- Load the page
- Click the second button on the page
- JavaScript Confirmation Alert appears
- Accept the JavaScript Confirmation Alert
- Assert that the result on the page is what we expect
- Close the browser

Outro

Happy Testing!

Chapter 11
How To Work With HTML Data Tables

The Problem

Odds are at some point you've come across the use of tables in a web application to display data or information to a user, giving them the option to sort and manipulate it. Depending on your application it can be quite common and something you will want to write an automated test for.

But when the table has no helpful, semantic markup (e.g. easy to use `id` or `class` attributes) it quickly becomes more difficult to work with and write tests against it. And if you're able to pull something together, it will likely not work against older browsers.

A Solution

You can easily traverse a table through the use of CSS pseudo-classes.

But keep in mind that if you care about older browsers (e.g., Internet Explorer, et al), then this approach won't work on them. In those cases your best bet is to find a workable solution for the short term and get a front-end developer to update the table with helpful attributes.

A quick primer on Tables and CSS pseudo-classes

Understanding the broad strokes of an HTML table's structure goes a long way in writing effective automation against it. So here's a quick primer.

A table has...

- a header (e.g. `<thead>`)
- a body (e.g. `<tbody>`).
- rows (e.g. `<tr>`) -- horizontal slats of data
- columns -- vertical slats of data

Columns are made up of cells which are...

- a header (e.g., `<th>`)
- one or more standard cells (e.g., `<td>` -- which is short for table data)

CSS pseudo-classes work by walking through the structure of an object and targeting a specific part of it based on a relative number (e.g. the third `<td>` cell from a row in the table body). This works well with tables since we can grab all instances of a target (e.g. the third `<td>` cell from each `<tr>` in the table body) and use it in our test -- which would give us all of the data for the third column.

Let's step through some examples for a common set of table functionality like sorting columns in ascending and descending order.

An Example

NOTE: You can see the application under test here. It's an example from the-internet. In the example there are 2 tables. We will start with the first table and then work with the second.

We kick things off by pulling in our requisite libraries, declare our test class, and wire up some test setup and teardown methods.

```
// filename: test/tables.spec.js
const assert = require("assert");
const { Builder, By, Key } = require("selenium-webdriver");

describe("Tables", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("chrome").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Here is the markup from the first table example we're working with. Note that it does not have any `id` or `class` attributes.

```html
<table id="table1" class="tablesorter">
    <thead>
      <tr>
        <th><span>Last Name</span></th>
        <th><span>First Name</span></th>
        <th><span>Email</span></th>
        <th><span>Due</span></th>
        <th><span>Web Site</span></th>
        <th><span>Action</span></th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>Smith</td>
        <td>John</td>
        <td>jsmith@gmail.com</td>
        <td>$50.00</td>
        <td>http://www.jsmith.com</td>
        <td>
          <a href='#edit'>edit</a>
          <a href='#delete'>delete</a>
        </td>
      </tr>
```

There are 6 columns (`Last Name`, `First Name`, `Email`, `Due`, `Web Site`, and `Action`). Each one is sortable by clicking on the column header. The first click should sort them in ascending order, the second click in descending order.

There is a small sampling of data in the table to work with (4 rows worth). So we should be able to sort the data, grab it, and confirm that it sorted correctly. So lets do that in our first test with the `Due` column using a CSS pseudo Class.

```
// filename: test/tables.spec.js
// ...
  it("should sort number column in ascending order", async function() {
    await driver.get("http://the-internet.herokuapp.com/tables");
    await driver
      .findElement(By.css("#table1 thead tr th:nth-of-type(4)"))
      .click();
    const due_column = await driver.findElements(
      By.css("#table1 tbody tr td:nth-of-type(4)")
    );
    let dues = [];
    for (const entry in due_column) {
      const text = await due_column[entry].getText();
      dues.push(Number(text.replace("$", "")));
    }
    assert(dues === dues.sort());
  });
// ...
```

After loading the page we find and click the column heading that we want with a CSS pseudo-class (e.g. `#table1 thead tr th:nth-of-type(4)`). This locator targets the 4th `<th>` element in the table heading section (e.g., `<thead>`) (which is the `Due` column heading).

We then use another pseudo-class to find all `<td>` elements within the `Due` column by looking for the 4th `<td>` of each row in the table body. Once we have them we grab each of their text values, clean them up (`.replace('$','')`), convert them to a number (`Number()`), and store them all in a array called `dues`. We then compare this collection to a sorted version of itself to see if they match. If they do, then the `Due` column was sorted in ascending order and the test will pass.

If we wanted to test for descending order, we would need to click the `Due` heading twice after loading the page. Other than that the code is identical except for the assertion which is checking the same thing but reversing the sort order.

```
// filename: test/tables.spec.js
// ...
  it("should sort number column in descending order", async function() {
    await driver.get("http://the-internet.herokuapp.com/tables");
    await driver
      .findElement(By.css("#table1 thead tr th:nth-of-type(4)"))
      .click();
    await driver
      .findElement(By.css("#table1 thead tr th:nth-of-type(4)"))
      .click();
    const due_column = await driver.findElements(
      By.css("#table1 tbody tr td:nth-of-type(4)")
    );
    let dues = [];
    for (const entry in due_column) {
      const text = await due_column[entry].getText();
      dues.push(Number(text.replace("$", "")));
    }
    assert(dues === dues.sort().reverse());
  });
// ...
```

We can easily use this approach to test a different column (e.g., one that doesn't deal with numbers) and see that it gets sorted correctly too. Here's a test that exercises the `Email` column.

```
// filename: test/tables.spec.js
// ...
  it("should sort text column in ascending", async function() {
    await driver.get("http://the-internet.herokuapp.com/tables");
    await driver
      .findElement(By.css("#table1 thead tr th:nth-of-type(3)"))
      .click();
    const email_column = await driver.findElements(
      By.css("#table1 tbody tr td:nth-of-type(3)")
    );
    let emails = [];
    for (const entry in email_column) {
      emails.push(await email_column[entry].getText());
    }
    assert(emails === emails.sort());
  });
// ...
```

The mechanism for this is the same as before, except that we don't need to clean the text up or convert it before performing our assertion.

But What About Better Locators?

Here is what the markup of our original table would look like with some helpful attributes added in. It's also the markup from the second table on our application under test.

```
<table id="table2" class="tablesorter">
    <thead>
      <tr> <th><span class='last-name'>Last Name</span></th>
        <th><span class='first-name'>First Name</span></th>
        <th><span class='email'>Email</span></th>
        <th><span class='dues'>Due</span></th>
        <th><span class='web-site'>Web Site</span></th>
        <th><span class='action'>Action</span></th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td class='last-name'>Smith</td>
        <td class='first-name'>John</td>
        <td class='email'>jsmith@gmail.com</td>
        <td class='dues'>$50.00</td>
        <td class='web-site'>http://www.jsmith.com</td>
        <td class='action'>
          <a href='#edit'>edit</a>
          <a href='#delete'>delete</a>
        </td>
      </tr>
```

With these new attributes the locators in our sorting tests become a lot simpler and more expressive. Let's write a new `Due` ascending order test to demonstrate.

```
// filename: test/tables.spec.js
// ...
  it("sort number column in ascending order with helpful locators", async function() {
    await driver.get("http://the-internet.herokuapp.com/tables");
    driver.findElement(By.css("#table2 thead .dues")).click();
    const due_column = await driver.findElements(By.css("#table2 tbody .dues"));
    let dues = [];
    for (const entry in due_column) {
      const text = await due_column[entry].getText();
      dues.push(Number(text.replace("$", "")));
    }
    assert(dues === dues.sort());
  });
});
```

Not only will these selectors work in current and legacy browsers, but they are also more resilient to changes in the table layout since they are not using hard-coded numbers that rely on the column order.

Expected Behavior

When we save this file and run it (e.g., `mocha` from the command-line) here is what will happen:

- Browser opens
- Load the page
- Click the column heading
- Grab the values for the target column
- Assert that the column is sorted in the correct order (ascending or descending depending on the test)
- Close the browser

Outro

Oftentimes you'll need to automate something which is lacking in reliable, semantic locators, but with CSS pseudo-classes you can cover a lot of ground in your tests by enabling a bit of CSS gymnastics.

But you can greatly simplify test automation efforts by adding some helpful attributes to key elements in the application under test.

Happy Testing!

Chapter 12
How To Work with Frames

The Problem

On occasion you'll run into a relic of the front-end world -- frames. And when writing a test against them, you can easily get tripped up if you're not paying attention.

A Solution

Rather than gnash your teeth when authoring your tests, you can easily work with the elements in a frame by telling Selenium to switch to that frame first. Then the rest of your test should be business as usual.

Let's dig in with some examples.

An Example

We'll first need to pull in our requisite libraries, declare our test class, and wire up some setup and teardown methods for our tests.

```
// filename: test/frames.spec.js
const assert = require("assert");
const { Builder, By } = require("selenium-webdriver");

describe("Frame Test", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("firefox").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now onto our test. In it we'll step through an example of nested frames which can be found on the-internet.

```
// filename: test/frames.spec.js
  it("nested_frames", async function() {
    await driver.get("http://the-internet.herokuapp.com/nested_frames");
    await driver
      .switchTo()
      .frame(await driver.findElement(By.name("frame-top")));
    await driver
      .switchTo()
      .frame(await driver.findElement(By.name("frame-middle")));
    let content = await driver.findElement(By.id("content")).getText();
    assert.equal(content, "MIDDLE");
  });
// ...
```

With Selenium's `.switchTo().frame` method we can easily switch to the frame we want. It accepts either a numbered index or WebElement (e.g., the result of a `findElement`).

In order to get the text of the middle frame (e.g., a frame nested within another frame), we first need to switch to the parent frame (e.g., the top frame) and then switch to the child frame (e.g., the middle frame). Once we've done that we're able to find the element we need, grab its text, and assert that it's what we expect.

While this example helps illustrate the point of frame switching, it's not very practical.

A More Practical Example

Here is a more likely example you'll run into -- working with a WYSIWYG (What You See Is What You Get) Editor like TinyMCE. You can see the page we're testing here.

```
// filename: test/frames.spec.js
// ...
  it("iframes", async function() {
    await driver.get("http://the-internet.herokuapp.com/tinymce");
    await driver.switchTo().frame(await driver.findElement(By.id("mce_0_ifr")));
    const editor = await driver.findElement(By.id("tinymce"));
    let beforeText = await editor.getText();
    await editor.clear();
    await editor.sendKeys("Hello World!");
    let afterText = await editor.getText();
    assert.notEqual(afterText, beforeText);
  });
});
```

Once the page loads we switch into the frame that contains TinyMCE and...

- grab the original text and store it
- clear and input new text
- grab the new text value
- assert that the original and new texts are not the same

Keep in mind that if we need to access a part of the page outside of the frame we are currently in we'll need to switch to it. Thankfully Selenium has method that enables us to quickly jump back to the top level of the page -- `switchTo.defaultContent()`.

Here is what that looks like in practice.

```
// filename: test/frames.spec.js
  it("iframes", async function() {
    // ...
    await driver.switchTo().defaultContent();
    let txt = await driver.findElement(By.css("h3")).getText();
    assert.equal(txt, "An iFrame containing the TinyMCE WYSIWYG Editor");
  });
});
```

Expected Behavior

If we save the file and run it (e.g., `mocha` from the command-line) here is what will happen:

Example 1

- Open the browser
- Visit the page
- Switch to the nested frame
- Grab the text from the frame and assert that Selenium is in the correct place
- Close the browser

Example 2

- Open the browser
- Visit the page
- Switch to the frame that contains the TinyMCE editor
- Grab and clear the text in the editor
- Input and grab new text in the edtitor
- Assert that the original and new text entries don't match
- Switch to the top level of the page
- Grab the text from the top of the page and assert that it's not empty
- Close the browser

Outro

Now you're ready to handily defeat frames when they cross your path.

Happy Testing!

Chapter 13
How To Work with Multiple Windows

The Problem

Occasionally you'll run into a link or action in the application you're testing that will open a new window. In order to work with both the new and originating windows you'll need to switch between them.

On the face of it, this is a pretty straightforward concept. But lurking within it is a small gotcha to watch out for that will bite you in some browsers and not others.

Let's step through a couple of examples to demonstrate.

An Example

First we'll need to pull in our requisite libraries, declare our test class, and wire up some test setup and teardown methods.

```
// filename: test/multiple-windows.spec.js
const assert = require("assert");
const { Builder, By, Key } = require("selenium-webdriver");

describe("Multiple Windows", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("chrome").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now let's write a test that exercises new window functionality from an application. In this case, we'll be using the new window example found on the-internet.

```
// filename: test/multiple-windows.spec.js
// ...
  it("non-deterministic switching", async function() {
    await driver.get("http://the-internet.herokuapp.com/windows");
    await driver.findElement(By.css(".example a")).click();
    const windowHandles = await driver.getAllWindowHandles();
    await driver.switchTo().window(windowHandles[0]);
    assert((await driver.getTitle()) !== "New Window");
    await driver.switchTo().window(windowHandles[windowHandles.length - 1]);
    assert((await driver.getTitle()) === "New Window");
  });
// ...
```

After loading the page we click the link which spawns a new window. We then grab the window handles (a.k.a. unique identifier strings which represent each open browser window) and switch between them based on their order (assuming that the first one is the originating window, and that the last one is the new window). We round this test out by performing a simple check against the title of the page to make sure Selenium is focused on the correct window.

While this may seem like a good approach, it can present problems later. That's because the order of the window handles is not consistent across all browsers. Some return in the order opened, others alphabetically.

Here's a more resilient approach. One that will work across all browsers.

A Better Example

```
// filename: test/multiple-windows.spec.js
// ...
  it("browser agnostic switching", async function() {
    await driver.get("http://the-internet.herokuapp.com/windows");
    const windowHandlesBefore = await driver.getAllWindowHandles();
    await driver.findElement(By.css(".example a")).click();
    const windowHandlesAfter = await driver.getAllWindowHandles();
    const newWindow = windowHandlesAfter.find(
      handle => !windowHandlesBefore.includes(handle)
    );
    await driver.switchTo().window(windowHandlesBefore[0]);
    assert((await driver.getTitle()) !== "New Window");
    await driver.switchTo().window(newWindow);
    assert((await driver.getTitle()) === "New Window");
  });
});
```

After loading the page we store the window handles in a variable (e.g., `windowHandlesBefore`) and then proceed with clicking the new window link.

Now that we have two windows open we grab all of the window handles again (e.g., `windowHandlesAfter`) and search through them to find the new window handle (e.g., the handle that's in the new window handle collection but not the initial one). We store the result in another variable (e.g., `newWindow`) and then switch between the windows. Each time checking the page title to make sure the correct window is in focus.

Expected Behavior

When we save this file and run it (e.g., `mocha` from the command-line) here is what will happen:

- Open the browser
- Visit the page
- Find the window handle for the current window
- Click a link that opens a new window
- Find the window handle out of all available window handles
- Switch between the windows
- Assert that the correct window is in focus
- Close the browser

Outro

Hat tip to Jim Evans for providing the info for this tip.

Happy Testing!

Chapter 14

How To Press Keyboard Keys

The Problem

On occasion you'll come across functionality that requires the use of keyboard key presses in your tests.

Perhaps you'll need to tab to traverse from one portion of the page to another, back out of some kind of menu or overlay with the escape key, or even submit a form with Enter.

But how do you do it and where do you start?

A Solution

You can easily issue a key press by using the `sendKeys` command.

This can be done to a specific element, or generically with Selenium's Action Builder (link). Either approach will send a key press. The latter will send it to the element that's currently in focus in the browser (so you don't have to specify a locator along with it), whereas the prior approach will send the key press directly to the element found.

Let's step through a couple of examples.

An Example

First we'll include our requisite libraries, declare the test class, and wire up some simple setup and teardown methods.

```
// filename: test/keyboard-keys.spec.js
const assert = require("assert");
const { Builder, By, Key } = require("selenium-webdriver");

describe("Keyboard Keys", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("firefox").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now we can wire up our test.

Let's use an example from the-internet that will display what key has been pressed (link). We'll use the result text that gets displayed to perform our assertion.

```
// filename: test/keyboard-keys.spec.js
  it("send keys", async function() {
    await driver.get("http://the-internet.herokuapp.com/key_presses");
    await driver.findElement(By.id("target")).sendKeys(Key.SPACE);
    assert(
      (await driver.findElement(By.id("result")).getText()) ===
        "You entered: SPACE"
    );
  });
});
```

After visiting the page we find an element that's both visible and interactable with the keyboard (e.g., the input element on the page) and send the space key to it (e.g., `.sendKeys(Key.SPACE`). Then we grab the resulting text (e.g., `driver.findElement(By.id('result')).getText()`) and assert that it says what we expect (e.g., `'You entered: SPACE'`).

Alternatively, we can issue a key press without finding the element first, by using the Action Builder.

```
// filename: test/keyboard-keys.spec.js
// ...
  it("send keys", async function() {
        // ...
    await driver
      .actions({ bridge: true })
      .sendKeys(Key.TAB)
      .perform();
    assert(
      (await driver.findElement(By.id("result")).getText()) ==
        "You entered: TAB"
    );
  });
});
```

NOTE: In order for this approach to work in browsers other than Firefox, we need to specify **`{bridge: true}`** when calling **`driver.actions`** (e.g., **`driver.actions({ bridge: true })`**). You can see the documentation for further details.

Expected Behavior

When we save this file and run it (e.g. `mocha` from the command-line) here is what will happen:

- Open the browser
- Visit the page
- Find the element and send the space key to it
- Find the result text on the page and check that it's what we expect
- Send the tab key to the element that's currently in focus
- Find the result text on the page and check that it's what we expect
- Close the browser

Outro

If you have a specific element that you want to issue key presses to, then finding the element first is the way to go. But if you don't have a receiving element, or you need to string together multiple key presses, then the Action Builder is what you should reach for.

Happy Testing!

Chapter 15
How To Right-click

The Problem

Sometimes you'll run into an app that has functionality hidden behind a right-click menu (a.k.a. a context menu). These menus tend to be system level menus that are untouchable by Selenium. So how do you test this functionality?

A Solution

By leveraging Selenium's Actions we can issue a right-click command (a.k.a. a `contextClick`).

You could then select an option from the menu by traversing it with keyboard keys (if a system dialog) or through `findElement` (if a rendered element). It depends on how the application under test has implemented it.

Let's dig in with an example.

An Example

Let's start by pulling in our requisite libraries, declare the test class, and wire up some simple setup and teardown methods.

```
// filename: test/right-click.spec.js
const assert = require("assert");
const { Builder, By, Key } = require("selenium-webdriver");

describe("Right click", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("firefox").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now we're ready to write our test.

We'll use an example from the-internet that will trigger a JavaScript alert when when we right-click on a specific area of the page (link). It will say `You selected a context menu`. We'll grab this text and use it to assert that the menu was actually triggered.

```
// filename: test/right-click.spec.js
// ...
  it("displays browser context menu", async function() {
    await driver.get("http://the-internet.herokuapp.com/context_menu");
    const menuArea = await driver.findElement(By.id("hot-spot"));
    await driver
      .actions({ bridge: true })
      .contextClick(menuArea)
      .perform();
    const alertText = await driver
      .switchTo()
      .alert()
      .getText();
    assert(alertText === "You selected a context menu");
  });
});
```

Expected Behavior

When we save this file and run it (e.g., `mocha`) from the command-line) here is what will happen:

- Open the browser and visit the page
- Find and right-click the area which will render a custom context menu
- Select the context menu option with keyboard keys
- JavaScript alert appears
- Grab the text of the JavaScript alert
- Assert that the text from the alert is what we expect
- Close the browser

Outro

Happy Testing!

Chapter 16

How To Opt-out of A/B Tests

The Problem

Occasionally when running tests you may see unexpected behavior due to A/B testing (a.k.a. split testing) of the application you're working with.

In order to keep your tests running without issue we need a clean way to opt-out of these split tests.

A quick primer on A/B testing

Split testing is a simple way to experiment with an application's features to see which changes lead to higher user engagement.

A simple example would be testing variations of an e-mail landing page to see if more people sign up. In such a split test there would be the control (how the application looks and behaves now) and variants (e.g., 2 or 3 changes that could include changing text or images on the page, element positioning, color of the submit button, etc).

Once the variants are configured, they are put into rotation, and the experiment starts. During this experiment each user will see a different version of the feature and their engagement with it will be tracked. Split tests live for the length of the experiment or until a winner is found (e.g., tracking indicates that a variant converted higher than the control). If no winner is found, new variants may be created and another experiment tried. If a winner is found, then the winning variant becomes the new control and the feature gets updated accordingly.

A Solution

Thankfully there are some standard opt-out mechanisms built into A/B testing platforms. They tend to come in the form of an appended URL or forging a cookie. Let's dig in with an example of each approach with a popular A/B testing platform, Optimizely.

An Example

Our example page is from the-internet and can be seen here. There are three different versions of the page that are available. On each page the heading text will vary:

- Control: `A/B Test Control`
- Variation 1: `A/B Test Variation 1`
- Opt-out: `No A/B Test`

Let's kick things off by loading our requisite libraries, declare our test class, and wire up some setup and teardown methods for our tests.

```
// filename: test/abOptOut.spec.js
const assert = require("assert");
const { Builder, By } = require("selenium-webdriver");

describe("A/B opt-out", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("chrome").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now let's wire up our first test to step through loading the split testing page and verifying that the text changes after we forge an opt-out cookie.

```
// filename: test/abOptOut.spec.js
// ...
  it("with cookie after visiting page", async function() {
    await driver.get("http://the-internet.herokuapp.com/abtest");
    let headingText = await driver.findElement(By.css("h3")).getText();
    if (headingText.startsWith("A/B Test")) {
      await driver
        .manage()
        .addCookie({ name: "optimizelyOptOut", value: "true" });
      await driver.navigate().refresh();
      headingText = await driver.findElement(By.css("h3")).getText();
    }
    assert.equal(headingText, "No A/B Test");
  });
// ...
```

After navigating to the page we confirm that we are in one of the A/B test groups by grabbing the heading text and checking to see if it matches what we expect. After that we add the opt-out cookie, refresh the page, and then confirm that we are no longer in the A/B test group by checking the heading text again.

We could also load the opt-out cookie before navigating to this page.

```
// filename: test/abOptOut.spec.js
  it("with cookie before visiting page", async function() {
    await driver.get("http://the-internet.herokuapp.com");
    await driver
      .manage()
      .addCookie({ name: "optimizelyOptOut", value: "true" });
    await driver.get("http://the-internet.herokuapp.com/abtest");
    assert.equal(
      await driver.findElement(By.css("h3")).getText(),
      "No A/B Test"
    );
  });
// ...
```

Here we are navigating to the main page of the site first and then adding the opt-out cookie. After that we navigate to the split test page and then perform our check. Alternatively, we could opt out without forging a cookie. Instead we just need to append an opt out parameter to the URL.

```
// filename: test/abOptOut.spec.js
// ...
  it("with opt out URL", async function() {
    await driver.get(
      "http://the-internet.herokuapp.com/abtest?optimizely_opt_out=true"
    );
    await driver
      .switchTo()
      .alert()
      .dismiss();
    assert.equal(
      await driver.findElement(By.css("h3")).getText(),
      "No A/B Test"
    );
  });
});
```

By appending `?optimizely_opt_out=true` we achieve the same outcome as before. Keep in mind that this approach triggers a JavaScript alert, so we have to switch to and dismiss it (e.g., `driver.switchTo().alert().dismiss()`) before performing our check.

Expected Behavior

When we save this file and run it (e.g., `mocha` from the command-line) here is what will happen with either of the tests:

- Open the browser
- Opt-out of the split tests (either by cookie or appended URL)
- Visit the split testing page
- Grab the header text
- Confirm that the session is opted out of the split test
- Close the browser

Outro

Happy Testing!

Chapter 17

How To Access Basic Auth

The Problem

Sometimes you'll work with applications that are secured behind Basic HTTP Authentication (a.k.a. Basic Auth). In order to access them you'll need to pass credentials to the site when requesting a page. Otherwise you'll get a system level pop-up prompting you for a username and password -- rendering Selenium helpless.

Before Selenium 2 we were able to accomplish this by injecting credentials into a custom header, but now the cool kid way to do it it was something like BrowserMob Proxy. And some people are solving this with browser specific configurations too.

But all of this feels heavy. Instead, let's look at a simple approach that is browser agnostic and quick to setup.

A Solution

By specifying the username and password in the URL when visiting a page with Selenium, we can side-step the system level dialog box and avoid setting up a proxy server. This approach will work for both HTTP or HTTPS pages.

Let's take a look at an example.

An Example

Let's start by requiring our requisite libraries, declare our test class, and wire up some test setup and teardown methods.

```
// filename: test/basic-auth.spec.js
// ...
const assert = require("assert");
const { Builder, By, Key } = require("selenium-webdriver");

describe("Basic Auth", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("chrome").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now to add our test.

```
// filename: test/basic-auth-1.spec.js
// ...
  it("should visit basic auth secured page directly", async function() {
    await driver.get(
      "http://admin:admin@the-internet.herokuapp.com/basic_auth"
    );
    const page_message = await driver
      .findElement(By.css(".example p"))
      .getText();
    assert(
      page_message === "Congratulations! You must have the proper credentials."
    );
  });
});
```

In the test we're loading the page by passing in the username and password in the front of the URL (e.g., `http://admin:admin@`). Once it loads we grab text from the page to make sure we ended up in the right place.

Alternatively, we could have accessed this page before the test (e.g., as part of the test setup). This would have cached the Basic Auth session in the browser, enabling us to visit the page again without having to specify credentials. This is particularly useful if you have numerous pages behind Basic Auth.

```
// filename: test/basic-auth.spec-2.js
// ...
  beforeEach(async function() {
    driver = await new Builder().forBrowser("chrome").build();
    await driver.get(
      "http://admin:admin@the-internet.herokuapp.com/basic_auth"
    );
  });

  // ...

  it("should visit basic auth without credentials after visiting page with them", async
function() {
    await driver.get("http://the-internet.herokuapp.com/basic_auth");
    const page_message = await driver
      .findElement(By.css(".example p"))
      .getText();
    assert(
      page_message === "Congratulations! You must have the proper credentials."
    );
  });
});
```

NOTE: If your application serves both HTTP and HTTPS pages from behind Basic Auth then you will need to load one of each type before executing your test steps. Otherwise you will get authorization errors when switching between HTTP and HTTPS because the browser can't use Basic Auth credentials interchangeably (e.g. HTTP for HTTPS and vice versa).

Expected Behavior

When you save these files and run them (e.g., `mocha`), here is what will happen:

basic-auth-1.spec.js

- Open the browser
- Visit the page using Basic Auth
- Get the page text
- Assert that the text is what we expect
- Close the browser

basic-auth-2.spec.js

- Open the browser
- Visit the page using Basic Auth as part of the test setup
- Visit the page without the Basic Auth credentials (successfully)

- Get the page text
- Assert that the text is what we expect
- Close the browser

Outro

Hopefully this tip will help save you from getting tripped by Basic Auth when you come across it.

Happy Testing!

Chapter 18
How To Visually Verify Your Locators

This is a pseudo guest post from Brian Goad. I've adapted a blog post of his with permission. You can see the original here. Brian is a Test Engineer at Digitalsmiths. You can follow him on Twitter at @bbbco and check out his testing blog here.

The Problem

It's likely that you'll run into odd test behavior that makes you question the locators you're using in a test. But how do you interrogate your locators to make sure they are doing what you expect?

A Solution

By leveraging some simple JavaScript and CSS styling, we can highlight the element on the page that we're targeting. This way we can visually inspect it to make sure it is the one that we want.

Let's take a look at an example.

An Example

For our initial setup let's pull in our requisite libraries, declare our test class, and wire up some setup and teardown methods.

```
// filename: test/highlight-elements.spec.js
const assert = require("assert");
const { Builder, By, Key } = require("selenium-webdriver");

describe("Highlight elements", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("firefox").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now let's create a `highlight` helper function that will accept a found `element` from Selenium and a time to wait (e.g., `duration`) as arguments.

By setting a duration, we can control how long to highlight an element on the page before reverting the styling back. We can also make this an optional argument by setting a default value for it (e.g., 3 seconds).

```
// filename: test/highlight-elements.spec.js
// ...
  async function highlight(element, duration = 2000) {
    // store original style so it can be reset later
    const originalStyle = await element.getAttribute("style");

    // style element with callout (e.g., dashed red border)
    await driver.executeScript(
      "arguments[0].setAttribute(arguments[1], arguments[2])",
      element,
      "style",
      "border: 2px solid red; border-style: dashed;"
    );

    // keep element highlighted for the duration and then revert
    await driver.sleep(duration);
    await driver.executeScript(
      "arguments[0].setAttribute(arguments[1], arguments[2])",
      element,
      "style",
      originalStyle
    );
  }
// ...
```

There are three things going on here.

1. We store the style of the element so we can revert it back when we're done
2. We change the style of the element so it visually stands out (e.g., a red dashed border)
3. We revert the style of the element back after 3 seconds

We're accomplishing the style change through JavaScript's `setAttribute` function. And we're executing it with Selenium's `executeScript` command.

Now to use this in our test is simple, we just prepend a `findElement` command with a call to the `highlight` method.

```
// filename: test/highlight-elements.spec.js
// ...
  it("highlights target element", async function() {
    await driver.get("http://the-internet.herokuapp.com/large");
    await highlight(await driver.findElement(By.id("sibling-2.3")));
  });
});
```

Expected Behavior

When we save this file and run it (e.g., `mocha` from the command-line) here is what will happen.

- Browser opens
- Load the page
- Find the element
- Change the styling of the element so it has a red dashed-line border
- Wait 3 seconds
- Revert the styling to remove the border
- Browser closes

Outro

Alternatively, you could use the developer tools and inspect the locators in a test one at a time for debugging. This approach is more helpful if you want to have some kind of real time review of what's happening as your tests run.

Happy Testing!

Chapter 19
How To Add Growl Notifications To Your Tests

The Problem

Good test reports are a fundamental component of successful test automation. But running down a test failure by looking at a test report can be a real pain sometimes.

Leaving you with no choice but to roll up your sleeves and get your hands dirty with debug statements, or step through things piece by piece -- all for the sake of trying to track down a transient issue.

A Solution

By leveraging something like jQuery Growl you can output non-interactive debugging statements directly to the page you're testing. This way you can see helpful information and more-likely correlate it to the test actions that are being taken. This can a boon for your test runs when coupled with screenshots and/or video recordings of your test runs

Let's step through an example of how to set this up.

An Example

First we'll need to pull in our requisite libraries, declare our test suite, and wire up some setup and teardown methods.

```
// filename: test/growl.spec.js
const assert = require("assert");
const { Builder, By, Key } = require("selenium-webdriver");

describe("Growl", function() {
  let driver;

  beforeEach(async function() {
    driver = await new Builder().forBrowser("firefox").build();
  });

  afterEach(async function() {
    await driver.quit();
  });
// ...
```

Now for our test. We'll need to visit the page we want to display notifications on and do some work with JavaScript to load jQuery, jQuery Growl, and styles for jQuery Growl. After that we can issue commands to jQuery Growl to make notification messages display on the page.

```
// filename: test/growl.spec.js
// ...
  it("runs and shows growl debugging output", async function() {
    await driver.get("http://the-internet.herokuapp.com");

    // check for jQuery on the page, add it if needbe
    await driver.executeScript(
      `if (!window.jQuery) { var jquery = document.createElement('script'); jquery.type
= 'text/javascript'; jquery.src =
'https://ajax.googleapis.com/ajax/libs/jquery/2.2.4/jquery.min.js'; document.
getElementsByTagName('head')[0].appendChild(jquery);}`
    );

    // use jQuery to add jquery-growl to the page
    await driver.executeScript(
      "$.getScript('http://the-internet.herokuapp.com/js/vendor/jquery.growl.js');"
    );

    // use jQuery to add jquery-growl styles to the page
    await driver.executeScript(
      `$('head').append("<link rel=stylesheet
href=http://the-internet.herokuapp.com/css/jquery.growl.css type=text/css />");`
    );

    await driver.sleep(1000);

    await driver.executeScript("$.growl({ title: 'GET', message: '/' });");

    await driver.sleep(3000);
  });
});
```

In addition to loading the scripts and styles, we also need to have Selenium work at the right pace for our needs. So we add a slight delay (e.g., 1 second) before the call to jquery-growl to make sure it will be available. After that, we add a slightly longer delay (e.g., 3 seconds) so the rendered growl notification stays on the page long enough for viewing.

If we wanted to see color-coded notifications, then we could use one of the following:

```
// filename: test/growl.spec.js
// ...
  it("runs and shows growl debugging output", async function() {
    // ...
    await driver.executeScript(
      "$.growl.error({ title: 'ERROR', message: 'your error message goes here' });"
    );
    await driver.executeScript(
      "$.growl.notice({ title: 'Notice', message: 'your notice message goes here' });"
    );
    await driver.executeScript(
      "$.growl.warning({ title: 'Warning!', message: 'your warning message goes here'
});"
    );

    await driver.sleep(3000);
  });
});
```

Expected Behavior

When we save this file and run it (e.g., `mocha`) here is what will happen:

- Browser opens
- Load the page
- Add jQuery, jQuery Growl, and jQuery Growl notifications to the page
- Display a set of notification messages in the top-right corner of the page
- Notification messages disappear
- Browser closes

Outro

In order to use this approach, you will need to load jQuery Growl on every page you want to display output to -- which can be a bit of overhead. But if you want rich messaging like this then that's the price you have to pay (unless you can get your team to add it to the application under test).

I'd like to give a big thanks to Jon Austen (GitHub) for giving me the idea to use jQuery Growl with Selenium.

Happy Testing!