

[Technologies ▾](#)[References & Guides ▾](#)[Feedback ▾](#)[Sign in !\[\]\(d66ff64371a51729ac8c1cdaa685ba6f_img.jpg\)](#)

Setting up your own test automation environment

[← Previous](#)[↑ Overview: Cross browser testing](#)

In this article, we will teach you how to install your own automation environment and run your own tests using Selenium/WebDriver and a testing library such as selenium-webdriver for Node. We will also look at how to integrate your local testing environment with commercial tools like the ones discussed in the previous article.

Prerequisites:	Familiarity with the core HTML, CSS, and JavaScript languages; an idea of the high level principles of cross browser testing, and automated testing.
Objective:	To show how to set up a Selenium testing environment locally and run tests with it, and how to integrate it with tools like Sauce Labs and BrowserStack.

Selenium

🔗 [Selenium](#) is the most popular browser automation tool. There are other ways, but the best way to use Selenium is via WebDriver, a powerful API that builds on top of Selenium and makes calls to a browser to automate it, carrying out actions such as "open this web page", "move over this element on the page", "click this link", "see whether the link opens this URL", etc. This is ideal for running automated tests.

How you install and use WebDriver depends on what programming environment you want to use to write and run your tests. Most popular environments have available a package or framework that will install WebDriver and the bindings required to communicate with WebDriver using this language, for example Java, C#, Ruby, Python, JavaScript (Node), etc. See 🔗 [Setting Up a Selenium-WebDriver Project](#) for more details of Selenium setups for different languages.

Different browsers require different drivers to allow WebDriver to communicate with and control them. See 🔗 [Platforms Supported by Selenium](#) for more information on where to get browser drivers from, etc.


We will cover writing and running Selenium tests using Node.js, as it is quick and easy to get started, and a more familiar environment for front end devs.

📌 **Note:** If you want to find out how to use WebDriver with other server-side environments, also check out 🔗 [Platforms Supported by Selenium](#) for some useful links.

Setting up Selenium in Node


1. To start with, set up a new npm project, as discussed in [Setting up Node and npm](#) in the last chapter. Call it something different, like `selenium-test`.
2. Next, we need to install a framework to allow us to work with Selenium from inside Node. We are going to choose selenium's official 🔗 [selenium-webdriver](#), as the documentation seems fairly up-to-date, and it is well-maintained. If you want a different options, 🔗 [webdriver.io](#) and 🔗 [nightwatch.js](#) are also good options. To install selenium-webdriver, run the following command, making sure you are inside your project folder:

```
npm install selenium-webdriver
```

 **Note:** It is still a good idea to follow these steps even if you previously installed selenium-webdriver and downloaded the browser drivers. You should make sure that everything is up-to-date.

Next, you need to download the relevant drivers to allow WebDriver to control the browsers you want to test. You can find details of where to get them from on the [selenium-webdriver](#) page (see the table in the first section.) Obviously, some of the browsers are OS-specific, but we're going to stick with Firefox and Chrome, as they are available across all the main OSes.

1. Download the latest [GeckoDriver](#) (for Firefox) and [ChromeDriver](#) drivers.
2. Unpack them into somewhere fairly easy to navigate to, like the root of your home user directory.
3. Add the chromedriver and geckodriver driver's location to your system PATH variable. This should be an absolute path from the root of your hard disk, to the directory containing the drivers. For example, if we were using a Mac OS X machine, our user name was bob, and we put our drivers in the root of our home folder, the path would be `/Users/bob`.

 **Note:** Just to reiterate, the path you add to PATH needs to be the path to the directory containing the drivers, not the paths to the drivers themselves! This is a common mistake.

To set your PATH variable on Mac OS X/most Linux systems:

1. Open your `.bash_profile` (or `.bashrc`) file (if you can't see hidden files, you'll need to display them, see [Show/Hide hidden files in Mac OS X](#) or [Show hidden folders in Ubuntu](#)).
2. Paste the following into the bottom of your file (updating the path as it actually is on your machine):

```
1 | #Add WebDriver browser drivers to PATH
2 |
3 | export PATH=$PATH:/Users/bob
```

3. Save and close this file, then restart your Terminal/command prompt to reapply your Bash configuration.
4. Check that your new paths are in the PATH variable by entering the following into your terminal:

```
1 | echo $PATH
```

5. You should see it printed out in the terminal.

To set your PATH variable on Windows, follow the instructions at [How can I add a new folder to my system path?](#)

OK, let's try a quick test to make sure everything is working.

1. Create a new file inside your project directory called `google_test.js`:
2. Give it the following contents, then save it:

```
1  var webdriver = require('selenium-webdriver'),
2      By = webdriver.By,
3      until = webdriver.until;
4
5  var driver = new webdriver.Builder()
6      .forBrowser('firefox')
7      .build();
8
9  driver.get('http://www.google.com');
10
11 driver.findElement(By.name('q')).sendKeys('webdriver');
12
13 driver.sleep(1000).then(function() {
14     driver.findElement(By.name('q')).sendKeys(webdriver.Key.TAB);
15 });
16
17 driver.findElement(By.name('btnK')).click();
18
19 driver.sleep(2000).then(function() {
20     driver.getTitle().then(function(title) {
21         if(title === 'webdriver - Google Search') {
22             console.log('Test passed');
23         } else {
24             console.log('Test failed');
25         }
26     });
27 });
28
29 driver.quit();
```

3. In terminal, make sure you are inside your project folder, then enter the following command:

```
1 | node google_test
```

You should see an instance of Firefox automatically open up! Google should automatically be loaded in a tab, "webdriver" should be entered in the search box, and the search button will be clicked. WebDriver will then wait for 2 seconds; the document title is then accessed, and if it is "webdriver - Google Search", we will return a message to claim the test is passed. WebDriver will then close down the Firefox instance and stop.

Testing in multiple browsers at once

There is also nothing to stop you running the test on multiple browsers simultaneously. Let's try this!

1. Create another new file inside your project directory called `google_test_multiple.js`. You can feel free to change the references to some of the other browsers we added, remove them, etc., depending on what browsers you have available to test on your operating system. You'll need to make sure you have the right browser drivers set up on your system. In terms of what string to use inside the `.forBrowser()` method for other browsers, see the [Browser enum](#) reference page.
2. Give it the following contents, then save it:

```
1 | var webdriver = require('selenium-webdriver'),
2 |     By = webdriver.By,
3 |     until = webdriver.until;
4 |
5 | var driver_fx = new webdriver.Builder()
6 |     .forBrowser('firefox')
7 |     .build();
8 |
9 | var driver_chr = new webdriver.Builder()
10 |    .forBrowser('chrome')
11 |    .build();
12 |
13 | searchTest(driver_fx);
14 | searchTest(driver_chr);
```

```
15
16 function searchTest(driver) {
17     driver.get('http://www.google.com');
18     driver.findElement(By.name('q')).sendKeys('webdriver');
19
20     driver.sleep(1000).then(function() {
21         driver.findElement(By.name('q')).sendKeys(webdriver.Key.TAB);
22     });
23
24     driver.findElement(By.name('btnK')).click();
25
26     driver.sleep(2000).then(function() {
27         driver.getTitle().then(function(title) {
28             if(title === 'webdriver - Google Search') {
29                 console.log('Test passed');
30             } else {
31                 console.log('Test failed');
32             }
33         });
34     });
35
36     driver.quit();
37 }
```

3. In terminal, make sure you are inside your project folder, then enter the following command:

```
1 | node google_test_multiple
```

4. If you are using a Mac and do decide to test Safari, you might get an error message along the lines of "Could not create a session: You must enable the 'Allow Remote Automation' option in Safari's Develop menu to control Safari via WebDriver." If you get this, follow the given instruction and try again.

So here we've done the test as before, except that this time we've wrapped it inside a function, `searchTest()`. We've created new browser instances for multiple browsers, then passed each one to the function so the test is performed on all three browsers!

Fun huh? Let's move on, look at the basics of WebDriver syntax, in a bit more detail.

WebDriver syntax crash course

Let's have a look at a few key features of the webdriver syntax. For more complete details, you should consult the [selenium-webdriver JavaScript API reference](#) for a detailed reference, and the Selenium main documentation's [Selenium WebDriver](#) and [WebDriver: Advanced Usage](#) pages, which contain multiple examples to learn from written in different languages.

Starting a new test

To start up a new test, you need to include the `selenium-webdriver` module like this:

```
1 | var webdriver = require('selenium-webdriver'),
2 |     By = webdriver.By,
3 |     until = webdriver.until;
```

Next, you need to create a new instance of a driver, using the new `webdriver.Builder()` constructor. This needs to have the `forBrowser()` method chained onto it to specify what browser you want to test with this builder, and the `build()` method to actually build it (see the [Builder class reference](#) for detailed information on these features).

```
1 | var driver = new webdriver.Builder()
2 |     .forBrowser('firefox')
3 |     .build();
```

Note that it is possible to set specific configuration options for browsers to be tested, for example you can set a specific version and OS to test in the `forBrowser()` method:

```
1 | var driver = new webdriver.Builder()
2 |     .forBrowser('firefox', '46', 'MAC')
3 |     .build();
```

You could also set these options using an environment variable, for example:

```
SELENIUM_BROWSER=firefox:46:MAC
```

Let's create a new test to allow us to explore this code as we talk about it. Inside your selenium test project directory, create a new file called `quick_test.js`, and add the

following code to it:

```
1 | var webdriver = require('selenium-webdriver'),
2 |   By = webdriver.By,
3 |   until = webdriver.until;
4 |
5 | var driver = new webdriver.Builder()
6 |   .forBrowser('firefox')
7 |   .build();
```

Getting the document you want to test

To load the page you actually want to test, you use the `get()` method of the driver instance you created earlier, for example:

```
1 | driver.get('http://www.google.com');
```

 **Note:** See the [WebDriver class reference](#) for details of the features in this section and the ones below it.

You can use any URL to point to your resource, including a `file://` URL to test a local document:

```
1 | driver.get('file:///Users/chrisfills/git/learning-area/tools-testing/cross
```

or

```
1 | driver.get('http://localhost:8888/fake-div-buttons.html');
```

But it is better to use a remote server location so the code is more flexible — when you start using a remote server to run your tests (see later on), your code will break if you try to use local paths.

Add this line to the bottom of `quick_test.js` now:

```
1 | driver.get('http://mdn.github.io/learning-area/tools-testing/cross-browser
```


Interacting with the document

Now we've got a document to test, we need to interact with it in some way, which usually involves first selecting a specific element to test something about. You can [select UI elements in many ways](#) in WebDriver, including by ID, class, element name, etc. The actual selection is done by the `findElement()` method, which accepts as a parameter a selection method. For example, to select an element by ID:

```
1 | var element = driver.findElement(By.id('myElementId'));
```

One of the most useful ways to find an element by CSS — the `By.css` method allows you to select an element using a CSS selector

Enter the following at the bottom of your `quick_test.js` code now:

```
1 | var button = driver.findElement(By.css('button:nth-of-type(1)'));
```

Testing your element

There are many ways to interact with your web documents and elements on them. You can see useful common examples starting at [Getting text values](#) on the WebDriver docs.

If we wanted to get the text inside our button, we could do this:

```
1 | button.getText().then(function(text) {  
2 |     console.log('Button text is \'' + text + '\');  
3 | });
```

Add this to `quick_test.js` now.

Making sure you are inside your project directory, try running the test:

```
1 | node quick_test.js
```

You should see the button's text label reported inside the console.

let's do something a bit more useful. delete the previous code entry, then add this line at the bottom instead:

```
1 | button.click();
```

Try running your test again; the button will be clicked, and the `alert()` popup should appear. At least we know the button is working!

You can interact with the popup too. Add the following to the bottom of the code, and try testing it again:

```
1 | var alert = driver.switchTo().alert();
2 |
3 | alert.getText().then(function(text) {
4 |     console.log('Alert text is \'' + text + '\');
5 | });
6 |
7 | alert.accept();
```

Next, let's try entering some text into one of the form elements. Add the following code and try running your test again:

```
1 | var input = driver.findElement(By.id('input1'));
2 | input.sendKeys('Filling in my form');
```

You can submit key presses that can't be represented by normal characters using properties of the `webdriver.Key` object. For example, above we used this construct to tab out of the form input before submitting it:

```
1 | driver.sleep(1000).then(function() {
2 |     driver.findElement(By.name('q')).sendKeys(webdriver.Key.TAB);
3 | });
```

Waiting for something to complete

There are times where you'll want to make `WebDriver` wait for something to complete before carrying on. For example if you load a new page, you'll want to wait for the page's DOM to finish loading before you try to interact with any of its elements, otherwise the test will likely fail.

In our `google_test.js` test for example, we included this block:


```
1 driver.sleep(2000).then(function() {
2   driver.getTitle().then(function(title) {
3     if(title === 'webdriver - Google Search') {
4       console.log('Test passed');
5     } else {
6       console.log('Test failed');
7     }
8   });
9 });
```

The `sleep()` method accepts a value that specifies the time to wait in milliseconds — the method returns a promise that resolves at the end of that time, at which point the code inside the `then()` executes. In this case we get the title of the current page with the `getTitle()` method, then return a pass or fail message depending on what its value is.

We could add a `sleep()` method to our `quick_test.js` test too — try wrapping your last line of code in a block like this:

```
1 driver.sleep(2000).then(function() {
2   input.sendKeys('Filling in my form');
3   input.getAttribute("value").then(function(value) {
4     if(value !== '') {
5       console.log('Form input editable');
6     }
7   });
8 });
```

WebDriver will now wait for 2 seconds before filling in the form field. We then test whether its value got filled in (i.e. is not empty) by using `getAttribute()` to retrieve its value attribute value, and print a message to the console if it is not empty.

 **Note:** There is also a method called `wait()`, which repeatedly tests a condition for a certain length of time, and then carries on executing the code. This also makes use of the `util` library, which defines common conditions to use along with `wait()`.

Shutting down drivers after use

After you've finished running a test, you should shut down any driver instances you've opened, to make sure that you don't end up with loads of rogue browser instances open on your machine! This is done using the `quit()` method. Simply call this on your driver instance when you are finished with it. Add this line to the bottom of your `quick_test.js` test now:

```
1 | driver.quit();
```

When you run it, you should now see the test execute and the browser instance shut down again after the test is complete. This is useful for not cluttering up your computer with loads of browser instances, especially if you have so many that it is causing the computer to slow down.

Test best practices

There has been a lot written about best practices for writing tests. You can find some good background information at [Test Design Considerations](#). In general, you should make sure that your tests are:

1. Using good locator strategies: When you are [Interacting with the document](#), make sure that you use locators and page objects that are unlikely to change — if you have a testable element that you want to perform a test on, make sure that it has a stable ID, or position on the page that can be selected using a CSS selector, which isn't going to just change with the next site iteration. You want to make your tests as non-brittle as possible, i.e. they won't just break when something changes.
2. Write atomic tests: Each test should test one thing only, making it easy to keep track of what test file is testing which criterion. As an example, the `google_test.js` test we looked at above is pretty good, as it just tests a single thing — whether the title of a search results page is set correctly. We could work on giving it a better name so it is easier to work out what it does if we add more google tests. Perhaps `results_page_title_set_correctly.js` would be slightly better?
3. Write autonomous tests: Each test should work on its own, and not depend on other tests to work.

In addition, we should mention test results/reporting — we've been reporting results in our above examples using simple `console.log()` statements, but this is all done in JavaScript, so you can use whatever test running and reporting system you want, be it [Mocha](#)/[Chai](#)/some other kind of combination.


1. For example, try making a local copy of our [mocha_test.js](#) example inside your project directory. Put it inside a subfolder called `test`. This example uses a long chain of promises to run all the steps required in our test — the promise-based methods WebDriver uses need to resolve for it to work properly.
2. Install the mocha test harness by running the following command inside your project directory:

```
1 | npm install --save-dev mocha
```

3. you can now run the test (and any others you put inside your `test` directory) using the following command:

```
1 | mocha --no-timeouts
```

4. You should include the `--no-timeouts` flag to make sure your tests don't end up failing because of Mocha's arbitrary timeout (which is 3 seconds).

 **Note:** [saucelabs-sample-test-frameworks](#) contains several useful examples showing how to set up different combinations of test/assertion tools.

Running remote tests

It turns out that running tests on remote servers isn't that much more difficult than running them locally. You just need to create your driver instance, but with a few more features specified, including the capabilities of the browser you want to test on, the address of the server, and the user credentials you need (if any) to access it.

BrowserStack

Getting Selenium tests to run remotely on BrowserStack is easy. The code you need should follow the pattern seen below.

Let's write an example:

1. Inside your project directory, create a new file called `bstack_google_test.js`.
2. Give it the following contents:

```
var webdriver = require('selenium-webdriver'),
    By = webdriver.By,
    until = webdriver.until;

// Input capabilities
var capabilities = {
  'browserName' : 'Firefox',
  'browser_version' : '56.0 beta',
  'os' : 'OS X',
  'os_version' : 'Sierra',
  'resolution' : '1280x1024',
  'browserstack.user' : 'YOUR-USER-NAME',
  'browserstack.key' : 'YOUR-ACCESS-KEY',
  'browserstack.debug' : 'true',
  'build' : 'First build'
};

var driver = new webdriver.Builder()
  .usingServer('http://hub-cloud.browserstack.com/wd/hub')
  .withCapabilities(capabilities)
  .build();

driver.get('http://www.google.com');
driver.findElement(By.name('q')).sendKeys('webdriver');

driver.sleep(1000).then(function() {
  driver.findElement(By.name('q')).sendKeys(webdriver.Key.TAB);
});

driver.findElement(By.name('btnK')).click();

driver.sleep(2000).then(function() {
  driver.getTitle().then(function(title) {
    if(title === 'webdriver - Google Search') {
      console.log('Test passed');
    } else {
      console.log('Test failed');
    }
  });
});

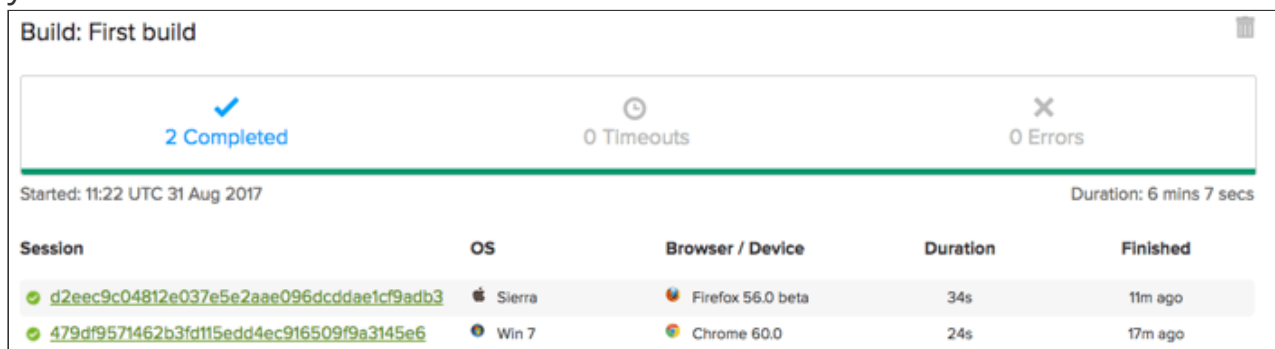
driver.quit();
```

3. From your [BrowserStack automation dashboard](#), get your user name and access key (see *Username and Access Keys*). Replace the YOUR-USER-NAME and YOUR-ACCESS-KEY placeholders in the code with your actual user name and access key values (and make sure you keep them secure).
4. Run your test with the following command:

```
1 | node bstack_google_test
```

The test will be sent to BrowserStack, and the test result will be returned to your console. This shows the importance of including some kind of result reporting mechanism!

5. Now if you go back to the [BrowserStack automation dashboard](#) page, you'll see your test listed:



The screenshot shows the BrowserStack automation dashboard for a build titled "Build: First build". At the top, there are three status indicators: "2 Completed" with a blue checkmark, "0 Timeouts" with a clock icon, and "0 Errors" with a red X icon. Below these, it says "Started: 11:22 UTC 31 Aug 2017" and "Duration: 6 mins 7 secs". A table lists the test sessions with columns for Session ID, OS, Browser / Device, Duration, and Finished time.

Session	OS	Browser / Device	Duration	Finished
d2eec9c04812e037e5e2aae096dcddae1cf9adb3	Sierra	Firefox 56.0 beta	34s	11m ago
479df9571462b3fd115edd4ec916509f9a3145e6	Win 7	Chrome 60.0	24s	17m ago

If you click on the link for your test, you'll get to a new screen where you will be able to see a video recording of the test, and multiple detailed logs of information pertaining to it.

Note: The *Resources* menu option on the Browserstack automation dashboard contains a wealth of useful information on using it to run automated tests. See [Node JS Documentation for writing automate test scripts in Node JS](#) for the node-specific information. Explore the docs to find out all the useful things BrowserStack can do.

Note: If you don't want to write out the capabilities objects for your tests by hand, you can generate them using the generators embedded in the docs. See [Run tests on mobile browsers](#) and [Run tests on desktop browsers](#).

Filling in BrowserStack test details programmatically

You can use the BrowserStack REST API and some other capabilities to annotate your test with more details, such as whether it passed, why it passed, what project the test is part of, etc. BrowserStack doesn't know these details by default!

Let's update our `bstack_google_test.js` demo, to show how these features work:

1. First, we'll need to import the node request module, so we can use it to send requests to the REST API. Add the following line at the very top of your code:

```
1 | var request = require("request");
```

2. Now we'll update our `capabilities` object to include a project name — add the following line before the closing curly brace, remembering to add a comma at the end of the previous line (you can vary the build and project names to organize the tests in different windows in the BrowserStack automation dashboard):

```
1 | 'project' : 'Google test 2'
```

3. Next we need to access the `sessionId` of the current session, so we know where to send the request (the ID is included in the request URL, as you'll see later). Include the following lines just below the block that creates the `driver` object (`var driver ...`):

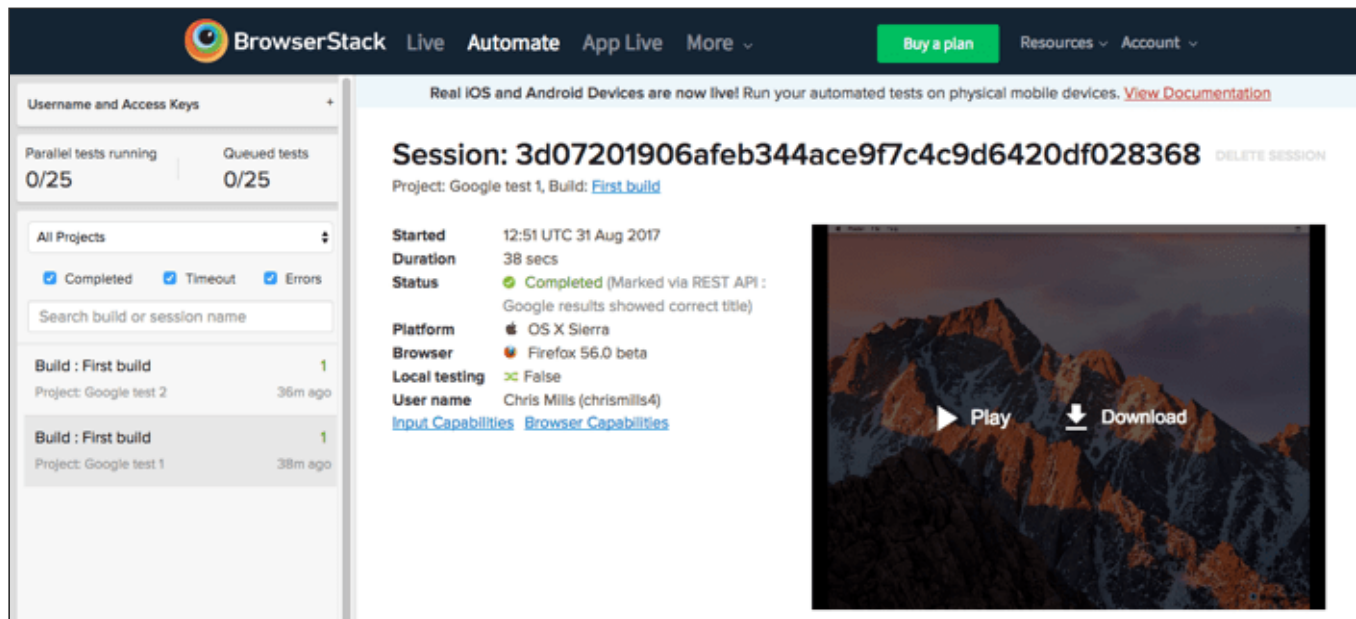
```
1 | var sessionId;  
2 |  
3 | driver.session_.then(function(sessionData) {  
4 |     sessionId = sessionData.id_;  
5 | });
```

4. Finally, update the `driver.sleep(2000) ...` block near the bottom of the code to add REST API calls (again, replace the `YOUR-USER-NAME` and `YOUR-ACCESS-KEY` placeholders in the code with your actual user name and access key values):

```
driver.sleep(2000).then(function() {  
    driver.getTitle().then(function(title) {  
        if(title === 'webdriver - Google Search') {  
            console.log('Test passed');  
            request({uri: "https://YOUR-USER-NAME:YOUR-ACCESS-KEY@www.browse  
        } else {  
            console.log('Test failed');  
            request({uri: "https://YOUR-USER-NAME:YOUR-ACCESS-KEY@www.browse  
        }  
    });  
});
```


These are fairly intuitive — once the test completes, we send an API call to BrowserStack to update the test with a passed or failed status, and a reason for the result.

If you now go back to your [BrowserStack](#) automation dashboard page, you should see your test session available, as before, but with the updated data attached to it:



Sauce Labs

Getting Selenium tests to run remotely on Sauce Labs is also very simple, and very similar to BrowserStack albeit with a few syntactic differences. The code you need should follow the pattern seen below.

Let's write an example:

1. Inside your project directory, create a new file called `sauce_google_test.js`.
2. Give it the following contents:

```
1  var webdriver = require('selenium-webdriver'),
2      By = webdriver.By,
3      until = webdriver.until,
4      username = "YOUR-USER-NAME",
5      accessKey = "YOUR-ACCESS-KEY";
6
7  var driver = new webdriver.Builder()
8      .withCapabilities({
9      'browserName': 'chrome',
10     'platform': 'Windows XP',
11     'version': '43.0',
12     'username': username,
```

```
13     'accessKey': accessKey
14   })
15   .usingServer("https://" + username + ":" + accessKey +
16     "@ondemand.saucelabs.com:443/wd/hub")
17   .build();
18
19   driver.get('http://www.google.com');
20
21   driver.findElement(By.name('q')).sendKeys('webdriver');
22
23   driver.sleep(1000).then(function() {
24     driver.findElement(By.name('q')).sendKeys(webdriver.Key.TAB);
25   });
26
27   driver.findElement(By.name('btnK')).click();
28
29   driver.sleep(2000).then(function() {
30     driver.getTitle().then(function(title) {
31       if(title === 'webdriver - Google Search') {
32         console.log('Test passed');
33       } else {
34         console.log('Test failed');
35       }
36     });
37   });
38
39   driver.quit();
```

3. From your [Sauce Labs user settings](#), get your user name and access key. Replace the YOUR-USER-NAME and YOUR-ACCESS-KEY placeholders in the code with your actual user name and access key values (and make sure you keep them secure).
4. Run your test with the following command:

```
1 | node sauce_google_test
```

The test will be sent to Sauce Labs, and the test result will be returned to your console. This shows the importance of including some kind of result reporting mechanism!

5. Now if you go to your [Sauce Labs Automated Test dashboard](#) page, you'll see your test listed; from here you'll be able to see videos, screenshots, and other such data.

Automated Builds	Automated Tests	Manual Tests
Monday, Oct 24th		
? Unnamed job 63754eb819124e6d80ae214c79ca4f8b started at 4:29PM by @chrisdavidmills		
Complete ran for 13s		

❏ **Note:** Sauce Labs' [Platform Configurator](#) is a useful tool for generating capability objects to feed to your driver instances, based on what browser/OS you want to test on.

❏ **Note:** for more useful details on testing with Sauce Labs and Selenium, check out [Getting Started with Selenium for Automated Website Testing](#), and [Instant Selenium Node.js Tests](#).

Filling in Sauce Labs test details programmatically

You can use the Sauce Labs API to annotate your test with more details, such as whether it passed, the name of the test, etc. Sauce Labs doesn't know these details by default!

To do this, you need to:

1. Install the Node Sauce Labs wrapper using the following command (if you've not already done it for this project):

```
npm install saucelabs --save-dev
```

2. Require saucelabs — put this at the top of your `sauce_google_test.js` file, just below the previous variable declarations:

```
1 | var SauceLabs = require('saucelabs');
```

3. Create a new instance of SauceLabs, by adding the following just below that:

```
1 | var saucelabs = new SauceLabs({  
2 |   username : "YOUR-USER-NAME",  
3 |   password : "YOUR-ACCESS-KEY"  
4 | });
```

Again, replace the `YOUR-USER-NAME` and `YOUR-ACCESS-KEY` placeholders in the code with your actual user name and access key values (note that the saucelabs npm

package rather confusingly uses `password`, not `accessKey`). Since you are using these twice now, you may want to create a couple of helper variables to store them in.

- Below the block where you define the `driver` variable (just below the `build()` line), add the following block — this gets the correct driver `sessionID` that we need to write data to the job (you can see it action in the next code block):

```
1 | driver.getSession().then(function (sessionid){
2 |     driver.sessionID = sessionid.id_;
3 | });
```

- Finally, replace the `driver.sleep(2000) ...` block near the bottom of the code with the following:

```
1 | driver.sleep(2000).then(function() {
2 |     driver.getTitle().then(function(title) {
3 |         if(title === 'webdriver - Google Search') {
4 |             console.log('Test passed');
5 |             var testPassed = true;
6 |         } else {
7 |             console.log('Test failed');
8 |             var testPassed = false;
9 |         }
10 |
11 |         saucelabs.updateJob(driver.sessionID, {
12 |             name: 'Google search results page title test',
13 |             passed: testPassed
14 |         });
15 |     });
16 | });
```

Here we've set a `testPassed` variable to `true` or `false` depending on whether the test passed or fails, then we've used the `saucelabs.updateJob()` method to update the details.

If you now go back to your [Sauce Labs Automated Test dashboard](#) page, you should see your new job now has the updated data attached to it:

Tuesday, Oct 25th

 **Google search results page title test**
started a minute ago by @chrisdavidmills

 XP

 43

Success
ran for 14s

Your own remote server

If you don't want to use a service like Sauce Labs or BrowserStack, you can always set up your own remote testing server. Let's look at how to do this.

1. The Selenium remote server requires Java to run. Download the latest JDK for your platform from the [Java SE downloads page](#). Install it when it is downloaded.
2. Next, download the latest [Selenium standalone server](#) — this acts as a proxy between your script and the browser drivers. Choose the latest stable version number (i.e. not a beta), and from the list choose a file starting with "selenium-server-standalone". When this has downloaded, put it in a sensible place, like in your home directory. If you've not already added the location to your PATH, do so now (see the [Setting up Selenium in Node](#) section).
3. Run the standalone server by entering the following into a terminal on your server computer

```
java -jar selenium-server-standalone-3.0.0.jar
```

(update the .jar filename) so it matches exactly what file you've got.

4. The server will run on <http://localhost:4444/wd/hub> — try going there now to see what you get.

Now we've got the server running, let's create a demo test that will run on the remote selenium server.

1. Create a copy of your `google_test.js` file, and call it `google_test_remote.js`; put it in your project directory.
2. Update the second code block (which starts with `var driver =`) like so

```
1 | var driver = new webdriver.Builder()  
2 |   .forBrowser('firefox')  
3 |   .usingServer('http://localhost:4444/wd/hub')  
4 |   .build();
```

3. Run your test, and you should see it run as expected; this time however you will be running it on the standalone server:

```
1 | node google_test_remote.js
```

So this is pretty cool. We have tested this locally, but you could set this up on just about any server along with the relevant browser drivers, and then connect your scripts to it

using the URL you choose to expose it at.

Integrating selenium with CI tools

As another point, it is also possible to integrate Selenium and related tools like Sauce Labs with continuous integration (CI) tools — this is useful, as it means you can run your tests via a CI tool, and only commit new changes to your code repository if the tests pass.

It is out of scope to look at this area in detail in this article, but we'd suggest getting started with Travis CI — this is probably the easiest CI tool to get started with, and has good integration with web tools like GitHub and Node.

To get started, see for example:

- [Travis CI for complete beginners](#)
 - [Building a Node.js project \(with Travis\)](#)
 - [Using Sauce Labs with Travis CI](#)
-

Summary

This module should have proven fun, and should have given you enough of an insight into writing and running automated tests for you to get going with writing your own automated tests.

[← Previous](#)[↑ Overview: Cross browser testing](#)

In this module

- Introduction to cross browser testing
 - Strategies for carrying out testing
 - Handling common HTML and CSS problems
 - Handling common JavaScript problems
 - Handling common accessibility problems
 - Implementing feature detection
 - Introduction to automated testing
 - Setting up your own test automation environment
-