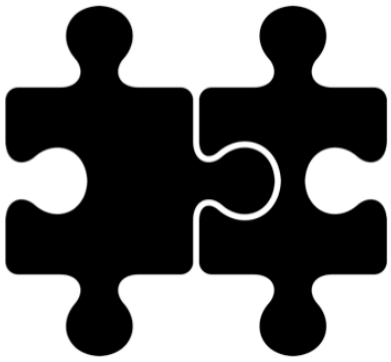
SELENIUM CHEAT SHEETS

JavaScript Edition



by Dave Haeffner

Version 1.1.0

Table of Contents

Chapter 1

Local Configuration

Firefox

1. Download the latest geckodriver binary from here
2. Add it to your path
3. Create an instance of Firefox

```
var webdriver = require('selenium-webdriver');
var vendorDirectory = process.cwd() + '/vendor';
process.env.PATH = vendorDirectory + ":$PATH";
driver = new webdriver.Builder().forBrowser('firefox').build();
```

To use the legacy FirefoxDriver:

1. Set an environment variable of `SELENIUM_MARIONETTE` to `false`
2. Create an instance of Firefox

```
var webdriver = require('selenium-webdriver');
process.env.SELENIUM_MARIONETTE=false
driver = new webdriver.Builder().forBrowser('firefox').build();
```

NOTE: For more information about FirefoxDriver check out the Selenium project Wiki page for FirefoxDriver and the Mozilla project page for geckodriver

Chrome

In order to use Chrome you need to download the ChromeDriver binary for your operating system from here (pick the highest number for the latest version). You either need to manually add it to your System Path or add to the System Path for the current terminal session.

```
process.env.PATH = '/path/to/chromedriver/folder' + ":$PATH";
var driver = new webdriver.Builder().forBrowser('chrome').build();
```

NOTE: For more information about ChromeDriver check out the Selenium project Wiki page for ChromeDriver

Internet Explorer

For Internet Explorer on Windows you need to download the IEDriverServer.exe from here (pick the highest number for the latest version) and either manually add it to your System Path or add to the System Path for the current terminal session.

```
var driver = new webdriver.Builder().forBrowser('ie').build();
```

NOTE: As of July 19, 2016 Internet Explorer 8 and older are no longer supported by the Selenium project. Also, if you're trying to run Windows 11 then you will need to add a registry key to your system. For more information about this and other InternetExplorerDriver details check out the Selenium project Wiki page for InternetExplorerDriver.

Edge

In order to use Microsoft Edge you need to have access to Windows 10. You can download a free virtual machine with it from Microsoft for testing purposes from Microsoft's Modern.IE developer portal. After that you need to download the appropriate `Microsoft WebDriver` server for your build of Windows. To find that go to `Start`, `Settings`, `System`, `About` and locate the number next to `OS Build` on the screen. Then it's just a simple matter of requesting a new instance of Edge.

```
var driver = new webdriver.Builder().forBrowser('edge').build();
```

NOTE: Currently Edge is only supported in the C#, Java, and JavaScript bindings. For more information about EdgeDriver check out the main page on the Microsoft Developer portal and the download page for the EdgeDriver binary.

Safari

To use Safari you need to download and install a SafariDriver browser extension for Selenium which you can get from this direct download link from the Selenium project.

```
var driver = new webdriver.Builder().forBrowser('safari').build();
```

NOTE: There is no Selenium support for Safari on Windows. For more information about SafariDriver check out the Selenium project Wiki page for SafariDriver

Opera

Versions 15 and new of Opera are built from the same rendering engine as Chrome. So if you run your tests with ChromeDriver then you are essentially testing Opera too.

There are some slight differences with it though. So if you have a business need to test with Opera, be sure to check out the OperaChromiumDriver for current versions of Opera and the OperaPrestoDriver for older versions of Opera.

Chapter 2
Cloud Configuration

Sauce Labs

Initial Setup

1. Create run-time flags with sensible defaults that can be overridden
2. Specify the browser and operating system you want through Desired Capabilitaies
3. Connect to Sauce Labs' end-point through Selenium Remote -- providing the Desired Capabilities
4. Store the WebDriver instance returned for use in your tests

```
// filename: lib/config.js
module.exports = {
  baseUrl: process.env.BASE_URL || 'http://the-internet.herokuapp.com',
  host: process.env.HOST || 'saucelabs',
  browser: process.env.BROWSER || 'internet explorer',
  browserVersion: process.env.BROWSER_VERSION || '11.0',
  platform: process.env.PLATFORM || 'Windows 7',
  sauceUsername: process.env.SAUCE_USERNAME,
  sauceAccessKey: process.env.SAUCE_ACCESS_KEY
};
```

```
// filename: lib/DriverFactory.js
var url = 'http://ondemand.saucelabs.com:80/wd/hub';
builder = new webdriver.Builder().usingServer(url);
builder.withCapabilities({
  browserName: config.browser,
  browserVersion: config.browserVersion,
  platform: config.platform,
  username: config.sauceUsername,
  accessKey: config.sauceAccessKey
});
var driver = builder.build();
```

For more info:

- Sauce Labs Available Platforms page
- Sauce Labs Automated Test Configurator

Setting the Test Name

1. Grab the test class and test method name dynamically after the test runs (in the `afterEach`)
2. Update the Sauce Labs job through the JavaScript executor

```
var testName = this.currentTest.fullTitle();
driver.executeScript('sauce:job-name=' + testName);
```

Setting the Job Status

1. Grab the test result from Mocha
2. Update the Sauce Labs job through the JavaScript executor

```
var testResult = (this.currentTest.state === 'passed') ? true : false;
driver.executeScript('sauce:job-result=' + testResult);
```

Chapter 3

Common Commands

Visit a page

```
driver.get('http://the-internet.herokuapp.com')
```

Find an element

Works using locators, which are covered in the next section.

```
// find just one, the first one Selenium finds
driver.findElement(locator);

// find all instances of the element on the page
// returns a collection
driver.findElements(locator);
```

Work with a found element

```
// chain commands together
driver.findElement(locator).click();

// store the element
// and then perform a command with it
var element = driver.findElement(locator);
element.click();
```

Perform an action

```
element.click();                 // clicks an element
element.submit();                // submits a form
element.clear();                 // clears an input field of its text
element.sendKeys('input text');  // types into an input field
```

Ask a question

Each of these returns a Boolean.

```
element.isDisplayed();    // is it visible?
element.isEnabled();      // can it be selected?
element.isSelected();     // is it selected?
```

Retrieve information

```
// by attribute name
element.getAttribute('href');

// directly from an element
element.getText();
```

For more info:

- the WebElement documentation for the Selenium JavaScript bindings

Chapter 4
Locators

Guiding principles

Good Locators are:

- unique
- descriptive
- unlikely to change

Be sure to:

1. Start with ID and Class
2. Use CSS selectors (or XPath) when you need to traverse
3. Talk with a developer on your team when the app is hard to automate
    1. tell them what you're trying to automate
    2. work with them to get more semantic markup added to the page

ID

```
driver.findElement({id: 'username'});
```

Class

```
driver.findElement({class: 'dues'});
```

CSS Selectors

```
driver.findElement({css: '#example'});
```

Approach	Locator	Description
ID	`#example`	`#` denotes an ID
Class	`.example`	`.` denotes a Class
Classes	`.flash.success`	use `.` in front of each class for multiple
Direct child	`div > a`	finds the element in the next child
Child/subschild	`div a`	finds the element in a child or child's child
Next sibling	`input.username + input`	finds the next adjacent element
Attribute values	`form input[name='username']`	a great alternative to id and class matches
Attribute values	`input[name='continue'][type='button']`	can chain multiple attribute filters together
Location	`li:nth-child(4)`	finds the 4th element only if it is an li
Location	`li:nth-of-type(4)`	finds the 4th li in a list
Location	`*:nth-child(4)`	finds the 4th element regardless of type
Sub-string	`a[id^='beginning_']`	finds a match that starts with (prefix)
Sub-string	`a[id$='_end']`	finds a match that ends with (suffix)
Sub-string	`a[id*='gooey_center']`	finds a match that contains (substring)
Inner text	`a:contains('Log Out')`	an alternative to substring matching

NOTE: Older browser (e.g., Internet Explorer 8) don't support CSS Pseudo-classes, so some of these locator approaches won't work (e.g., Location matches and Inner text matches).

For more info see one of the following resources:

- CSS Selectors Reference
- XPath Syntax Reference
- CSS & XPath Examples by Sauce Labs
- CSS vs. XPath Selenium benchmarks
- The difference between nth-child and nth-of-type
- How To Verify Your Locators
- CSS Selector Game

Chapter 5

Exception Handling

1. Use the Promise library that comes with the Selenium bindings
2. Defer the promise to create your own promise control flow
3. Perform the Selenium action and start a promise
4. Fulfill the promise if the action is successful
5. Fulfill the promise with `false` in the error callback if the error is `NoSuchElementError`
6. For all other errors reject the promise
7. Return the promise when done

```
Promise = require('selenium-webdriver').promise,
var defer = Promise.defer();
driver.findElement(locator).isDisplayed().then(function(isDisplayed) {
  defer.fulfill(isDisplayed);
}, function(error) {
  if (error.name === 'NoSuchElementError') {
    defer.fulfill(false);
  } else {
    defer.reject(error);
  }
});
return defer.promise;
```

For more info see:

- the Promise documentation for the Selenium JavaScript bindings

Chapter 6
Waiting

Implicit Wait

- Specify a timeout during test setup (in milliseconds)
- For every action that Selenium is unable to complete it will retry it until either:
  - the action can be accomplished, or
  - the amount of time specified has been reached and raise an exception (typically `NoSuchElementError`)
- Less flexible than explicit waits
- Not recommended

```
driver.manage().timeouts().implicitlyWait(15000);
```

Explicit Waits

- Recommended way to wait in your tests
- Specify an amount of time (in milliseconds) and an action
- Selenium will try the action repeatedly until either:
  - the action can be accomplished, or
  - the amount of time specified has been reached and raise an exception

```
var Until = require('selenium-webdriver').until;
driver.wait(Until.elementLocated(locator), 15000);
```

For more info:

- The case against mixing Implicit and Explicit Waits together
- Explicit vs Implicit Waits
- Selenium JavaScript bindings documentation on Wait conditions