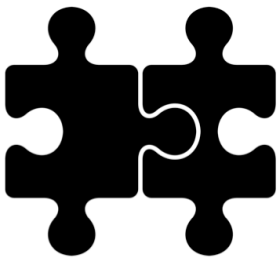
SELENIUM CHEAT SHEETS



by Dave Haeffner

Table of Contents

Local Configuration

Chrome

1. Download the latest ChromeDriver binary from here
2. Add it to your system path (or tell Selenium where to find it)
3. Create an instance of Chrome

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

protected WebDriver driver;
System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
driver = new ChromeDriver();
```

For more info see:

- the Selenium wiki page for ChromeDriver
- Google's ChromeDriver documentation

Firefox

Available out of the box.

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

protected WebDriver driver;
driver = new FirefoxDriver();
```

For more info see:

- the Selenium wiki page for FirefoxDriver

Internet Explorer

Only available on Microsoft Windows.

1. Download the latest IEDriverServer from here
2. Add the downloaded file location to your system path
3. Create an instance of Internet Explorer

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.ie.InternetExplorerDriver;

protected WebDriver driver;
driver = new InternetExplorerDriver();
```

For more info see:

- the Selenium wiki page for InternetExplorerDriver

Opera

Only works for version 12.16 or earlier. For newer versions of Opera, test using Chrome (since it uses the same back-end).

1. Download the latest Selenium Standalone Server from here
2. Create a Selenium specific environment variable pointing to the Selenium Standalone Server jar file
3. Create an instance of Opera

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.opera.OperaDriver;

protected WebDriver driver;
System.setProperty("SELENIUM_SERVER_JAR", "/path/to/selenium-server-standalone.jar");
driver = new OperaDriver();
```

For more info see:

- the old Selenium wiki page for OperaDriver

Safari

Available out of the box as of version 2.21 of Selenium.

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.safari.SafariDriver;

protected WebDriver driver;
driver = new SafariDriver();
```

For more info see:

- the Selenium wiki page for SafariDriver
- a write-up on how to use SafariDriver if you run into issues

Cloud Configuration

Sauce Labs

Initial Setup

1. Create field variables with sensible defaults that can be overridden at run-time
2. Specify the browser and operating system you want through Selenium's `DesiredCapabilities`
3. Create an instance of `RemoteWebDriver` using Sauce Labs' end-point -- providing your credentials and `DesiredCapabilities`
4. Store the instance in a field variable

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.RemoteWebDriver;

final String browser        = System.getProperty("browser", "firefox");
final String browserVersion = System.getProperty("browserVersion", "33");
final String platform       = System.getProperty("platform", "Windows XP");
final String sauceUser      = System.getenv("SAUCE_USERNAME");
final String sauceKey       = System.getenv("SAUCE_ACCESS_KEY");

DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setCapability("browserName", browser);
capabilities.setCapability("version", browserVersion);
capabilities.setCapability("platform", platform);

String sauceUrl = String.format("http://%s:%s@ondemand.saucelabs.com:80/wd/hub",
        sauceUser, sauceKey);

protected WebDriver driver;
driver = new RemoteWebDriver(new URL(sauceUrl), capabilities);
```

For more info see:

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

Setting the Test Name

1. Create a field variable to store the test name in
2. Add a Test Watcher Rule that uses the `starting()` method

3. Grab the display name of the test from within the Test Watcher and store it in the field variable
4. Pass the field variable value as a `"name"` capability in `DesiredCapabilities`

```
import org.junit.Rule;
import org.junit.rules.TestRule;
import org.junit.rules.TestWatcher;
import org.junit.runner.Description;

private String testName;

@Rule
public TestRule watcher = new TestWatcher() {
    protected void starting(Description description) {
        testName = description.getDisplayName();
    }
};

DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setCapability("browserName", browser);
capabilities.setCapability("version", browserVersion);
capabilities.setCapability("platform", platform);
capabilities.setCapability("name", testName);
```

For more info see:

- JUnit TestWatcher Rules documentation

Setting the Job Status

1. Install the `SauceREST` library
2. Create field variables to store `SauceREST` session and the Selenium session ID
3. Grab and store the Selenium session ID after a Sauce Labs instance is created
4. Add `failed()` and `succeeded()` Test Watcher methods
5. Create an instance of `SauceREST` to mark the Sauce job as passed or failed by using the Selenium session ID
6. BONUS POINTS: output the Sauce Labs job URL to the console when a test fails

```
<!-- filename: pom.xml -->
...
        <dependency>
            <groupId>com.saucelabs</groupId>
            <artifactId>saucerest</artifactId>
            <version>1.0.17</version>
            <scope>test</scope>
        </dependency>

    </dependencies>

    <repositories>
        <repository>
            <id>saucelabs-repository</id>
            <url>https://repository-saucelabs.forge.cloudbees.com/release</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </repository>
    </repositories>

</project>
```

```
// other import statements omitted for brevity
import com.saucelabs.saucerest.SauceREST;

protected WebDriver driver;
private String sessionId;
private SauceREST sauceClient;

driver = new RemoteWebDriver(new URL(sauceUrl), capabilities);
sessionId = ((RemoteWebDriver) driver).getSessionId().toString();
sauceClient = new SauceREST(sauceUser, sauceKey);

@Rule
public TestRule watcher = new TestWatcher() {

    @Override
    protected void failed(Throwable throwable, Description description) {
        if (host.equals("saucelabs")) {
            sauceClient.jobFailed(sessionId);
            System.out.println(String.format("https://saucelabs.com/tests/%s",
sessionId));
        }
    }

    @Override
    protected void succeeded(Description description) {
        if (host.equals("saucelabs")) {
            sauceClient.jobPassed(sessionId);
        }
    }

};
```

Common Actions

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
driver.findElements(locator);
// returns a collection
```

Work with a found element

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

// store the element
WebElement element = driver.findElement(locator);
element.click();
```

Perform an action

```
element.click();                  // clicks an element
element.submit();                 // submits a form
element.clear();                  // clearing an input field of its text
element.sendKeys("input text");   // typing 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

Each of these returns a String.

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

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

For more info see:

- the Selenium WebElement API Documentation

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(By.id("username"));
```

Class

```
driver.findElement(By.className("dues"));
```

CSS Selectors

```
driver.findElement(By.cssSelector("#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:

- CSS vs. XPath benchmarks
- CSS & XPath Examples by Sauce Labs
- CSS Selector Game
- The difference between nth-child and nth-of-type
- w3schools CSS Selectors Reference
- w3schools XPath Syntax Reference
- How To Verify Your Locators

Exception Handling

1. Try the action you want
2. Catch the relevant exception and return `false` instead

```
try {
    return driver.findElement(locator).isDisplayed();
} catch (org.openqa.selenium.NoSuchElementException exception) {
    return false;
}
```

For more info see:

- the Selenium WebDriverException API Documentation

Waiting

Implicit Wait

- Only needs to be configured once
- Tells Selenium to wait for a specified amount of time before raising a `NoSuchElementException`
- Less flexible than explicit waits

```
driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
```

Explicit Waits

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

```
wait = Selenium::WebDriver::Wait.new(timeout: seconds)
wait.until { driver.find_element(locator).displayed? }
```

For more info see:

- The case against using Implicit and Explicit Waits together
- Explicit vs. Implicit Waits