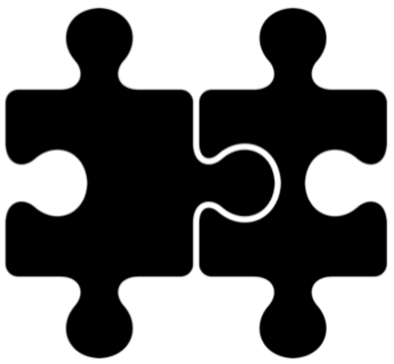
SELENIUM CHEAT SHEETS

Ruby Edition



by Dave Haeffner

Version 4.0.0

Table of Contents

Chapter 1
Local Configuration

Firefox

Option 1

1. Download the latest geckodriver binary
2. Add its location to your path
3. Create an instance

```
service = Selenium::WebDriver::Service.firefox(path: driver_path)
@driver = Selenium::WebDriver.for :firefox, service: service
```

Option 2

1. Download the latest geckodriver binary
2. Add its location to a system property in your setup code
3. Create an instance

```
@driver = Selenium::WebDriver.for :firefox
```

NOTE: For more information about geckodriver check out its project page

Chrome

Option 1

1. Download the latest ChromeDriver binary
2. Add its location to your path
3. Create an instance

Option 2

1. Download the latest ChromeDriver binary
2. Add its location to a system property in your setup code
3. Create an instance

```
service = Selenium::WebDriver::Service.chrome(path: driver_path)
@driver = Selenium::WebDriver.for :chrome, service: service
```

NOTE: For more information about ChromeDriver check out its project page

Internet Explorer

Option 1

1. Download the latest IEDriverServer
2. Add its location to your path
3. Create an instance

```
@driver = Selenium::WebDriver.for :internet_explorer
```

Option 2

1. Download the latest IEDriverServer
2. Add its location to a system property in your setup code
3. Create an instance

```
service = Selenium::WebDriver::Service.internet_explorer(path: driver_path)
@driver = Selenium::WebDriver.for :internet_explorer, service: service
```

NOTE: There is additional setup required to make Internet Explorer work with Selenium. For more information check out the Selenium project Wiki page for InternetExplorerDriver.

Edge

In order to use Microsoft Edge you need to have access to Windows 10.

Option 1

1. Download EdgeDriver
2. Add its location to your path
3. Create an instance

```
@driver = Selenium::WebDriver.for :edge
```

Option 2

1. Download EdgeDriver
2. Add its location to a system property in your setup code
3. Create an instance

```
service = Selenium::WebDriver::Service.edge(path: driver_path)
@driver = Selenium::WebDriver.for :edge, service: service
```

NOTE: You can download a free virtual machine with Windows 10 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.

Safari

For Safari, you'll need SafariDriver, which ships with the latest version of Safari.

You just need to enable it from the command-line.

```
> safaridriver --enable
```

```
@driver = Selenium::WebDriver.for :safari
```

NOTE: For additional details, or information on setup requirements for older versions of macOS, see the SafariDriver documentation from Apple.

Chapter 2
Cloud Configuration

Sauce Labs

Initial Setup

1. Store your Sauce Labs Username and Access Key in environment variables
2. Specify the browser and operating system you want through Selenium's Capabilities
3. Create an instance of Selenium using the Sauce Labs end-point, passing in the Capabilities

```
caps = Selenium::WebDriver::Remote::Capabilities.send(config[:browser_name])
caps[:browser_version] = config[:browser_version]
caps[:platform_name] = config[:platform_name]
url = "http://#{ENV['SAUCE_USERNAME]}:#{ENV['SAUCE_ACCESS_KEY']]}
@ondemand.saucelabs.com:80/wd/hub"
@driver = Selenium::WebDriver.for(
  :remote,
  url: url,
  desired_capabilities: caps)
```

For more info:

- Sauce Labs Available Platforms page

Setting the Job Status

1. Install the `sauce_whisk` gem
2. Add `require 'sauce_whisk'` to your test harness configuration
3. Use `sauce_whisk` to mark the Sauce job as passed or failed by using Selenium's `session_id`

```ruby
# an RSpec example
require 'sauce_whisk'

after(:each) do |example|
  if example.exception.nil?
    SauceWhisk::Jobs.pass_job @driver.session_id
  else
    SauceWhisk::Jobs.fail_job @driver.session_id
  end
  @driver.quit
end
```

Using Sauce Connect for Private Apps

1. Download [Sauce Connect]
2. Start the Sauce Connect tunnel (e.g., `bin/sc -u YOUR_USERNAME -k YOUR_ACCESS_KEY`)
3. Run your tests

Chapter 3
Common Actions

Visit a page

```
driver.get 'url'
# e.g., driver.get 'http://the-internet.herokuapp.com'
```

Find an element

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

```
# just one, the first Selenium finds
driver.find_element(locator)

# all instances of the element on the page
driver.find_elements(locator)
# returns an Array
```

Work with a found element

```
# Chain actions together
driver.find_element(locator).click

# Store the element
element = driver.find_element(locator)
element.click
```

Work with a collection of found elements

```
collection = driver.find_elements(locator)

# by name
collection.first.click
collection.last.click

# by index
collection[0].click
collection[-1].click
```

Perform an action

```
element.click       # click on an element
element.clear       # clearing an input field of its text
element.send_keys   # typing into an input field
```

Ask a question

```
element.displayed? # is it visible?
element.enabled?   # can it be selected?
element.selected?  # is it selected?
```

Retrieve information

```
# by attribute name
element.attribute('href')

# directly from an element
element.text
```

For more info:

- Selenium's Element API documentation
- Selenium's `attribute` Element API documentation

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.find_element(id: 'username')
```

Class

```
driver.find_elements(class: 'dues')
```

CSS Selectors

```
# Example usage
driver.find_element(css: '#username')
driver.find_element(css: '.dues')
```

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

For more info:

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

Chapter 5
Exception Handling

1. Rescue the relevant exceptions, returning `false` for each
2. Place in a helper method for easy reuse

```
def is_displayed?(locator)
  begin
    driver.find_element(locator).displayed?
  rescue Selenium::WebDriver::Error::NoSuchElementError
    false
  rescue Selenium::WebDriver::Error::StaleElementReferenceError
    false
  end
end

is_displayed? locator
# Returns false if the element is not displayed.
# Otherwise, returns true.
```

For more info:

- Selenium's list of exceptions

Chapter 6
Waiting

Implicit Wait

- Specify a timeout in milliseconds (typically during test setup)
- For every command 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.implicit_wait = 5
```

Explicit Waits

- Specify a timeout (in milliseconds) and an expected condition to wait for
- Selenium will check for the expected condition repeatedly until either:
  - is is successful, or
  - the amount of time specified has been reached and raise an exception
- Recommended way to wait in your tests

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

For more info:

- Explicit vs Implicit Waits
- Selenium Ruby bindings documentation for explicit waits