Getting Started

1 Sample code 3
2 Features 5
3 Getting started 7
4 Basic browsing and interactions 9
5 JavaScript support 11
6 Walking on… 13
7 Drivers 15
   7.1 Browser based drivers 15
   7.2 Headless drivers 15
   7.3 Remote driver 15
8 Get in touch and contribute 17
   8.1 Why use Splinter? 17
   8.2 Install guide 18
   8.3 Splinter Tutorial 18
   8.4 Browser 20
   8.5 Finding elements 22
   8.6 Mouse interactions 23
   8.7 Interacting with elements in the page 25
   8.8 Matchers 27
   8.9 Cookies manipulation 28
   8.10 Take screenshot 28
   8.11 Take element screenshot 29
   8.12 Take html snapshot 29
   8.13 Executing javascript 29
   8.14 Chrome WebDriver 30
   8.15 Firefox WebDriver 36
   8.16 Remote WebDriver 42
   8.17 zope.testbrowser 43
   8.18 django client 44
   8.19 Flask client 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.20  Dealing with HTTP status code and exceptions</td>
<td>44</td>
</tr>
<tr>
<td>8.21  Using HTTP Proxies</td>
<td>45</td>
</tr>
<tr>
<td>8.22  Frames, alerts and prompts</td>
<td>46</td>
</tr>
<tr>
<td>8.23  API Documentation</td>
<td>46</td>
</tr>
<tr>
<td>8.24  Selenium Keys</td>
<td>54</td>
</tr>
<tr>
<td>8.25  Community</td>
<td>55</td>
</tr>
<tr>
<td>8.26  Contribute</td>
<td>55</td>
</tr>
<tr>
<td>8.27  Writing new splinter drivers</td>
<td>56</td>
</tr>
<tr>
<td>8.28  Setting up your splinter development environment</td>
<td>57</td>
</tr>
</tbody>
</table>

**Python Module Index**

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
</table>

**Index**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python Module Index</td>
<td>59</td>
</tr>
<tr>
<td>Index</td>
<td>61</td>
</tr>
</tbody>
</table>
Splinter is an open source tool for testing web applications using Python. It lets you automate browser actions, such as visiting URLs and interacting with their items.
Sample code

```python
from splinter import Browser

with Browser() as browser:
    # Visit URL
    url = "http://www.google.com"
    browser.visit(url)
    browser.fill('q', 'splinter - python acceptance testing for web applications')
    # Find and click the 'search' button
    button = browser.find_by_name('btnG')
    # Interact with elements
    button.click()
    if browser.is_text_present('splinter.readthedocs.io'):
        print("Yes, the official website was found!")
    else:
        print("No, it wasn't found... We need to improve our SEO techniques")
```

**Note:** if you don’t provide any driver to the `Browser` function, `firefox` will be used.
CHAPTER 2

Features

• simple api
• multiple webdrivers (chrome, firefox, zopetestbrowser, remote webdriver, Django, Flask)
• css and xpath selectors
• support for iframes and alerts
• can execute javascript
• works with ajax and async javascript

what’s new in splinter?
CHAPTER 3

Getting started

• Why use Splinter
• Installation
• Quick tutorial
Basic browsing and interactions

- Browser and navigation
- Finding elements
- Mouse interactions
- Interacting with elements and forms
- Verify the presence of texts and elements in a page, with matchers
- Cookies manipulation
- Take screenshot
CHAPTER 5

JavaScript support

• Executing JavaScript
CHAPTER 6

Walking on...

- Dealing with HTTP status code and exceptions
- Using HTTP proxies
- Interacting with iframes, alerts and prompts
- Full API documentation
CHAPTER 7

Drivers

7.1 Browser based drivers

The following drivers open a browser to run your actions:

- *Chrome WebDriver*
- *Firefox WebDriver*

7.2 Headless drivers

The following drivers don’t open a browser to run your actions (but each has its own dependencies, check the specific docs for each driver):

- *Chrome WebDriver (headless option)*
- *Firefox WebDriver (headless option)*
- *zope.testbrowser*
- *django client*
- *flask client*

7.3 Remote driver

The remote driver uses Selenium Remote to control a web browser on a remote machine.

- *Remote WebDriver*
8.1 Why use Splinter?

Splinter is an abstraction layer on top of existing browser automation tools such as Selenium and zope.testbrowser. It has a high-level API that makes it easy to write automated tests of web applications.

For example, to fill out a form field with Splinter:

```python
browser.fill('username', 'janedoe')
```

In Selenium, the equivalent code would be:

```python
elem = browser.find_element.by_name('username')
elem.send_keys('janedoe')
```

Because Splinter is an abstraction layer, it supports multiple web automation backends. With Splinter, you can use the same test code to do browser-based testing with Selenium as the backend and “headless” testing (no GUI) with zope.testbrowser as the backend.

Splinter has drivers for browser-based testing on:

- Chrome
- Firefox
- Browsers on remote machines

For headless testing, Splinter has drivers for:
8.2 Install guide

8.2.1 Install Python

In order to install Splinter, make sure Python is installed. Note: only Python 2.7+ is supported. Download Python from http://www.python.org. If you’re using Linux or Mac OS X, it is probably already installed.

8.2.2 Install splinter

Basically, there are two ways to install Splinter:

Install a stable release

If you’re interested on an official and almost bug-free version, just run from the Terminal:

```bash
$ [sudo] pip install splinter
```

Install under-development source-code

Otherwise, if you want Splinter’s latest-and-greatest features and aren’t afraid of running under development code, run:

```bash
$ git clone git://github.com/cobrateam/splinter.git
$ cd splinter
$ [sudo] python setup.py install
```

Notes:

• make sure you have already set up your development environment.
• in this second case, make sure Git is installed.
• in order to use Chrome webdriver, you need to set up Google Chrome properly.

8.3 Splinter Tutorial

Before starting, make sure Splinter is installed

This tutorial provides a simple example, teaching step by step how to:

• search for splinter – python acceptance testing for web applications' in google.com, and
• find if splinter official website is listed among the search results
8.3.1 Create a Browser instance

First of all, import `Browser` class and instantiate it.

```python
from splinter import Browser
browser = Browser()
```

**Note:** if you don’t provide any driver argument to the `Browser` function, `firefox` will be used (Browser function documentation).

8.3.2 Visit Google website

Visit any website using the `browser.visit` method. Let’s go to Google search page:

```python
browser.visit('http://google.com')
```

8.3.3 Input search text

After a page is loaded, you can perform actions, such as clicking, filling text input, checking radio and checkbox. Let’s fill Google’s search field with `splinter - python acceptance testing for web applications`:

```python
browser.fill('q', 'splinter - python acceptance testing for web applications')
```

8.3.4 Press the search button

Tell Splinter which button should be pressed. A button - or any other element - can be identified using its css, xpath, id, tag or name.

In order to find Google’s search button, do:

```python
button = browser.find_by_name('btnG')
```

Note that this `btnG` was found looking at Google’s page source code.

With the button in hands, we can then press it:

```python
button.click()
```

Note: Both steps presented above could be joined in a single line, such as:

```python
browser.find_by_name('btnG').click()
```

8.3.5 Find out that Splinter official website is in the search results

After pressing the button, you can check if Splinter official website is among the search responses. This can be done like this:

```python
if browser.is_text_present('splinter.readthedocs.io'):
    print "Yes, found it! :)
else:
    print "No, didn't find it :(" 
```

In this case, we are just printing something. You might use assertions, if you’re writing tests.
8.3.6 Close the browser

When you’ve finished testing, close your browser using `browser.quit`:

```
browser.quit()
```

8.3.7 All together

Finally, the source code will be:

```
from splinter import Browser

browser = Browser() # defaults to firefox
browser.visit('http://google.com')
browser.fill('q', 'splinter - python acceptance testing for web applications')
browser.find_by_name('btnG').click()

if browser.is_text_present('splinter.readthedocs.io'):
    print "Yes, the official website was found!"
else:
    print "No, it wasn't found... We need to improve our SEO techniques"

browser.quit()
```

8.4 Browser

To use splinter you need to create a Browser instance:

```
from splinter import Browser
browser = Browser()
```

Or, you can use it by a context manager, through the `with` statement:

```
from splinter import Browser
with Browser() as b:
    # stuff using the browser
```

This last example will create a new browser window and close it when the cursor reaches the code outside the `with` statement, automatically.

Splinter supports the following drivers:

- Chrome
- Firefox
- Browsers on remote machines
- zope.testbrowser
- Django client
- Flask client

The following examples create new Browser instances for specific drivers:

```
browser = Browser('chrome')
browser = Browser('firefox')
browser = Browser('zope.testbrowser')
```

8.4.1 Navigating with Browser.visit

You can use the `visit` method to navigate to other pages:
browser.visit('http://cobrateam.info')

The `visit` method takes only a single parameter - the `url` to be visited.

You can visit a site protected with basic HTTP authentication by providing the username and password in the `url`.

```python
browser.visit('http://username:password@cobrateam.info/protected')
```

### 8.4.2 Managing Windows

You can manage multiple windows (such as popups) through the `windows` object:

```python
browser.windows # all open windows
browser.windows[0] # the first window
browser.windows[window_name] # the window_name window
browser.windows.current # the current window
browser.windows.current = browser.windows[3] # set current window to window 3
```

```python
window = browser.windows[0]
window.is_current # boolean - whether window is current active window
window.is_current = True # set this window to be current window
window.next # the next window
window.prev # the previous window
window.close() # close this window
window.close_others() # close all windows except this one
```

This window management interface is not compatible with the undocumented interface exposed in v0.6.0 and earlier.

### 8.4.3 Reload a page

You can reload a page using the `reload` method:

```python
browser.reload()
```

### 8.4.4 Navigate through the history

You can move back and forward through your browsing history using the `back` and `forward` methods:

```python
browser.visit('http://cobrateam.info')
browser.visit('https://splinter.readthedocs.io')
browser.back()
browser.forward()
```

### 8.4.5 Browser.title

You can get the title of the visited page using the `title` attribute:

```python
browser.title
```
8.4.6 Verifying page content with Browser.html

You can use the html attribute to get the html content of the visited page:

\[
\text{browser.html}
\]

8.4.7 Verifying page url with Browser.url

The visited page’s url can be accessed by the url attribute:

\[
\text{browser.url}
\]

8.4.8 Changing Browser User-Agent

You can pass a User-Agent header on Browser instantiation.

\[
b = \text{Browser(user_agent=\"Mozilla/5.0 (iPhone; U; CPU like Mac OS X; en)\")}
\]

8.5 Finding elements

Splinter provides 6 methods for finding elements in the page, one for each selector type: css, xpath, tag, name, id, value, text. Examples:

\[
\text{browser.find_by_css('h1')}\\
\text{browser.find_by_xpath('h1')}\\
\text{browser.find_by_tag('h1')}\\
\text{browser.find_by_name('name')}\\
\text{browser.find_by_text('Hello World!')}\\
\text{browser.find_by_id('firstheader')}\\
\text{browser.find_by_value('query')}
\]

Each of these methods returns a list with the found elements. You can get the first found element with the first shortcut:

\[
\text{first_found = browser.find_by_name('name').first}
\]

There’s also the last shortcut – obviously, it returns the last found element:

\[
\text{last_found = browser.find_by_name('name').last}
\]

8.5.1 Get element using index

You also can use an index to get the desired element in the list of found elements:

\[
\text{second_found = browser.find_by_name('name')[1]}
\]

8.5.2 All elements and find_by_id

A web page should have only one id, so the find_by_id method returns always a list with just one element.
8.5.3 Finding links

If you want to target only links on a page, you can use the methods provided in the links namespace. This is available at both the browser and element level.

Examples:

```python
links_found = browser.links.find_by_text('Link for Example.com')
links_found = browser.links.find_by_partial_text('for Example')
links_found = browser.links.find_by_href('http://example.com')
links_found = browser.links.find_by_partial_href('example')

links_found = browser.find_by_css('.main').links.find_by_text('Link for Example.com')
links_found = browser.find_by_css('.main').links.find_by_partial_text('for Example.com')
links_found = browser.find_by_css('.main').links.find_by_href('http://example.com')
links_found = browser.find_by_css('.main').links.find_by_partial_href('example')
```

As the other find_* methods, these returns a list of all found elements.

8.5.4 Chaining find of elements

Finding methods are chainable, so you can find the descendants of a previously found element.

```python
divs = browser.find_by_tag("div")
within_elements = divs.first.find_by_name("name")
```

8.5.5 ElementDoesNotExist exception

If an element is not found, the find_* methods return an empty list. But if you try to access an element in this list, the method will raise the splinter.exceptions.ElementDoesNotExist exception.

8.6 Mouse interactions

Note: Most mouse interaction currently works only on Chrome driver and Firefox 27.0.1.

Splinter provides some methods for mouse interactions with elements in the page. This feature is useful to test if an element appears on mouse over and disappears on mouse out (e.g.: subitems of a menu).

It’s also possible to send a click, double click or right click to the element.

Here is a simple example: imagine you have this jQuery event for mouse over and out:

```javascript
$('.menu-links').mouseover(function(){
    $(this).find('.subitem').show();
});
$('.menu-links').mouseout(function(){
    $(this).find('.subitem').hide();
});
```

You can use Splinter to fire the event programatically:
The methods available for mouse interactions are:

**8.6.1 mouse_over**

Puts the mouse above the element. Example:

```ruby
browser.find_by_tag('h1').mouse_over()
```

**8.6.2 mouse_out**

Puts the mouse out of the element. Example:

```ruby
browser.find_by_tag('h1').mouse_out()
```

**8.6.3 click**

Clicks on the element. Example:

```ruby
browser.find_by_tag('h1').click()
```

**8.6.4 double_click**

Double-clicks on the element. Example:

```ruby
browser.find_by_tag('h1').double_click()
```

**8.6.5 right_click**

Right-clicks on the element. Example:

```ruby
browser.find_by_tag('h1').right_click()
```

**8.6.6 drag_and_drop**

Yes, you can drag an element and drop it to another element! The example below drags the `<h1>...</h1>` element and drop it to a container element (identified by a CSS class).

```ruby
draggable = browser.find_by_tag('h1')
target = browser.find_by_css('.container')
draggable.drag_and_drop(target)
```
8.7 Interacting with elements in the page

8.7.1 Get value of an element

In order to retrieve an element’s value, use the value property:

```python
browser.find_by_css('h1').first.value
```

or

```python
element = browser.find_by_css('h1').first
element.value
```

8.7.2 Clicking links

You can click in links. To click in links by href, partial href, text or partial text you can use this. IMPORTANT: These methods return the first element always.

```python
browser.click_link_by_href('http://www.the_site.com/my_link')
```

or

```python
browser.click_link_by_partial_href('my_link')
```

or

```python
browser.click_link_by_text('my link')
```

or

```python
browser.click_link_by_partial_text('part of link text')
```

or

```python
browser.click_link_by_id('link_id')
```

8.7.3 Clicking buttons

You can click in buttons. Splinter follows any redirects, and submits forms associated with buttons.

```python
browser.find_by_name('send').first.click()
```

or

```python
browser.find_link_by_text('my link').first.click()
```

8.7.4 Interacting with forms
To trigger JavaScript events, like KeyDown or KeyUp, you can use the `type` method.

```python
browser.type('type', 'typing text')
```

If you pass the argument `slowly=True` to the `type` method you can interact with the page on every key pressed. Useful for testing field’s autocompletion (the browser will wait until next iteration to type the subsequent key).

```python
for key in browser.type('type', 'typing slowly', slowly=True):
    pass # make some assertion here with the key object :
```

You can also use `type` and `fill` methods in an element:

```python
browser.find_by_name('name').type('Steve Jobs', slowly=True)
browser.find_by_css('.city').fill('San Francisco')
```

### 8.7.5 Verifying if element is visible or invisible

To check if an element is visible or invisible, use the `visible` property. For instance:

```python
browser.find_by_css('h1').first.visible
```

will be True if the element is visible, or False if it is invisible.

### 8.7.6 Verifying if element has a className

To check if an element has a className, use the `has_class` method. For instance:

```python
browser.find_by_css('.content').first.has_class('content')
```

### 8.7.7 Interacting with elements through a ElementList object

Don’t you like to always use `first` when selecting an element for clicking, for example:

```python
browser.find_by_css('a.my-website').first.click()
```

You can invoke any `Element` method on `ElementList` and it will be proxied to the `first` element of the list. So the two lines below are equivalent:

```python
assert browser.find_by_css('a.banner').first.visible
assert browser.find_by_css('a.banner').visible
```
8.8 Matchers

When working with AJAX and asynchronous JavaScript, it’s common to have elements which are not present in the HTML code (they are created with JavaScript, dynamically). In this case you can use the methods `is_element_present` and `is_text_present` to check the existence of an element or text – Splinter will load the HTML and JavaScript in the browser and the check will be performed before processing JavaScript.

There is also the optional argument `wait_time` (given in seconds) – it’s a timeout: if the verification method gets `True` it will return the result (even if the `wait_time` is not over), if it doesn’t get `True`, the method will wait until the `wait_time` is over (so it’ll return the result).

8.8.1 Checking the presence of text

The method `is_text_present` is responsible for checking if a text is present in the page content. It returns `True` or `False`.

```python
browser = Browser()
browser.visit('https://splinter.readthedocs.io/')
browser.is_text_present('splinter')  # True
browser.is_text_present('splinter', wait_time=10)  # True, using wait_time
browser.is_text_present('text not present')  # False
```

There’s also a method to check if the text `is not` present: `is_text_not_present`. It works the same way but returns `True` if the text is not present.

```python
browser.is_text_not_present('text not present')  # True
browser.is_text_not_present('text not present', wait_time=10)  # True, using wait_time
browser.is_text_not_present('splinter')  # False
```

8.8.2 Checking the presence of elements

Splinter provides 6 methods to check the presence of elements in the page, one for each selector type: `css, xpath, tag, name, id, value, text`. Examples:

```python
browser.is_element_present_by_css('h1')
browser.is_element_present_by_xpath('//h1')
browser.is_element_present_by_tag('h1')
browser.is_element_present_by_name('name')
browser.is_element_not_present_by_text('Hello World!')
browser.is_element_not_present_by_id('firstheader')
browser.is_element_not_present_by_value('query', wait_time=10)  # using wait_time
```

As expected, these methods returns `True` if the element is present and `False` if it is not present.

There’s also the negative forms of these methods, as in `is_text_present`:

```python
browser.is_element_not_present_by_css('h6')
browser.is_element_not_present_by_xpath('//h6')
browser.is_element_not_present_by_tag('h6')
browser.is_element_not_present_by_name('unexisting-name')
browser.is_element_not_present_by_text('Not here :(')
browser.is_element_not_present_by_id('unexisting-header')
browser.is_element_not_present_by_id('unexisting-header', wait_time=10)  # using wait_—time
```
8.8.3 Checking the visibility of elements

There are two methods to check if the element is visible or hidden in the current page using either the selector type `css` or `xpath`. It returns `True` if the element is visible and `False` if the element in not visible.

```python
browser.is_element_visible_by_css('h5')
browser.is_element_visible_by_css('h5', wait_time=10)
browser.is_element_visible_by_xpath('//h5')
```

8.9 Cookies manipulation

It is possible to manipulate cookies using the `cookies` attribute from a `Browser` instance. The `cookies` attribute is an instance of a `CookieManager` class that manipulates cookies, like adding and deleting them.

8.9.1 Create cookie

To add a cookie use the add method:

```python
browser.cookies.add({'whatever': 'and ever'})
```

8.9.2 Retrieve all cookies

To retrieve all cookies use the `all` method:

```python
browser.cookies.all()
```

8.9.3 Delete a cookie

You can delete one or more cookies with the `delete` method:

```python
browser.cookies.delete('mwahahahaha')  # deletes the cookie 'mwahahahaha'
browser.cookies.delete('whatever', 'wherever')  # deletes two cookies
```

8.9.4 Delete all cookies

You can also delete all cookies: just call the `delete` method without any parameters:

```python
browser.cookies.delete()  # deletes all cookies
```

For more details check the API reference of the `CookieManager` class.

8.10 Take screenshot

Splinter can take current view screenshot easily:

```python
browser = Browser()
screenshot_path = browser.screenshot('absolute_path/your_screenshot.png')
```
You should use the absolute path to save screenshot. If you don’t use an absolute path, the screenshot will be saved in a temporary file.

Take a full view screenshot:

```python
browser = Browser()
screenshot_path = browser.screenshot('absolute_path/your_screenshot.png', full=True)
```

### 8.11 Take element screenshot

First, if you want to use this function, you should install the Pillow dependency:

```bash
pip install Pillow
```

If the element in the current view:

```python
browser = Browser()
browser.visit('http://example.com')
screenshot_path = browser.find_by_xpath('xpath_rule').first.screenshot('absolute_path/...your_screenshot.png')
```

If the element not in the current view, you should do it like this:

```python
browser = Browser()
browser.visit('http://example.com')
screenshot_path = browser.find_by_xpath('xpath_rule').first.screenshot('absolute_path/...your_screenshot.png', full=True)
```

### 8.12 Take html snapshot

Splinter can also take a snapshot of the current HTML:

```python
browser = Browser()
screenshot_path = browser.html_snapshot('absolute_path/your_screenshot.html')
```

### 8.13 Executing javascript

You can easily execute JavaScript, in drivers which support it:

```python
browser.execute_script("$('body').empty()")
```

You can return the result of the script:

```python
browser.evaluate_script("4+4") == 8
```

### 8.13.1 Example: manipulating text fields with JavaScript

Some text input actions cannot be “typed” thru `browser.fill()`, like new lines and tab characters. Below is an example how to work around this using `browser.execute_script()`. This is also much faster than `browser.fill()` as there is no simulated key typing delay, making it suitable for longer texts.
def fast_fill_by_javascript(browser: DriverAPI, elem_id: str, text: str):
    """Fill text field with copy-paste, not by typing key by key.
    Otherwise you cannot type enter or tab.
    :param id: CSS id of the textarea element to fill
    """
    text = text.replace("\t", "\\t")
    text = text.replace("\n", "\\n")
    # Construct a JavaScript snippet that is executed on the browser side
    snippet = f"""document.querySelector("#{elem_id}").value = "{text}";"""
    browser.execute_script(snippet)

8.14 Chrome WebDriver

Chrome WebDriver is provided by Selenium2. To use it, you need to install Selenium2 via pip:

```bash
[sudo] pip install selenium
```

It’s important to note that you also need to have Google Chrome installed in your machine.

Chrome can also be used from a custom path. To do this pass the executable path as a dictionary to the **kwargs argument. The dictionary should be set up with executable_path as the key and the value set to the path to the executable file.

```python
from splinter import Browser
executable_path = {'executable_path': '/path/to/chrome'}

browser = Browser('chrome', **executable_path)
```

8.14.1 Setting up Chrome WebDriver

In order to use **Google Chrome** with Splinter, since we’re using Selenium 2.3.x, you need to setup Chrome webdriver properly.

8.14.2 Mac OS X

The recommended way is by using Homebrew:

```bash
$ brew cask install chromedriver
```

8.14.3 Linux

Go to the download page on the Chromium project and choose the correct version for your Linux installation. Then extract the downloaded file in a directory in the PATH (e.g. /usr/bin). You can also extract it to any directory and add that directory to the PATH:
Linux 64bits

```bash
$ cd $HOME/Downloads
$ wget https://chromedriver.storage.googleapis.com/2.41/chromedriver_linux64.zip
$ unzip chromedriver_linux64.zip
$ mkdir -p $HOME/bin
$ mv chromedriver $HOME/bin
$ echo "export PATH=$PATH:$HOME/bin" >> $HOME/.bash_profile
```

8.14.4 Windows

**Note:** We don’t provide official support for Windows, but you can try it by yourself.

All you need to do is go to download page on Selenium project and choose “ChromeDriver server for win”. Your browser will download a zip file, extract it and add the `.exe` file to your PATH.

If you don’t know how to add an executable to the PATH on Windows, check these link out:

- Environment variables
- How to manage environment variables in Windows XP
- How to manage environment variables in Windows 8 & 10

8.14.5 Using Chrome WebDriver

To use the Chrome driver, all you need to do is pass the string `chrome` when you create the `Browser` instance:

```python
from splinter import Browser
browser = Browser('chrome')
```

**Note:** if you don’t provide any driver to the `Browser` function, `firefox` will be used.

**Note:** if you have trouble with `$HOME/.bash_profile`, you can try `$HOME/.bashrc`.

8.14.6 Using headless option for Chrome

Starting with Chrome 59, we can run Chrome as a headless browser. Make sure you read google developers updates

```python
from splinter import Browser
browser = Browser('chrome', headless=True)
```

8.14.7 Using incognito option for Chrome

We can run Chrome as a incognito browser.

```python
from splinter import Browser
browser = Browser('chrome', incognito=True)
```
8.14.8 Using emulation mode in Chrome

Chrome options can be passed to customize Chrome’s behaviour; it is then possible to leverage the experimental emulation mode.

```python
from selenium import webdriver
from splinter import Browser

mobile_emulation = {"deviceName": "Google Nexus 5"}
chrome_options = webdriver.ChromeOptions()
chrome_options.add_experimental_option("mobileEmulation",
    mobile_emulation)
browser = Browser('chrome', options=chrome_options)
```

Refer to the Chrome driver documentation.

8.14.9 API docs

```python
class splinter.driver.webdriver.chrome.WebDriver(options=None, user_agent=None,
    wait_time=2, fullscreen=False, incognito=False, headless=False,**kwargs)

attach_file(name, value)
    Fill the field identified by name with the content specified by value.

back()
    Back to the last URL in the browsing history.
    If there is no URL to back, this method does nothing.

check(name)
    Checks a checkbox by its name.
    Example:

    >>> browser.check("agree-with-terms")

    If you call browser.check n times, the checkbox keeps checked, it never get unchecked.
    To uncheck a checkbox, take a look in the uncheck method.

choose(name, value)
    Chooses a value in a radio buttons group.
    Suppose you have the two radio buttons in a page, with the name gender and values ‘F’ and ‘M’. If you use the choose method the following way:

    >>> browser.choose('gender', 'F')

    Then you’re choosing the female gender.

click_link_by_href(href)
    Clicks in a link by its href attribute.

click_link_by_id(id)
    Clicks in a link by id.

click_link_by_partial_href(partial_href)
    Clicks in a link by looking for partial content of href attribute.
```
click_link_by_partial_text (partial_text)
Clicks in a link by partial content of its text.

click_link_by_text (text)
Clicks in a link by its text.

cookies
A CookieManager instance.
For more details, check the cookies manipulation section.

evaluate_script (script, *args)
Similar to execute_script method.
Executes javascript in the browser and returns the value of the expression.
e.g.: ::
   >>> assert 4 == browser.evaluate_script('2 + 2')

execute_script (script, *args)
Executes a given JavaScript in the browser.
e.g.: ::
   >>> browser.execute_script('document.getElementById("body").innerHTML = "<p>Hello world!</p>"')

fill (name, value)
Fill the field identified by name with the content specified by value.

fill_form (field_values, form_id=None, name=None)
Fill the fields identified by name with the content specified by value in a dict.
Currently, fill_form supports the following fields: text, password, textarea, checkbox, radio and select.
Checkboxes should be specified as a boolean in the dict.

find_by (finder, selector, original_find=None, original_query=None, wait_time=None)
Wrapper for finding elements.
Must be attached to a class.

Returns: ElementList

find_by_css (css_selector, wait_time=None)
Returns an instance of ElementList, using a CSS selector to query the current page content.

find_by_id (id, wait_time=None)
Finds an element in current page by its id.
Even when only one element is find, this method returns an instance of ElementList

find_by_name (name, wait_time=None)
Finds elements in current page by their name.
Returns an instance of ElementList.

find_by_tag (tag, wait_time=None)
Find all elements of a given tag in current page.
Returns an instance of ElementList.
find_by_text(\text{text}=\text{None}, \text{wait\_time}=\text{None})
Finds elements in current page by their text.

Returns an instance of \text{ElementList}

find_by_value(\text{value}, \text{wait\_time}=\text{None})
Finds elements in current page by their value.

Returns an instance of \text{ElementList}

find_by_xpath(\text{xpath}, \text{original\_find}=\text{None}, \text{original\_query}=\text{None}, \text{wait\_time}=\text{None})
Returns an instance of \text{ElementList}, using a xpath selector to query the current page content.

find_link_by_href(\text{href})
Find all elements of a given tag in current page.

Returns an instance of \text{ElementList}

find_link_by_partial_href(\text{partial\_href})
Find links by looking for a partial str in their href attribute.

Returns an instance of \text{ElementList}

find_link_by_partial_text(\text{partial\_text})
Find links by looking for a partial str in their text.

Returns an instance of \text{ElementList}

find_link_by_text(\text{text})
Find links querying for their text.

Returns an instance of \text{ElementList}

find_option_by_text(\text{text})
Finds <option> elements by their text.

Returns an instance of \text{ElementList}

find_option_by_value(\text{value})
Finds <option> elements by their value.

Returns an instance of \text{ElementList}

forward()
Forward to the next URL in the browsing history.

If there is no URL to forward, this method does nothing.

get_alert(\text{wait\_time}=\text{None})
Changes the context for working with alerts and prompts.

For more details, check the \text{docs about iframes, alerts and prompts}

get_iframe(**\text{kwds})
Changes the context for working with iframes.

For more details, check the \text{docs about iframes, alerts and prompts}

html
Source of current page.

html_snapshot(\text{name}=\text{}, \text{suffix}='.\text{html}', \text{encoding}=\text{'utf-8'})
Write the current html to a file.

is_element_not_present_by_css(\text{css\_selector}, \text{wait\_time}=\text{None})
Verify if the element is not present in the current page by css, and wait the specified time in wait\_time.
Returns True if the element is not present and False if is present.

**is_element_not_present_by_id**(id, wait_time=None)
Verify if the element is present in the current page by id, and wait the specified time in wait_time.
Returns True if the element is not present and False if is present.

**is_element_not_present_by_name**(name, wait_time=None)
Verify if the element is not present in the current page by name, and wait the specified time in wait_time.
Returns True if the element is not present and False if is present.

**is_element_not_present_by_tag**(tag, wait_time=None)
Verify if the element is not present in the current page by tag, and wait the specified time in wait_time.
Returns True if the element is not present and False if is present.

**is_element_not_present_by_text**(text, wait_time=None)
Verify if the element is not present in the current page by text, and wait the specified time in wait_time.
Returns True if the element is not present and False if is present.

**is_element_not_present_by_value**(value, wait_time=None)
Verify if the element is not present in the current page by value, and wait the specified time in wait_time.
Returns True if the element is not present and False if is present.

**is_element_not_present_by_xpath**(xpath, wait_time=None)
Verify if the element is not present in the current page by xpath, and wait the specified time in wait_time.
Returns True if the element is not present and False if is present.

**is_element_present_by_css**(css_selector, wait_time=None)
Verify if the element is present in the current page by css, and wait the specified time in wait_time.
Returns True if the element is present and False if is not present.

**is_element_present_by_id**(id, wait_time=None)
Verify if the element is present in the current page by id, and wait the specified time in wait_time.
Returns True if the element is present and False if is not present.

**is_element_present_by_name**(name, wait_time=None)
Verify if the element is present in the current page by name, and wait the specified time in wait_time.
Returns True if the element is present and False if is not present.

**is_element_present_by_tag**(tag, wait_time=None)
Verify if the element is present in the current page by tag, and wait the specified time in wait_time.
Returns True if the element is present and False if is not present.

**is_element_present_by_text**(text, wait_time=None)
Verify if the element is present in the current page by text, and wait the specified time in wait_time.
Returns True if the element is present and False if is not present.

**is_element_present_by_value**(value, wait_time=None)
Verify if the element is present in the current page by value, and wait the specified time in wait_time.
Returns True if the element is present and False if is not present.

**is_element_present_by_xpath**(xpath, wait_time=None)
Verify if the element is present in the current page by xpath, and wait the specified time in wait_time.
Returns True if the element is present and False if is not present.
**is_text_present**(text, wait_time=None)

Searchs for text in the browser and wait the seconds specified in wait_time.

Returns True if finds a match for the text and False if not.

**quit()**

Quits the browser, closing its windows (if it has one).

After quit the browser, you can’t use it anymore.

**reload()**

Revisits the current URL.

**screenshot**(name='', suffix='.png', full=False)

Takes a screenshot of the current page and saves it locally.

**select**(name, value)

Selects an `<option>` element in an `<select>` element using the name of the `<select>` and the value of the `<option>`.

Example:

```
>>> browser.select("state", "NY")
```

**title**

Title of current page.

**type**(name, value, slowly=False)

Types the value in the field identified by name.

It’s useful to test javascript events like keyPress, keyUp, keyDown, etc.

If slowly is True, this function returns an iterator which will type one character per iteration.

**uncheck**(name)

Unchecks a checkbox by its name.

Example:

```
>>> browser.uncheck("send-me-emails")
```

If you call browser.uncheck n times, the checkbox keeps unchecked, it never get checked.

To check a checkbox, take a look in the check method.

**url**

URL of current page.

**visit**(url)

Visits a given URL.

The url parameter is a string.

### 8.15 Firefox WebDriver

Firefox WebDriver is provided by Selenium 2.0. To use it, you need to install Selenium 2.0 via pip:

```
$ [sudo] pip install selenium
```

It’s important to note that you also need to have Firefox and geckodriver installed in your machine and available on PATH environment variable. Once you have it installed, there is nothing you have to do, just use it :)
8.15.1 Using Firefox WebDriver

To use the Firefox driver, all you need to do is pass the string `firefox` when you create the `Browser` instance:

```python
from splinter import Browser
browser = Browser('firefox')
```

**Note:** if you don’t provide any driver to `Browser` function, `firefox` will be used.

8.15.2 Using headless option for Firefox

Starting with Firefox 55, we can run Firefox as a headless browser in Linux.

```python
from splinter import Browser
browser = Browser('firefox', headless=True)
```

8.15.3 Using incognito option for Firefox

We can run Firefox as a private browser.

```python
from splinter import Browser
browser = Browser('firefox', incognito=True)
```

8.15.4 How to use a specific profile for Firefox

You can specify a Firefox profile for using `Browser` function using the `profile` keyword (passing the name of the profile as a `str` instance):

```python
from splinter import Browser
browser = Browser('firefox', profile='my_profile')
```

If you don’t specify a profile, a new temporary profile will be created (and deleted when you close the browser).

8.15.5 How to use specific extensions for Firefox

An extension for firefox is a .xpi archive. To use an extension in Firefox webdriver profile you need to give the path of the extension, using the extensions keyword (passing the extensions as a `list` instance):

```python
from splinter import Browser
browser = Browser('firefox', extensions=['extension1.xpi', 'extension2.xpi'])
```

If you give an extension, after you close the browser, the extension will be deleted from the profile, even if is not a temporary one.

8.15.6 How to use selenium capabilities for Firefox

```python
from splinter import Browser
browser = Browser('firefox', capabilities={'acceptSslCerts': True})
```

You can pass any selenium read-write `DesiredCapabilities` parameters for Firefox.
8.15.7 API docs

class splinter.driver.webdriver.firefox.WebDriver

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attach_file(name, value)</td>
<td>Fill the field identified by name with the content specified by value.</td>
</tr>
<tr>
<td>back()</td>
<td>Back to the last URL in the browsing history.</td>
</tr>
<tr>
<td>check(name)</td>
<td>Checks a checkbox by its name.</td>
</tr>
<tr>
<td>choose(name, value)</td>
<td>Chooses a value in a radio buttons group.</td>
</tr>
<tr>
<td>click_link_by_href(href)</td>
<td>Clicks in a link by its href attribute.</td>
</tr>
<tr>
<td>click_link_by_id(id)</td>
<td>Clicks in a link by id.</td>
</tr>
<tr>
<td>click_link_by_partial_href(partial_href)</td>
<td>Clicks in a link by looking for partial content of href attribute.</td>
</tr>
<tr>
<td>click_link_by_partial_text(partial_text)</td>
<td>Clicks in a link by partial content of its text.</td>
</tr>
<tr>
<td>click_link_by_text(text)</td>
<td>Clicks in a link by its text.</td>
</tr>
<tr>
<td>cookies</td>
<td>A CookieManager instance.</td>
</tr>
<tr>
<td>evaluate_script(script, *args)</td>
<td>Similar to execute_script method. Execute javascript in the browser and returns the value of the expression.</td>
</tr>
</tbody>
</table>
e.g.: ::

```python
>>> assert 4 == browser.evaluate_script('2 + 2')
```

**execute_script**(script, *args)
Executes a given JavaScript in the browser.

e.g.: ::

```python
>>> browser.execute_script('document.getElementById("body").innerHTML = "<p>Hello world!</p>"')
```

**fill**(name, value)
Fill the field identified by name with the content specified by value.

**fill_form**(field_values, form_id=None, name=None)
Fill the fields identified by name with the content specified by value in a dict.
Currently, fill_form supports the following fields: text, password, textarea, checkbox, radio and select.
Checkboxes should be specified as a boolean in the dict.

**find_by**(finder, selector, original_find=None, original_query=None, wait_time=None)
Wrapper for finding elements.
Must be attached to a class.

Returns: ElementList

**find_by_css**(css_selector, wait_time=None)
Returns an instance of ElementList, using a CSS selector to query the current page content.

**find_by_id**(id, wait_time=None)
Finds an element in current page by its id.
Even when only one element is find, this method returns an instance of ElementList

**find_by_name**(name, wait_time=None)
Finds elements in current page by their name.
Returns an instance of ElementList.

**find_by_tag**(tag, wait_time=None)
Find all elements of a given tag in current page.
Returns an instance of ElementList

**find_by_text**(text=None, wait_time=None)
Finds elements in current page by their text.
Returns an instance of ElementList

**find_by_value**(value, wait_time=None)
Finds elements in current page by their value.
Returns an instance of ElementList

**find_by_xpath**(xpath, original_find=None, original_query=None, wait_time=None)
Returns an instance of ElementList, using a xpath selector to query the current page content.

**find_link_by_href**(href)
Find all elements of a given tag in current page.
Returns an instance of ElementList
**find_link_by_partial_href**(*partial_href*)
Find links by looking for a partial str in their href attribute.
Returns an instance of `ElementList`

**find_link_by_partial_text**(*partial_text*)
Find links by looking for a partial str in their text.
Returns an instance of `ElementList`

**find_link_by_text**(*text*)
Find links querying for their text.
Returns an instance of `ElementList`

**find_option_by_text**(*text*)
Finds `<option>` elements by their text.
Returns an instance of `ElementList`

**find_option_by_value**(*value*)
Finds `<option>` elements by their value.
Returns an instance of `ElementList`

**forward**()
Forward to the next URL in the browsing history.
If there is no URL to forward, this method does nothing.

**get_alert**(*wait_time=None*)
Changes the context for working with alerts and prompts.
For more details, check the docs about iframes, alerts and prompts

**get_iframe**(**kwds**)  
Changes the context for working with iframes.
For more details, check the docs about iframes, alerts and prompts

**html**
Source of current page.

**html_snapshot**(*name=", suffix=’.html’, encoding=’utf-8’*)
Write the current html to a file.

**is_element_not_present_by_css**(*css_selector, wait_time=None*)
Verify if the element is not present in the current page by css, and wait the specified time in `wait_time`.
Returns True if the element is not present and False if is present.

**is_element_not_present_by_id**(*id, wait_time=None*)
Verify if the element is present in the current page by id, and wait the specified time in `wait_time`.
Returns True if the element is not present and False if is present.

**is_element_not_present_by_name**(*name, wait_time=None*)
Verify if the element is not present in the current page by name, and wait the specified time in `wait_time`.
Returns True if the element is not present and False if is present.

**is_element_not_present_by_tag**(*tag, wait_time=None*)
Verify if the element is not present in the current page by tag, and wait the specified time in `wait_time`.
Returns True if the element is not present and False if is present.
**is_element_not_present_by_text** *(text, wait_time=None)*
Verify if the element is not present in the current page by text, and wait the specified time in *wait_time*.
Returns True if the element is not present and False if is present.

**is_element_not_present_by_value** *(value, wait_time=None)*
Verify if the element is not present in the current page by value, and wait the specified time in *wait_time*.
Returns True if the element is not present and False if is present.

**is_element_not_present_by_xpath** *(xpath, wait_time=None)*
Verify if the element is not present in the current page by xpath, and wait the specified time in *wait_time*.
Returns True if the element is not present and False if is present.

**is_element_present_by_css** *(css_selector, wait_time=None)*
Verify if the element is present in the current page by css, and wait the specified time in *wait_time*.
Returns True if the element is present and False if is not present.

**is_element_present_by_id** *(id, wait_time=None)*
Verify if the element is present in the current page by id, and wait the specified time in *wait_time*.
Returns True if the element is present and False if is not present.

**is_element_present_by_name** *(name, wait_time=None)*
Verify if the element is present in the current page by name, and wait the specified time in *wait_time*.
Returns True if the element is present and False if is not present.

**is_element_present_by_tag** *(tag, wait_time=None)*
Verify if the element is present in the current page by tag, and wait the specified time in *wait_time*.
Returns True if the element is present and False if is not present.

**is_element_present_by_text** *(text, wait_time=None)*
Verify if the element is present in the current page by text, and wait the specified time in *wait_time*.
Returns True if the element is present and False if is not present.

**is_element_present_by_value** *(value, wait_time=None)*
Verify if the element is present in the current page by value, and wait the specified time in *wait_time*.
Returns True if the element is present and False if is not present.

**is_element_present_by_xpath** *(xpath, wait_time=None)*
Verify if the element is present in the current page by xpath, and wait the specified time in *wait_time*.
Returns True if the element is present and False if is not present.

**is_text_present** *(text, wait_time=None)*
Searchs for *text* in the browser and wait the seconds specified in *wait_time*.
Returns True if finds a match for the *text* and False if not.

**quit** ()
Quits the browser, closing its windows (if it has one).
After quit the browser, you can’t use it anymore.

**reload** ()
Revisits the current URL.

**screenshot** *(name=", suffix='.png', full=False)*
Takes a screenshot of the current page and saves it locally.

---

8.15. Firefox WebDriver
select \((name, value)\)
Selects an \(<option>\) element in an \(<select>\) element using the \(name\) of the \(<select>\) and the \(value\) of the \(<option>\).

Example:

```python
>>> browser.select("state", "NY")
```

title
Title of current page.

type \((name, value, slowly=False)\)
Types the \(value\) in the field identified by \(name\).

It’s useful to test javascript events like keyPress, keyUp, keyDown, etc.

If \(slowly\) is True, this function returns an iterator which will type one character per iteration.

uncheck \((name)\)
Unchecks a checkbox by its name.

Example:

```python
>>> browser.uncheck("send-me-emails")
```

If you call browser.uncheck \(n\) times, the checkbox keeps unchecked, it never get checked.

To check a checkbox, take a look in the check method.

url
URL of current page.

visit \((url)\)
Visits a given URL.

The \(url\) parameter is a string.

---

### 8.16 Remote WebDriver

Remote WebDriver is provided by Selenium. To use it, you need to install Selenium via pip:

```bash
$ [sudo] pip install selenium
```

#### 8.16.1 Setting up the Remote WebDriver

To use Remote WebDriver, you need to have access to a Selenium remote WebDriver server. Setting up one of these servers is beyond the scope of this document. However, some companies provide access to a Selenium Grid as a service.

#### 8.16.2 Using the Remote WebDriver

To use the Remote WebDriver, use \texttt{driver_name=\"remote\"} when you create the \texttt{Browser} instance.

The \texttt{browser_name} argument should then be used to specify the web browser. The other arguments match Selenium’s Remote WebDriver arguments.
Desired Capabilities will be set automatically based on Selenium’s defaults. These can be expanded and/or replaced by providing your own.

The following example uses Sauce Labs (a company that provides Selenium Remote WebDriver servers as a service) to request an Internet Explorer 9 browser instance running on Windows 7.

```python
# Specify the server URL
remote_server_url = 'http://YOUR_SAUCE_USERNAME:YOUR_SAUCE_ACCESS_KEY@ondemand.saucelabs.com:80/wd/hub'

with Browser(
    driver_name="remote",
    browser='internetexplorer',
    command_executor=remote_server_url,
    desired_capabilities = {
        'platform': 'Windows 7',
        'version': '9',
        'name': 'Test of IE 9 on WINDOWS',
    },
    keep_alive=True,
) as browser:
    print("Link to job: https://saucelabs.com/jobs/{}").format(
        browser.driver.session_id)
    browser.visit("https://splinter.readthedocs.io")
    browser.find_by_text('documentation').first.click()
```

### 8.17 zope.testbrowser

To use the `zope.testbrowser` driver, you need to install `zope.testbrowser`, `lxml` and `cssselect`. You can install all of them in one step by running:

```bash
$ pip install splinter[zope.testbrowser]
```

#### 8.17.1 Using zope.testbrowser

To use the `zope.testbrowser` driver, all you need to do is pass the string `zope.testbrowser` when you create the `Browser` instance:

```python
from splinter import Browser
browser = Browser('zope.testbrowser')
```

By default `zope.testbrowser` respects any robots.txt preventing access to a lot of sites. If you want to circumvent this you can call

```python
browser = Browser('zope.testbrowser', ignore_robots=True)
```

**Note:** if you don’t provide any driver to `Browser` function, `firefox` will be used.
To use the django driver, you need to install django, lxml and cssselect. You can install all of them in one step by running:

```bash
$ pip install splinter[django]
```

### 8.18.1 Using django client

To use the django driver, all you need to do is pass the string `django` when you create the `Browser` instance:

```python
from splinter import Browser
browser = Browser('django')
```

**Note:** if you don’t provide any driver to `Browser` function, firefox will be used.

### 8.18.2 API docs

### 8.19 Flask client

To use the flask driver, you need to install Flask, lxml and cssselect. You can install all of them in one step by running:

```bash
$ pip install splinter[flask]
```

### 8.19.1 Using Flask client

To use the flask driver, you’ll need to pass the string `flask` and an app instances via the `app` keyword argument when you create the `Browser` instance:

```python
from splinter import Browser
browser = Browser('flask', app=app)
```

**Note:** if you don’t provide any driver to `Browser` function, firefox will be used.

When visiting pages with the Flask client, you only need to provide a path rather than a full URL. For example:

```python
browser.visit('/my-path')
```

### 8.19.2 API docs

### 8.20 Dealing with HTTP status code and exceptions

**Note:** After 0.8 version the webdriver (firefox, chrome) based drivers does not support http error handling.
8.20.1 Dealing with HTTP status code

It’s also possible to check which HTTP status code a browser.visit gets. You can use `status_code.is_success` to do the work for you or you can compare the status code directly:

```python
browser.visit('http://cobrateam.info')
browser.status_code.is_success() # True
# or
browser.status_code == 200 # True
# or
browser.status_code.code # 200
```

The difference between those methods is that if you get a redirect (or something that is not an HTTP error), `status_code.is_success` will consider your response as successfully. The numeric status code can be accessed via `status_code.code`.

8.20.2 Handling HTTP exceptions

Whenever you use the `visit` method, Splinter will check if the response is success or not, and if not, it will raise an `HttpResponseError` exception. But don’t worry, you can easily catch it:

```python
try:
    browser.visit('http://cobrateam.info/i-want-cookies')
except HttpResponseError, e:
    print "Oops, I failed with the status code %s and reason %s" % (e.status_code, e.reason)
```

*Note: status_code and this HTTP exception handling is available only for selenium webdriver*

8.21 Using HTTP Proxies

Unauthenticated proxies are simple, you need only configure the browser with the hostname and port.

Authenticated proxies are rather more complicated, (see RFC2617)

8.21.1 Using an unauthenticated HTTP proxy with Firefox

```python
profile = {
    'network.proxy.http': YOUR_PROXY_SERVER_HOST,
    'network.proxy.http_port': YOUR_PROXY_SERVER_PORT,
    'network.proxy.ssl': YOUR_PROXY_SERVER_HOST,
    'network.proxy.ssl_port': YOUR_PROXY_SERVER_PORT,
    'network.proxy.type': 1
}
self.browser = Browser(self.browser_type, profile_preferences=profile)
```

8.21.2 Authenticated HTTP proxy with Firefox

If you have access to the browser window, then the same technique will work for an authenticated proxy, but you will have to type the username and password in manually.
If this is not possible, for example on a remote CI server, then it is not currently clear how to do this. This document will be updated when more information is known. If you can help, please follow up on https://github.com/cobrateam/splinter/issues/359.

### 8.22 Frames, alerts and prompts

#### 8.22.1 Using iframes

You can use the `get_iframe` method and the `with` statement to interact with iframes. You can pass the iframe's name, id, index, or web element to `get_iframe`.

```python
with browser.get_iframe('iframemodal') as iframe:
    iframe.do_stuff()
```

#### 8.22.2 Handling alerts and prompts

Chrome support for alerts and prompts is new in Splinter 0.4.

**IMPORTANT:** Only webdriver (Firefox and Chrome) has support for alerts and prompts.

You can interact with alerts and prompts using the `get_alert` method.

```python
alert = browser.get_alert()
alert.text
alert.accept()
alert.dismiss()
```

In case of prompts, you can answer it using the `send_keys` method.

```python
prompt = browser.get_alert()
prompt.text
prompt.send_keys('text')
prompt.accept()
prompt.dismiss()
```

You can also use the `with` statement to interact with both alerts and prompts.

```python
with browser.get_alert() as alert:
    alert.do_stuff()
```

If there’s no prompt or alert, `get_alert` will return `None`. Remember to always use at least one of the alert/prompt ending methods (accept and dismiss). Otherwise, your browser instance will be frozen until you accept or dismiss the alert/prompt correctly.

### 8.23 API Documentation

Welcome to the Splinter API documentation! Check what’s inside:
8.23.1 Browser

`splinter.browser.Driver` *(driver_name='firefox', retry_count=3, *args, **kwargs)*

Returns a driver instance for the given name.

When working with `firefox`, it's possible to provide a profile name and a list of extensions.

If you don’t provide any `driver_name`, then `firefox` will be used.

If there is no driver registered with the provided `driver_name`, this function will raise a `splinter.exceptions.DriverNotFoundError` exception.

8.23.2 DriverAPI

```python
class splinter.driver.DriverAPI
    Basic driver API class.

    back()
    Back to the last URL in the browsing history.

    check(name)
    Checks a checkbox by its name.

    >>> browser.check("agree-with-terms")

    If you call `browser.check` n times, the checkbox keeps checked, it never get unchecked.

    To uncheck a checkbox, take a look in the `uncheck` method.

    choose(name, value)
    Chooses a value in a radio buttons group.

    Suppose you have the two radio buttons in a page, with the name `gender` and values ‘F’ and ‘M’. If you use the `choose` method the following way:

    >>> browser.choose('gender', 'F')

    Then you're choosing the female gender.

    click_link_by_href(href)
    Clicks in a link by its `href` attribute.

    click_link_by_id(id)
    Clicks in a link by id.

    click_link_by_partial_href(partial_href)
    Clicks in a link by looking for partial content of `href` attribute.

    click_link_by_partial_text(partial_text)
    Clicks in a link by partial content of its text.

    click_link_by_text(text)
    Clicks in a link by its `text`.

    cookies
    A `CookieManager` instance.

    For more details, check the `cookies manipulation section`.```
**evaluate_script**(*script, *args*)
Similar to `execute_script` method.
Executes javascript in the browser and returns the value of the expression.

```
>>> assert 4 == browser.evaluate_script('2 + 2')
```

**execute_script**(*script, *args*)
Executes a given JavaScript in the browser.

```
>>> browser.execute_script('document.getElementById("body").innerHTML = "<p>Hello world!</p>"')
```

**fill**(*name, value*)
Fill the field identified by `name` with the content specified by `value`.

**fill_form**(*field_values, form_id=None, name=None*)
Fill the fields identified by `name` with the content specified by `value` in a dict.
Currently, fill_form supports the following fields: text, password, textarea, checkbox, radio and select.
Checkboxes should be specified as a boolean in the dict.

**find_by_css**(*css_selector*)
Returns an instance of `ElementList`, using a CSS selector to query the current page content.

**find_by_id**(*id*)
Finds an element in current page by its id.
Even when only one element is find, this method returns an instance of `ElementList`.

**find_by_name**(*name*)
Finds elements in current page by their name.
Returns an instance of `ElementList`.

**find_by_tag**(*tag*)
Find all elements of a given tag in current page.
Returns an instance of `ElementList`.

**find_by_text**(*text*)
Finds elements in current page by their text.
Returns an instance of `ElementList`.

**find_by_value**(*value*)
Finds elements in current page by their value.
Returns an instance of `ElementList`.

**find_by_xpath**(*xpath*)
Returns an instance of `ElementList`, using a xpath selector to query the current page content.

**find_link_by_href**(*href*)
Find all elements of a given tag in current page.
Returns an instance of `ElementList`
**`find_link_by_partial_href`** (<code>partial_href</code>)

Find links by looking for a partial string in their href attribute.

Returns an instance of `ElementList`.

**`find_link_by_partial_text`** (<code>partial_text</code>)

Find links by looking for a partial string in their text.

Returns an instance of `ElementList`.

**`find_link_by_text`** (<code>text</code>)

Find links querying for their text.

Returns an instance of `ElementList`.

**`find_option_by_text`** (<code>text</code>)

Finds `<option>` elements by their text.

Returns an instance of `ElementList`.

**`find_option_by_value`** (<code>value</code>)

Finds `<option>` elements by their value.

Returns an instance of `ElementList`.

**`forward`** ()

Forward to the next URL in the browsing history.

If there is no URL to forward, this method does nothing.

**`get_alert`** ()

Changes the context for working with alerts and prompts.

For more details, check the [docs about iframes, alerts and prompts](#).

**`get_iframe`** (<code>name</code>)

Changes the context for working with iframes.

For more details, check the [docs about iframes, alerts and prompts](#).

**`html`**

Source of current page.

**`is_element_not_present_by_css`** (<code>css_selector</code>, <code>wait_time=None</code>)

Verify if the element is not present in the current page by css, and wait the specified time in <code>wait_time</code>. Returns True if the element is not present and False if is present.

**`is_element_not_present_by_id`** (<code>id</code>, <code>wait_time=None</code>)

Verify if the element is present in the current page by id, and wait the specified time in <code>wait_time</code>. Returns True if the element is not present and False if is present.

**`is_element_not_present_by_name`** (<code>name</code>, <code>wait_time=None</code>)

Verify if the element is not present in the current page by name, and wait the specified time in <code>wait_time</code>. Returns True if the element is not present and False if is present.

**`is_element_not_present_by_tag`** (<code>tag</code>, <code>wait_time=None</code>)

Verify if the element is not present in the current page by tag, and wait the specified time in <code>wait_time</code>. Returns True if the element is not present and False if is present.

**`is_element_not_present_by_text`** (<code>text</code>, <code>wait_time=None</code>)

Verify if the element is not present in the current page by text, and wait the specified time in <code>wait_time</code>. Returns True if the element is not present and False if is present.
**is_element_not_present_by_value** *(value, wait_time=None)*
Verify if the element is not present in the current page by value, and wait the specified time in `wait_time`.

Returns True if the element is not present and False if is present.

**is_element_not_present_by_xpath** *(xpath, wait_time=None)*
Verify if the element is not present in the current page by xpath, and wait the specified time in `wait_time`.

Returns True if the element is not present and False if is present.

**is_element_present_by_css** *(css_selector, wait_time=None)*
Verify if the element is present in the current page by css, and wait the specified time in `wait_time`.

Returns True if the element is present and False if is not present.

**is_element_present_by_id** *(id, wait_time=None)*
Verify if the element is present in the current page by id, and wait the specified time in `wait_time`.

Returns True if the element is present and False if is not present.

**is_element_present_by_name** *(name, wait_time=None)*
Verify if the element is present in the current page by name, and wait the specified time in `wait_time`.

Returns True if the element is present and False if is not present.

**is_element_present_by_tag** *(tag, wait_time=None)*
Verify if the element is present in the current page by tag, and wait the specified time in `wait_time`.

Returns True if the element is present and False if is not present.

**is_element_present_by_text** *(text, wait_time=None)*
Verify if the element is present in the current page by text, and wait the specified time in `wait_time`.

Returns True if the element is present and False if is not present.

**is_element_present_by_xpath** *(xpath, wait_time=None)*
Verify if the element is present in the current page by xpath, and wait the specified time in `wait_time`.

Returns True if the element is present and False if is not present.

**is_text_present** *(text, wait_time=None)*
Searchs for `text` in the browser and wait the seconds specified in `wait_time`.

Returns True if finds a match for the `text` and False if not.

**quit** *
Quits the browser, closing its windows (if it has one).

After quit the browser, you can’t use it anymore.

**reload** *
Revisits the current URL.

**screenshot** *(name=None, suffix=None)*
Takes a screenshot of the current page and saves it locally.

**select** *(name, value)*
Selects an `<option>` element in an `<select>` element using the `name` of the `<select>` and the value of the `<option>`.
Example:

```python
>>> browser.select("state", "NY")
```

title
Title of current page.

type (name, value, slowly=False)
Types the value in the field identified by name.
It’s useful to test javascript events like keyPress, keyUp, keyDown, etc.
If slowly is True, this function returns an iterator which will type one character per iteration.

uncheck (name)
Unchecks a checkbox by its name.
Example:

```python
>>> browser.uncheck("send-me-emails")
```

If you call browser.uncheck n times, the checkbox keeps unchecked, it never get checked.
To check a checkbox, take a look in the check method.

url
URL of current page.

visit (url)
Visits a given URL.
The url parameter is a string.

8.23.3 ElementAPI

class splinter.driver.ElementAPI
Basic element API class.
Any element in the page can be represented as an instance of ElementAPI.
Once you have an instance, you can easily access attributes like a dict:

```python
>>> element = browser.find_by_id("link-logo").first
>>> assert element['href'] == 'https://splinter.readthedocs.io'
```

You can also interact with the instance using the methods and properties listed below.

check ()
Checks the element, if it’s “checkable” (e.g.: a checkbox).
If the element is already checked, this method does nothing. For unchecking elements, take a loot in the uncheck method.

checked
Boolean property that says if the element is checked or not.

Example:

```python
>>> element.check()
>>> assert element.checked
>>> element.uncheck()
>>> assert not element.checked
```
clear()
Reset the field value.

click()
Clicks in the element.

fill(value)
Fill the field with the content specified by value.

has_class(class_name)
Indicates whether the element has the given class.

mouse_out()
Moves the mouse away from the element.

mouse_over()
Puts the mouse over the element.

screenshot()
Take screenshot of the element.

select(value, slowly=False)
Selects an `<option>` element in the element using the value of the `<option>`.
Example:

```python
>>> element.select("NY")
```

text
String of all of the text within the element. HTML tags are stripped.

type(value, slowly=False)
Types the value in the field.
It’s useful to test javascript events like keyPress, keyUp, keyDown, etc.
If slowly is True, this function returns an iterator which will type one character per iteration.
Also supports selenium.
Example:

```python
>>> from selenium.webdriver.common.keys import Keys
>>> ElementAPI.type(Keys.RETURN)
```

uncheck()
Unchecks the element, if it’s “checkable” (e.g.: a checkbox).
If the element is already unchecked, this method does nothing. For checking elements, take a loot in the check method.

value
Value of the element, usually a form element

visible
Boolean property that says if the element is visible or hidden in the current page.

8.23.4 CookieManager

class splinter.cookie_manager.CookieManagerAPI
An API that specifies how a splinter driver deals with cookies.
You can add cookies using the `add` method, and remove one or all cookies using the `delete` method.

A CookieManager acts like a `dict`, so you can access the value of a cookie through the `[]` operator, passing the cookie identifier:

```python
>>> cookie_manager.add({'name': 'Tony'})
>>> assert cookie_manager['name'] == 'Tony'
```

### add(cookies)

Adds a cookie.

The `cookie` parameter is a `dict` where each key is an identifier for the cookie value (like any `dict`).

Example of use:

```python
>>> cookie_manager.add({'name': 'Tony'})
```

### all(verbos=False)

Returns all of the cookies.

**Note:** If you’re using any webdriver and want more info about the cookie, set the `verbose` parameter to `True` (in other drivers, it won’t make any difference). In this case, this method will return a list of dicts, each with one cookie’s info.

Examples:

```python
>>> cookie_manager.add({'name': 'Tony'})
>>> cookie_manager.all()
[{'name': 'Tony'}]
```

### delete(*cookies)

Deletes one or more cookies. You can pass all the cookies identifier that you want to delete.

If none identifier is provided, all cookies are deleted.

Examples:

```python
>>> cookie_manager.delete() # deletes all cookies
>>> cookie_manager.delete('name', 'birthday', 'favorite_color') # deletes these three cookies
>>> cookie_manager.delete('name') # deletes one cookie
```

---

### 8.23.5 ElementList

**class** `splinter.element_list.ElementList` *(list, driver=None, find_by=None, query=None)*

**Bases:** `object`

List of elements. Each member of the list is (usually) an instance of `ElementAPI`.

Beyond the traditional list methods, the `ElementList` provides some other methods, listed below.

There is a peculiar behavior on `ElementList`: you never get an `IndexError`. Instead, you can an `ElementDoesNotExist` exception when trying to access an inexistent item in the list:

```python
>>> element_list = ElementList([])
>>> element_list[0] # raises ElementDoesNotExist
```

**first**

An alias to the first element of the list:
 assert element_list[0] == element_list.first

**is_empty()**
Returns True if the list is empty.

**last**
An alias to the last element of the list:

 assert element_list[-1] == element_list.last

### 8.23.6 Request handling

**class splinter.request_handler.status_code.StatusCode**(status code, reason)

code = None
Code of the response (example: 200)

is_success()
Returns True if the response was succeed, otherwise, returns False.

reason = None
A message for the response (example: Success)

### 8.23.7 Exceptions

**class splinter.exceptions.DriverNotFoundError**
Exception raised when a driver is not found.

Example:

```python
>>> from splinter import Browser
>>> b = Browser('unknown driver') # raises DriverNotFoundError
```

**class splinter.exceptions.ElementDoesNotExist**
Exception raised when an element is not found in the page.

The exception is raised only when someone tries to access the element, not when the driver is finding it.

Example:

```python
>>> elements = browser.find_by_id('unknown-id') # returns an empty list
>>> elements[0] # raises ElementDoesNotExist
```

### 8.24 Selenium Keys

With Splinter You can use selenium keys implementation.

Here is a simple example:

```python
from selenium.webdriver.common.keys import Keys
ElementAPI.type(Keys.RETURN)
```

Full list of all support keys can be found on official selenium documentation [selenium.webdriver.common.keys](https://selenium-python.readthedocs.io/api.html#selenium.webdriver.common.keys.Keys)
8.25 Community

8.25.1 mailing list

- splinter-users - list for help and announcements
- splinter-developers - where the developers of splinter itself discuss new features

8.25.2 irc channel

#cobrateam channel on irc.freenode.net - chat with other splinter users and developers

8.25.3 ticket system

ticket system - report bugs and make feature requests

8.25.4 splinter around the world

Projects using splinter

- salad: splinter and lettuce integration

Blog posts

- Django Full Stack Testing and BDD with Lettuce and Splinter

Slides and talks

- [pt-br] Os complicados testes de interface
- [pt-br] Testes de aceitação com Lettuce e Splinter

8.26 Contribute

- Source hosted at GitHub
- Report issues on GitHub Issues

Pull requests are very welcome! Make sure your patches are well tested and documented :)

If you want to add any new driver, check out our docs for creating new splinter drivers.

8.26.1 running the tests

If you are using a virtualenv, all you need is:

```
$ make test
```

You can also specify one or more test files to run:
You can pass which test files you want to run, separated by comma, to the `which` variable.

### 8.26.2 some conventions we like

You can feel free to create and pull request new branches to Splinter project. When adding support for new drivers, we usually work in a separated branch.

### 8.26.3 writing docs

Splinter documentation is written using Sphinx, which uses RST. We use the Read the Docs Sphinx Theme. Check these tools’ docs to learn how to write docs for Splinter.

### 8.26.4 building docs

In order to build the HTML docs, just navigate to the project folder (the main folder, not the `docs` folder) and run the following on the terminal:

```
$ make doc
```

The requirements for building the docs are specified in `doc-requirements.txt` in the project folder.

### 8.27 Writing new splinter drivers

The process of creating a new splinter browser is really simple: you just need to implement a `TestCase` (extending `tests.base.BaseBrowserTests`) and make all tests green. For example:

Imagine you’re creating the Columbia driver, you would add the `test_columbia.py` file containing some code like...

```python
from splinter import Browser
from tests.base import BaseBrowserTests

class ColumbiaTest(BaseBrowserTests):
    
    @classmethod
    def setUpClass(cls):
        cls.browser = Browser('columbia')
        # ...
```

Now, to make the test green, you need to implement methods provided by the `DriverAPI` and the `ElementAPI`. Use `make test` to run the tests:

```
$ make test which=tests/test_columbia.py
```
8.28 Setting up your splinter development environment

Setting up a splinter development environment is a really easy task. Once you have some basic development tools on your machine, you can set up the entire environment with just one command.

8.28.1 Basic development tools

Let’s deal with those tools first.

macOS

If you’re a macOS user, you just need to install Xcode, which can be downloaded from Mac App Store (on Snow Leopard or later) or from Apple website.

Linux

If you are running a Linux distribution, you need to install some basic development libraries and headers. For example, on Ubuntu, you can easily install all of them using apt-get:

```
$ [sudo] apt-get install build-essential python-dev libxml2-dev libxslt1-dev
```

**PIP and virtualenv**

Make sure you have pip installed. We manage all splinter development dependencies with PIP, so you should use it too.

And please, for the sake of a nice development environment, use virtualenv. If you aren’t using it yet, start now. :)

**Dependencies**

Once you have all development libraries installed for your OS, just install all splinter development dependencies with make:

```
$ [sudo] make dependencies
```

**Note:** You will need sudo only if you aren’t using virtualenv (which means you’re a really bad guy - *no donuts for you*).

Also make sure you have properly configured your *Chrome driver*. 
Python Module Index

S
splinter.browser, 47
splinter.cookie_manager, 52
splinter.driver, 47
splinter.driver.djangoclient, 44
splinter.driver.flaskclient, 44
splinter.driver.webdriver.firefox, 36
splinter.driver.zopetestbrowser, 43
splinter.element_list, 53
splinter.exceptions, 54
splinter.request_handler.status_code, 54
A
add() (splinter.cookie_manager.CookieManagerAPI method), 53
all() (splinter.cookie_manager.CookieManagerAPI method), 53
attach_file() (splinter.driver.webdriver.chrome.WebDriver method), 32
attach_file() (splinter.driver.webdriver.firefox.WebDriver method), 38

B
back() (splinter.driver.DriverAPI method), 47
back() (splinter.driver.webdriver.chrome.WebDriver method), 32
back() (splinter.driver.webdriver.firefox.WebDriver method), 38
Browser() (in module splinter.browser), 47

C
check() (splinter.driver.DriverAPI method), 47
check() (splinter.driver.ElementAPI method), 51
check() (splinter.driver.webdriver.chrome.WebDriver method), 32
check() (splinter.driver.webdriver.firefox.WebDriver method), 38
checked (splinter.driver.ElementAPI attribute), 51
choose() (splinter.driver.DriverAPI method), 47
choose() (splinter.driver.webdriver.chrome.WebDriver method), 32
choose() (splinter.driver.webdriver.firefox.WebDriver method), 38
clear() (splinter.driver.ElementAPI method), 51
click() (splinter.driver.ElementAPI method), 52
click_link_by_href() (splinter.driver.DriverAPI method), 47
click_link_by_href() (splinter.driver.webdriver.chrome.WebDriver method), 32
click_link_by_href() (splinter.driver.webdriver.firefox.WebDriver method), 38
click_link_by_id() (splinter.driver.DriverAPI method), 47
click_link_by_id() (splinter.driver.webdriver.chrome.WebDriver method), 32
click_link_by_id() (splinter.driver.webdriver.firefox.WebDriver method), 38
click_link_by_partial_href() (splinter.driver.DriverAPI method), 47
click_link_by_partial_href() (splinter.driver.webdriver.chrome.WebDriver method), 32
click_link_by_partial_href() (splinter.driver.webdriver.firefox.WebDriver method), 38
click_link_by_partial_text() (splinter.driver.DriverAPI method), 47
click_link_by_partial_text() (splinter.driver.webdriver.chrome.WebDriver method), 33
click_link_by_partial_text() (splinter.driver.webdriver.firefox.WebDriver method), 38
click_link_by_text() (splinter.driver.DriverAPI method), 47
click_link_by_text() (splinter.driver.webdriver.chrome.WebDriver method), 33
click_link_by_text() (splinter.driver.webdriver.firefox.WebDriver method), 38
code (splinter.request_handler.status_code.StatusCode attribute), 54
CookieManagerAPI (class in splinter.cookie_manager), 52
cookies (splinter.driver.DriverAPI attribute), 47
cookies (splinter.driver.webdriver.chrome.WebDriver attribute), 33
cookies (splinter.driver.webdriver.firefox.WebDriver attribute), 38

delete() (splinter.cookie_manager.CookieManagerAPI method), 53
DriverAPI (class in splinter.driver), 47
DriverNotFoundError (class in splinter.exceptions), 54

ElementAPI (class in splinter.driver), 51
ElementDoesNotExist (class in splinter.exceptions), 54
ElementList (class in splinter.element_list), 53
evaluate_script() (splinter.driver.DriverAPI method), 47
evaluate_script() (splinter.driver.webdriver.chrome.WebDriver method), 33
evaluate_script() (splinter.driver.webdriver.firefox.WebDriver method), 38
execute_script() (splinter.driver.DriverAPI method), 48
execute_script() (splinter.driver.webdriver.chrome.WebDriver method), 33
execute_script() (splinter.driver.webdriver.firefox.WebDriver method), 39

fill() (splinter.driver.DriverAPI method), 48
fill() (splinter.driver.ElementAPI method), 52
fill() (splinter.driver.webdriver.chrome.WebDriver method), 33
fill() (splinter.driver.webdriver.firefox.WebDriver method), 39
fill_form() (splinter.driver.DriverAPI method), 48
fill_form() (splinter.driver.webdriver.chrome.WebDriver method), 33
fill_form() (splinter.driver.webdriver.firefox.WebDriver method), 39
find_by_css() (splinter.driver.webdriver.chrome.WebDriver method), 33
find_by_css() (splinter.driver.webdriver.firefox.WebDriver method), 39
find_by_id() (splinter.driver.DriverAPI method), 48
find_by_id() (splinter.driver.webdriver.chrome.WebDriver method), 33
find_by_id() (splinter.driver.webdriver.firefox.WebDriver method), 39
find_by_name() (splinter.driver.DriverAPI method), 48
find_by_name() (splinter.driver.webdriver.chrome.WebDriver method), 33
find_by_name() (splinter.driver.webdriver.firefox.WebDriver method), 39
find_by_tag() (splinter.driver.DriverAPI method), 48
find_by_tag() (splinter.driver.webdriver.chrome.WebDriver method), 33
find_by_tag() (splinter.driver.webdriver.firefox.WebDriver method), 39
find_by_text() (splinter.driver.DriverAPI method), 48
find_by_text() (splinter.driver.webdriver.chrome.WebDriver method), 33
find_by_text() (splinter.driver.webdriver.firefox.WebDriver method), 39
find_by_value() (splinter.driver.DriverAPI method), 48
find_by_value() (splinter.driver.webdriver.chrome.WebDriver method), 34
find_by_value() (splinter.driver.webdriver.firefox.WebDriver method), 39
find_by_xpath() (splinter.driver.DriverAPI method), 48
find_by_xpath() (splinter.driver.webdriver.chrome.WebDriver method), 34
find_by_xpath() (splinter.driver.webdriver.firefox WebDriver method), 39
find_link_by_href() (splinter.driver.DriverAPI method), 48
find_link_by_href() (splinter.driver.webdriver.chrome.WebDriver method), 34
find_link_by_href() (splinter.driver.webdriver.firefox.WebDriver method), 39
find_link_by_partial_href() (splinter.driver.DriverAPI method), 48
find_link_by_partial_href() (splinter.driver.webdriver.chrome.WebDriver method), 34
find_link_by_partial_href() (splinter.driver.webdriver.firefox.WebDriver method), 39
find_link_by_partial_text() (splinter.driver.DriverAPI method), 49
find_link_by_partial_text() (splinter.driver.webdriver.chromeWebDriver method), 34
find_link_by_partial_text() (splinter.driver.webdriver.firefox.WebDriver method), 40
find_link_by_text() (splinter.driver.DriverAPI method), 49
find_link_by_text() (splinter.driver.webdriver.chrome.WebDriver method), 34
find_link_by_text() (splinter.driver.webdriver.firefox.WebDriver method), 40
find_option_by_text() (splinter.driver.WebDriver method), 49
find_option_by_text() (splinter.driver.webdriver.chrome.WebDriver method), 34
find_option_by_text() (splinter.driver.webdriver.firefox.WebDriver method), 40
find_option_by_value() (splinter.driver.WebDriver method), 49
find_option_by_value() (splinter.driver.webdriver.chrome.WebDriver method), 34
find_option_by_value() (splinter.driver.webdriver.firefox.WebDriver method), 40
first (splinter.element_list.ElementList attribute), 53
forward() (splinter.driver.DriverAPI method), 49
forward() (splinter.driver.webdriver.chrome.WebDriver method), 34
forward() (splinter.driver.webdriver.firefox.WebDriver method), 40
get_alert() (splinter.driver.DriverAPI method), 49
get_alert() (splinter.driver.webdriver.chrome.WebDriver method), 34
get_alert() (splinter.driver.webdriver.firefox.WebDriver method), 40
get_iframe() (splinter.driver.DriverAPI method), 49
get_iframe() (splinter.driver.webdriver.chrome.WebDriver method), 34
get_iframe() (splinter.driver.webdriver.firefox.WebDriver method), 40
has_class() (splinter.driver.ElementAPI method), 52
html (splinter.driver.WebDriver attribute), 49
html (splinter.driver.webdriver.chrome.WebDriver attribute), 34
html (splinter.driver.webdriver.firefox.WebDriver attribute), 40
html_snapshot() (splinter.driver webdriver.chrome.WebDriver method), 34
html_snapshot() (splinter.driver.webdriver.firefox.WebDriver method), 40
is_element_not_present_by_css() (splinter.driver.DriverAPI method), 49
is_element_not_present_by_css() (splinter.driver.webdriver.chrome.WebDriver method), 34
is_element_not_present_by_css() (splinter.driver.webdriver.firefox.WebDriver method), 40
is_element_not_present_by_id() (splinter.driver.WebDriver method), 49
is_element_not_present_by_id() (splinter.driver.webdriver.chrome.WebDriver method), 35
is_element_not_present_by_id() (splinter.driver.webdriver.firefox.WebDriver method), 40
is_element_not_present_by_name() (splinter.driver.WebDriver method), 49
is_element_not_present_by_name() (splinter.driver.webdriver.chrome.WebDriver method), 35
is_element_not_present_by_name() (splinter.driver.webdriver.firefox.WebDriver method),
is_element_not_present_by_tag() (splinter.driver.DriverAPI method), 49
is_element_not_present_by_tag() (splinter.driver.webdriver.chrome.WebDriver method), 35
is_element_not_present_by_tag() (splinter.driver.webdriver.firefox.WebDriver method), 40
is_element_not_present_by_text() (splinter.driver.DriverAPI method), 49
is_element_not_present_by_text() (splinter.driver.webdriver.chrome.WebDriver method), 35
is_element_not_present_by_text() (splinter.driver.webdriver.firefox.WebDriver method), 40
is_element_not_present_by_value() (splinter.driver.DriverAPI method), 49
is_element_not_present_by_value() (splinter.driver.webdriver.chrome.WebDriver method), 35
is_element_not_present_by_value() (splinter.driver.webdriver.firefox.WebDriver method), 41
is_element_not_present_by_xpath() (splinter.driver.DriverAPI method), 50
is_element_not_present_by_xpath() (splinter.driver.webdriver.chrome.WebDriver method), 35
is_element_not_present_by_xpath() (splinter.driver.webdriver.firefox.WebDriver method), 41
is_element_present_by_css() (splinter.driver.DriverAPI method), 50
is_element_present_by_css() (splinter.driver.webdriver.chrome.WebDriver method), 35
is_element_present_by_css() (splinter.driver.webdriver.firefox.WebDriver method), 41
is_element_present_by_id() (splinter.driver.DriverAPI method), 50
is_element_present_by_id() (splinter.driver.webdriver.chrome.WebDriver method), 35
is_element_present_by_id() (splinter.driver.webdriver.firefox.WebDriver method), 41
is_element_present_by_name() (splinter.driver.DriverAPI method), 50
is_element_present_by_name() (splinter.driver.webdriver.chrome.WebDriver method), 35
is_element_present_by_name() (splinter.driver.webdriver.firefox.WebDriver method), 41
is_element_present_by_text() (splinter.driver.DriverAPI method), 50
is_element_present_by_text() (splinter.driver.webdriver.chrome.WebDriver method), 35
is_element_present_by_text() (splinter.driver.webdriver.firefox.WebDriver method), 41
is_element_present_by_value() (splinter.driver.DriverAPI method), 50
is_element_present_by_value() (splinter.driver.webdriver.chromeWebDriver method), 35
is_element_present_by_value() (splinter.driver.webdriver.firefox.WebDriver method), 41
is_element_present_by_xpath() (splinter.driver.DriverAPI method), 50
is_element_present_by_xpath() (splinter.driver.webdriver.chrome.WebDriver method), 35
is_element_present_by_xpath() (splinter.driver.webdriver.firefox.WebDriver method), 41

L
last (splinter.element_list.ElementList attribute), 54

M
mouse_out() (splinter.driver.ElementAPI method), 52
mouse_over() (splinter.driver.ElementAPI method), 52
quit() (splinter.driver.DriverAPI method), 50
quit() (splinter.driver.webdriver.chrome.WebDriver method), 36
quit() (splinter.driver.webdriver.firefox.WebDriver method), 41
reason (splinter.request_handler.status_code.StatusCode attribute), 54
reload() (splinter.driver.DriverAPI method), 50
reload() (splinter.driver.webdriver.chrome.WebDriver method), 36
reload() (splinter.driver.webdriver.firefox.WebDriver method), 41
screenshot() (splinter.driver.DriverAPI method), 50
screenshot() (splinter.driver.ElementAPI method), 52
screenshot() (splinter.driver.webdriver.chrome.WebDriver method), 36
screenshot() (splinter.driver.webdriver.firefox.WebDriver method), 41
select() (splinter.driver.DriverAPI method), 50
select() (splinter.driver.ElementAPI method), 52
select() (splinter.driver.webdriver.chrome.WebDriver method), 36
select() (splinter.driver.webdriver.firefox.WebDriver method), 41
splinter.browser (module), 47
splinter.cookie_manager (module), 52
splinter.driver (module), 47
splinter.driver.djangoclient (module), 44
splinter.driver.flaskclient (module), 44
splinter.driver.webdriver.firefox (module), 36
splinter.driver.zopetestbrowser (module), 43
splinter.element_list (module), 53
splinter.exceptions (module), 54
splinter.request_handler.status_code (module), 54
StatusCode (class in splinter.request_handler.status_code), 54
text (splinter.driver.ElementAPI attribute), 52
title (splinter.driver.webdriver.chrome.WebDriver attribute), 36
title (splinter.driver.webdriver.firefox.WebDriver attribute), 42
type() (splinter.driver.DriverAPI method), 51
type() (splinter.driver.ElementAPI method), 52
type() (splinter.driver.webdriver.chrome.WebDriver method), 36
type() (splinter.driver.webdriver.firefox.WebDriver method), 42
uncheck() (splinter.driver.DriverAPI method), 51
uncheck() (splinter.driver.ElementAPI method), 52
uncheck() (splinter.driver.webdriver.chrome.WebDriver method), 36
uncheck() (splinter.driver.webdriver.firefox.WebDriver method), 42
url (splinter.driver.DriverAPI attribute), 51
url (splinter.driver.webdriver.chrome.WebDriver attribute), 36
url (splinter.driver.webdriver.firefox.WebDriver attribute), 42
value (splinter.driver.ElementAPI attribute), 52
visible (splinter.driver.ElementAPI attribute), 52
visit() (splinter.driver.DriverAPI method), 51
visit() (splinter.driver.webdriver.chrome.WebDriver method), 36
visit() (splinter.driver.webdriver.firefox.WebDriver method), 42
WebDriver (class in splinter.driver.webdriver.chrome), 32
WebDriver (class in splinter.driver.webdriver.firefox), 38