
CasperJs Documentation

Release 1.1.0-DEV

Nicolas Perriault

Apr 04, 2017

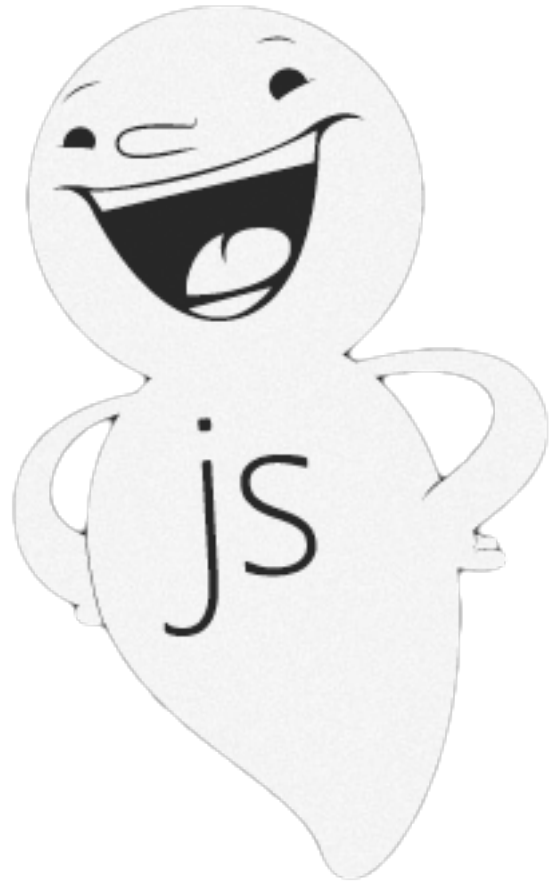
Contents

1	Installation	3
1.1	Prerequisites	3
1.2	Installing from Homebrew (OSX)	4
1.3	Installing from npm	4
1.4	Installing from git	4
1.5	Installing from an archive	5
1.6	CasperJS on Windows	5
1.7	Known Bugs & Limitations	6
2	Quickstart	7
2.1	A minimal scraping script	7
2.2	Now let's scrape Google!	8
2.3	CoffeeScript version	9
2.4	A minimal testing script	10
3	Using the command line	11
3.1	<i>casperjs</i> native options	12
3.2	Raw parameter values	13
4	Selectors	15
4.1	CSS3	15
4.2	XPath	16
5	Testing	17
5.1	Unit testing	17
5.2	Browser tests	18
5.3	Setting Casper options in the test environment	20
5.4	Advanced techniques	20
5.5	Test command args and options	20
5.6	Exporting results in XUnit format	21
5.7	CasperJS own tests	22
5.8	Extending Casper for Testing	23
6	API Documentation	25
6.1	The <i>casper</i> module	25
6.2	The <i>clientutils</i> module	68
6.3	The <i>colorizer</i> module	76

6.4	The <code>mouse</code> module	78
6.5	The <code>tester</code> module	81
6.6	The <code>utils</code> module	97
7	Writing CasperJS modules	103
8	Events & filters	105
8.1	Events	105
8.2	Filters	115
9	Logging	117
10	Extending	119
10.1	Using CoffeeScript	121
11	Debugging	123
11.1	Use the verbose mode	123
11.2	Hook in the deep using events	124
11.3	Dump serialized values to the console	124
11.4	Localize yourself in modules	124
11.5	Name your closures	124
12	FAQ	127
12.1	Is CasperJS a node.js library?	128
12.2	I'm stuck! I think there's a bug! What can I do?	128
12.3	The <code>casper.test</code> property is undefined, I can't write any test!	128
12.4	I keep copy and pasting stuff in my test scripts, that's boring	128
12.5	What is the versioning policy of CasperJS?	128
12.6	Can I use jQuery with CasperJS?	129
12.7	Can I use CasperJS without using the <code>casperjs</code> executable?	129
12.8	How can I catch HTTP 404 and other status codes?	130
12.9	Where does CasperJS write its logfile?	130
12.10	What's this mysterious <code>__utils__</code> object?	130
12.11	How does <code>then()</code> and the step stack work?	131
12.12	I'm having hard times downloading files using <code>download()</code>	132
12.13	Is it possible to achieve parallel browsing using CasperJS?	132
12.14	Can I access & manipulate DOM elements directly from the CasperJS environment?	132
12.15	Why can't I create a new <i>casper</i> instance in a test environment?	133
12.16	Okay, honestly, I'm stuck with Javascript.	133
12.17	How do I use PhantomJS page module API in <code>casperjs</code> ?	133
12.18	How do I provide my implementation of a remote resource?	134
12.19	I'm getting intermittent test failure, what can I do to fix them?	134
13	Cookbook	135
13.1	Creating a web service	135
13.2	Script to automatically check a page for 404 and 500 errors	136
13.3	Test drag&drop	138
13.4	Passing parameters into your tests	139
14	Changelog	141
15	Upgrading	143
15.1	Upgrading to 1.1	143
16	Known Issues	147
16.1	PhantomJS	147

17 Credits	149
17.1 Author	149
17.2 Contributors	149
17.3 Logo	150
18 License	153
19 Community	155

CasperJS is a navigation scripting & testing utility for the PhantomJS (WebKit) and SlimerJS (Gecko) headless browsers, written in Javascript.



CHAPTER 1

Installation

CasperJS can be installed on Mac OSX, Windows and most Linuxes.

Prerequisites

- [PhantomJS 1.9.1](#) or greater. Please read the [installation instructions for PhantomJS](#)
- [Python 2.6](#) or greater for `casperjs` in the `bin/` directory

Note: CoffeeScript is not natively supported in PhantomJS versions 2.0.0 and above. If you are going to use CoffeeScript you'll have to transpile it into vanilla Javascript. See [known issues](#) for more details.

New in version 1.1.

- **Experimental:** as of 1.1.0-beta1, [SlimerJS 0.8](#) or greater to run your tests against Gecko (Firefox) instead of Webkit (just add `-engine=slimerjs` to your command line options). The SlimerJS developers documented [the PhantomJS API compatibility of SlimerJS](#) as well as [the differences between PhantomJS and SlimerJS](#). Note that it is known that coffescript support breaks as of [SlimerJS 0.9.6](#); we are investigating that issue.

New in version 1.1.0-beta4.

Warning: Versions before 1.1.0-beta4 that were installed through npm required an unspecific PhantomJS version by means of an npm dependency. This led to lots of confusion and issues against CasperJS not working properly if installed through npm. Starting with 1.1.0 the installation of an engine (PhantomJS, SlimerJS) will be a real prerequisite, regardless of the installation method you choose for CasperJS.

Installing from Homebrew (OSX)

Installation of both PhantomJS and CasperJS can be achieved using [Homebrew](#), a popular package manager for Mac OS X.

Above all, don't forget to update Formulaes:

```
$ brew update
```

For the 1.1.* version (recommended):

```
$ brew install casperjs
```

If you have already installed casperjs and want to have the last release (stable|dev), use upgrade:

```
$ brew upgrade casperjs
```

Upgrade only update to the latest release branch (1.0.x|1.1.0-dev).

Installing from npm

New in version 1.1.0-beta3.

You can install CasperJS using [npm](#):

- For most users (current version 1.1.0-beta4):

```
$ npm install -g casperjs
```
- If you want a specific older version:
 - For beta3:

```
$ npm install -g casperjs@1.1.0-beta3
```
 - For beta2:

```
$ npm install -g casperjs@1.1.0-beta2
```
- If you want to install the current master from git using npm:

```
$ npm install -g git+https://github.com/casperjs/casperjs.git
```

Note: The `-g` flag makes the `casperjs` executable available system-wide.

Warning: While CasperJS is installable via npm, *it is not a NodeJS module* and will not work with NodeJS out of the box. **You cannot load casper by using `require('casperjs')` in node.** Note that CasperJS is not capable of using a vast majority of NodeJS modules out there. **Experiment and use your best judgement.**

Installing from git

Installation can be achieved using [git](#). The code is mainly hosted on [Github](#).

From the master branch

```
$ git clone git://github.com/casperjs/casperjs.git
$ cd casperjs
$ ln -sf `pwd`/bin/casperjs /usr/local/bin/casperjs
```

Once PhantomJS and CasperJS installed on your machine, you should obtain something like this:

```
$ phantomjs --version
1.9.2
$ casperjs
CasperJS version 1.1.0-beta4 at /Users/niko/Sites/casperjs, using phantomjs version 1.
↪ 9.2
# ...
```

Or if SlimerJS is your thing:

```
$ slimerjs --version
Innophi SlimerJS 0.8pre, Copyright 2012-2013 Laurent Jouanneau & Innophi
$ casperjs
CasperJS version 1.1.0 at /Users/niko/Sites/casperjs, using slimerjs version 0.8.0
```

You are now ready to write your *first script*!

Installing from an archive

You can download tagged archives of CasperJS code:

Latest development version (master branch):

- <https://github.com/casperjs/casperjs/zipball/master> (zip)
- <https://github.com/casperjs/casperjs/tarball/master> (tar.gz)

Latest stable version:

- <https://github.com/casperjs/casperjs/zipball/1.1.0> (zip)
- <https://github.com/casperjs/casperjs/tarball/1.1.0> (tar.gz)

Operations are then the same as with a git checkout.

CasperJS on Windows

Phantomjs installation additions

- Append ";C:\phantomjs" to your PATH environment variable.
- Modify this path appropriately if you installed PhantomJS to a different location.

Casperjs installation additions

New in version 1.1.0-beta3.

- Append "`C:\casperjs\bin`" to your `PATH` environment variable (for versions before 1.1.0-beta3 append "`C:\casperjs\batchbin`" to your `PATH` environment variable).
- Modify this path appropriately if you installed CasperJS to a different location.
- If your computer uses both discrete and integrated graphics you need to disable autoselect and explicitly choose graphics processor - otherwise `exit ()` will not exit casper.

You can now run any regular casper scripts that way:

```
C:> casperjs myscript.js
```

Colorized output

Note: New in version 1.1.0-beta1.

Windows users will get colorized output if `ansicon` is installed or if the user is using `ConEmu` with ANSI colors enabled.

Compilation (Optional)

- .NET Framework 3.5 or greater (or `Mono 2.10.8` or greater) for `casperjs.exe` in the `bin/` directory

Known Bugs & Limitations

- Due to its asynchronous nature, CasperJS doesn't work well with `PhantomJS' REPL`.

CHAPTER 2

Quickstart

Once CasperJS is *properly installed*, you can write your first script. You can use plain *Javascript* (or *CoffeeScript* with PhantomJS versions before 2.0).

Hint: If you're not too comfortable with Javascript, a *dedicated FAQ entry* is waiting for you.

A minimal scraping script

Fire up your favorite editor, create and save a `sample.js` file like below:

```
var casper = require('casper').create();

casper.start('http://casperjs.org/', function() {
  this.echo(this.getTitle());
});

casper.thenOpen('http://phantomjs.org', function() {
  this.echo(this.getTitle());
});

casper.run();
```

Run it (with python):

```
$ casperjs sample.js
```

Run it (with node):

```
$ node casperjs.js sample.js
```

Run it (with PhantomJS):

```
$ phantomjs casperjs.js sample.js
```

Run it (on windows):

```
C:\casperjs\bin> casperjs.exe sample.js
```

You should get something like this:

```
$ casperjs sample.js
CasperJS, a navigation scripting and testing utility for PhantomJS
PhantomJS: Headless WebKit with JavaScript API
```

What did we just do?

1. we created a new *Casper* instance
2. we started it and opened `http://casperjs.org/`
3. *once* the page has been loaded, we asked to print the title of that webpage (the content of its `<title>` tag)
4. *then* we opened another url, `http://phantomjs.org/`
5. *once* the new page has been loaded, we asked to print its title too
6. we executed the whole process

Now let's scrape Google!

In the following example, we'll query google for two terms consecutively, “*casperjs*” and “*phantomjs*”, aggregate the result links in a standard `Array` and output the result to the console.

Fire up your favorite editor and save the javascript code below in a `googlelinks.js` file:

```
var links = [];
var casper = require('casper').create();

function getLinks() {
    var links = document.querySelectorAll('h3.r a');
    return Array.prototype.map.call(links, function(e) {
        return e.getAttribute('href');
    });
}

casper.start('http://google.fr/', function() {
    // Wait for the page to be loaded
    this.waitForSelector('form[action="/search"]');
});

casper.then(function() {
    // search for 'casperjs' from google form
    this.fill('form[action="/search"]', { q: 'casperjs' }, true);
});

casper.then(function() {
    // aggregate results for the 'casperjs' search
```

```

links = this.evaluate(getLinks);
// now search for 'phantomjs' by filling the form again
this.fill('form[action="/search"]', { q: 'phantomjs' }, true);
});

casper.then(function() {
  // aggregate results for the 'phantomjs' search
  links = links.concat(this.evaluate(getLinks));
});

casper.run(function() {
  // echo results in some pretty fashion
  this.echo(links.length + ' links found:');
  this.echo(' - ' + links.join('\n - ')).exit();
});

```

Run it:

```

$ casperjs googlelinks.js
20 links found:
- https://github.com/casperjs/casperjs
- https://github.com/casperjs/casperjs/issues/2
- https://github.com/casperjs/casperjs/tree/master/samples
- https://github.com/casperjs/casperjs/commits/master/
- http://www.facebook.com/people/Casper-Js/100000337260665
- http://www.facebook.com/public/Casper-Js
- http://hashtags.org/tag/CasperJS/
- http://www.zerotohundred.com/newforums/members/casper-js.html
- http://www.yellowpages.com/casper-wy/j-s-enterprises
- http://local.trib.com/casperwy/j+s+chinese+restaurant.zq.html
- http://www.phantomjs.org/
- http://code.google.com/p/phantomjs/
- http://code.google.com/p/phantomjs/wiki/QuickStart
- http://svay.com/blog/index/post/2011/08/31/Paris-JS-10-%3A-Introduction-%C3%A0-
↳ PhantomJS
- https://github.com/ariya/phantomjs
- http://dailyjs.com/2011/01/28/phantoms/
- http://css.dzone.com/articles/phantom-js-alternative
- http://pilvee.com/blog/tag/phantom-js/
- http://ariya.blogspot.com/2011/01/phantomjs-minimalistic-headless-webkit.html
- http://www.readwriteweb.com/hack/2011/03/phantomjs-the-power-of-webkit.php

```

CoffeeScript version

You can also write Casper scripts using the [CoffeeScript](#) syntax:

```

getLinks = ->
  links = document.querySelectorAll "h3.r a"
  Array::map.call links, (e) -> e.getAttribute "href"

links = []
casper = require('casper').create()

casper.start "http://google.fr/", ->
  # search for 'casperjs' from google form

```

```
@fill "form[action='/search']", q: "casperjs", true

casper.then ->
  # aggregate results for the 'casperjs' search
  links = @evaluate getLinks
  # search for 'phantomjs' from google form
  @fill "form[action='/search']", q: "phantomjs", true

casper.then ->
  # concat results for the 'phantomjs' search
  links = links.concat @evaluate(getLinks)

casper.run ->
  # display results
  @echo links.length + " links found:"
  @echo(" - " + links.join("\n - ")).exit()
```

Just remember to suffix your script with the `.coffee` extension.

Note: CoffeeScript is not natively supported in PhantomJS versions 2.0.0 and above. If you are going to use CoffeeScript you'll have to transpile it into vanilla Javascript. See [known issues](#) for more details.

A minimal testing script

CasperJS is also a *testing framework*; test scripts are slightly different than scraping ones, though they share most of the API.

A simplest test script:

```
// hello-test.js
casper.test.begin("Hello, Test!", 1, function(test) {
  test.assert(true);
  test.done();
});
```

Run it using the `casperjs test` subcommand:

```
$ casperjs test hello-test.js
Test file: hello-test.js
# Hello, Test!
PASS Subject is strictly true
PASS 1 test executed in 0.023s, 1 passed, 0 failed, 0 dubious, 0 skipped.
```

Note: As you can see, there's no need to create a `casper` instance in a test script as a preconfigured one has already made available for you.

You can read more about testing in the *dedicated section*.

CHAPTER 3

Using the command line

CasperJS ships with a built-in command line parser on top of PhantomJS' parser, located in the `cli` module. It exposes passed arguments as **positional ones** and **named options**

A Casper instance always contains a ready-to-use `cli` property for easy access to these parameters, so you don't have to worry about manipulating the `cli` module parsing API.

Let's consider this simple casper script:

```
var casper = require("casper").create();

casper.echo("Casper CLI passed args:");
require("utils").dump(casper.cli.args);

casper.echo("Casper CLI passed options:");
require("utils").dump(casper.cli.options);

casper.exit();
```

Note: Please note the two `casper-path` and `cli` options; these are passed to the casper script through the `casperjs` Python executable.

Execution results:

```
$ casperjs test.js arg1 arg2 arg3 --foo=bar --plop anotherarg
Casper CLI passed args: [
  "arg1",
  "arg2",
  "arg3",
  "anotherarg"
]
Casper CLI passed options: {
  "casper-path": "/Users/niko/Sites/casperjs",
  "cli": true,
```

```
"foo": "bar",
"plop": true
}
```

Getting, checking or dropping parameters:

```
var casper = require("casper").create();
casper.echo(casper.cli.has(0));
casper.echo(casper.cli.get(0));
casper.echo(casper.cli.has(3));
casper.echo(casper.cli.get(3));
casper.echo(casper.cli.has("foo"));
casper.echo(casper.cli.get("foo"));
casper.cli.drop("foo");
casper.echo(casper.cli.has("foo"));
casper.echo(casper.cli.get("foo"));
casper.exit();
```

Execution results:

```
$ casperjs test.js arg1 arg2 arg3 --foo=bar --plop anotherarg
true
arg1
true
anotherarg
true
bar
false
undefined
```

Hint: You may need to wrap an option containing a space with escaped double quotes in Windows. `--foo="\space bar"`

Hint: What if you want to check if any arg or option has been passed to your script? Here you go:

```
// removing default options passed by the Python executable
casper.cli.drop("cli");
casper.cli.drop("casper-path");

if (casper.cli.args.length === 0 && Object.keys(casper.cli.options).length === 0) {
  casper.echo("No arg nor option passed").exit();
}
```

casperjs native options

New in version 1.1.

The *casperjs* command has three available options:

- `--direct`: to print out log messages to the console
- `--log-level=[debug|info|warning|error]` to set the *logging level*

- `--engine=[phantomjs|slimerjs]` to select the browser engine you want to use. CasperJS supports PhantomJS (default) that runs Webkit, and SlimerJS that runs Gecko.

Warning: Deprecated since version 1.1.

The `--direct` option has been renamed to `--verbose`. Although `--direct` will still work, it is now considered deprecated.

Example:

```
$ casperjs --verbose --log-level=debug myscript.js
```

Last but not least, you can still use all PhantomJS standard CLI options as you would do with any other PhantomJS script:

```
$ casperjs --web-security=no --cookies-file=/tmp/mycookies.txt myscript.js
```

Hint: To remember what the native PhantomJS cli options are available, run the `phantomjs --help` command. SlimerJS supports almost same options as PhantomJS.

Raw parameter values

New in version 1.0.

By default, the cli object will process every passed argument & cast them to the appropriate detected type; example script:

```
var casper = require('casper').create();
var utils = require('utils');

utils.dump(casper.cli.get('foo'));

casper.exit();
```

If you run this script:

```
$ casperjs c.js --foo=01234567
1234567
```

As you can see, the 01234567 value has been cast to a *Number*.

If you want the original string, use the `raw` property of the `cli` object, which contains the raw values of the passed parameters:

```
var casper = require('casper').create();
var utils = require('utils');

utils.dump(casper.cli.get('foo'));
utils.dump(casper.cli.raw.get('foo'));

casper.exit();
```

Sample usage:

```
$ casperjs c.js --foo=01234567
1234567
"01234567"
```

CHAPTER 4

Selectors

CasperJS makes a heavy use of selectors in order to work with the [DOM](#), and can transparently use either [CSS3](#) or [XPath](#) expressions.

All the examples below are based on this HTML code:

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>My page</title>
</head>
<body>
  <h1 class="page-title">Hello</h1>
  <ul>
    <li>one</li>
    <li>two</li>
    <li>three</li>
  </ul>
  <footer><p>©2012 myself</p></footer>
</body>
</html>
```

CSS3

By default, CasperJS accepts [CSS3 selector strings](#) to check for elements within the DOM.

To check if the `<h1 class="page-title">` element exists in the example page, you can use:

```
var casper = require('casper').create();

casper.start('http://domain.tld/page.html', function() {
  if (this.exists('h1.page-title')) {
    this.echo('the heading exists');
  }
});
```

```
    }  
  });  
  
  casper.run();
```

Or if you're using the *testing framework*:

```
casper.test.begin('The heading exists', 1, function suite(test) {  
  casper.start('http://domain.tld/page.html', function() {  
    test.assertExists('h1.page-title');  
  }).run(function() {  
    test.done();  
  });  
});
```

Some other convenient testing methods are relying on selectors:

```
casper.test.begin('Page content tests', 3, function suite(test) {  
  casper.start('http://domain.tld/page.html', function() {  
    test.assertExists('h1.page-title');  
    test.assertSelectorHasText('h1.page-title', 'Hello');  
    test.assertVisible('footer');  
  }).run(function() {  
    test.done();  
  });  
});
```

XPath

New in version 0.6.8.

You can alternatively use **XPath** expressions instead:

```
casper.start('http://domain.tld/page.html', function() {  
  this.test.assertExists({  
    type: 'xpath',  
    path: '//*[class="plop"]'  
  }, 'the element exists');  
});
```

To ease the use and reading of XPath expressions, a `selectXPath` helper is available from the `casper` module:

```
var x = require('casper').selectXPath;  
  
casper.start('http://domain.tld/page.html', function() {  
  this.test.assertExists(x('//*[id="plop"]'), 'the element exists');  
});
```

Warning: The only limitation of XPath use in CasperJS is in the *casper.fill()* method when you want to fill **file fields**; PhantomJS natively only allows the use of CSS3 selectors in its *uploadFile* method, hence this limitation.

CasperJS ships with its own *testing framework*, providing a handful set of tools to ease testing your webapps.

Warning: Changed in version 1.1.

The testing framework — hence its whole API — can only be used when using the `casperjs test` subcommand:

- If you try to use the `casper.test` property out of the testing environment, you'll get an error;
- As of 1.1-beta3, you can't override the preconfigured `casper` instance in this test environment. You can read more about the whys in the *dedicated FAQ entry*.

Unit testing

Imagine a dumb `Cow` object we want to unit test:

```
function Cow() {  
  this.mowed = false;  
  this.moo = function moo() {  
    this.mowed = true; // mootable state: don't do that at home  
    return 'moo!';  
  };  
}
```

Let's write a tiny test suite for it:

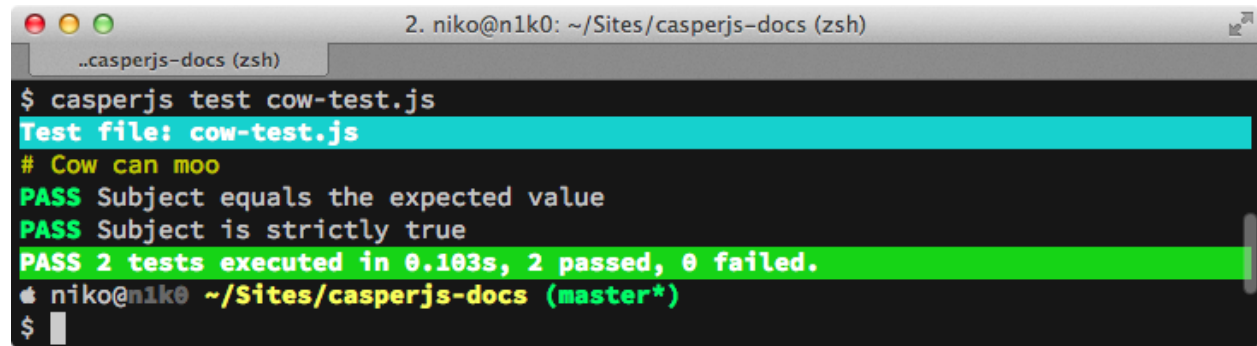
```
// cow-test.js  
casper.test.begin('Cow can moo', 2, function suite(test) {  
  var cow = new Cow();  
  test.assertEquals(cow.moo(), 'moo!');  
  test.assert(cow.mowed);  
});
```

```
test.done();
});
```

Run the tests using the `casperjs test` command:

```
$ casperjs test cow-test.js
```

You should theoretically get something like this:

A terminal window titled '2. niko@n1k0: ~/Sites/casperjs-docs (zsh)' shows the command '\$ casperjs test cow-test.js' being executed. The output is as follows:
Test file: cow-test.js
Cow can moo
PASS Subject equals the expected value
PASS Subject is strictly true
PASS 2 tests executed in 0.103s, 2 passed, 0 failed.
The prompt then changes to 'niko@n1k0 ~/Sites/casperjs-docs (master*)'.

Make it fail:

```
casper.test.begin('Cow can moo', 2, function suite(test) {
  var cow = new Cow();
  test.assertEquals(cow.moo(), 'BAZINGA!');
  test.assert(cow.mowed);
  test.done();
});
```

You'll get this instead:

Hint: The whole `tester` module API is documented [here](#).

Browser tests

Now let's write a suite for testing google search (yes, you read it well):

```
// googletesting.js
casper.test.begin('Google search retrieves 10 or more results', 5, function
↪suite(test) {
  casper.start("http://www.google.fr/", function() {
    test.assertTitle("Google", "google homepage title is the one expected");
    test.assertExists('form[action="/search"]', "main form is found");
    this.fill('form[action="/search"]', {
      q: "casperjs"
    }, true);
  });

  casper.then(function() {
    test.assertTitle("casperjs - Recherche Google", "google title is ok");
    test.assertUrlMatch(/q=casperjs/, "search term has been submitted");
  });
});
```



```

2. niko@n1k0: ~/Sites/casperjs-docs (zsh)
..casperjs-docs (zsh)
niko@n1k0 ~/Sites/casperjs-docs (master*)
$ casperjs test cow-test.js
Test file: cow-test.js
# Cow can moo
FAIL Subject equals the expected value
#   type: assertEquals
#   file: cow-test.js:11
#   code: test.assertEquals(cow.moo(), 'BAZINGA!');
#   subject: "moo!"
#   expected: "BAZINGA!"
⚠ Cow can moo: 2 tests planned, 1 tests executed
FAIL 1 tests executed in 0.106s, 0 passed, 1 failed.

Details for the 1 failed test:

In cow-test.js:11
  Cow can moo
    assertEquals: Subject equals the expected value
niko@n1k0 ~/Sites/casperjs-docs (master*)
$

```

```

test.assertEval(function() {
    return __utils__.findAll("h3.r").length >= 10;
}, "google search for \"casperjs\" retrieves 10 or more results");
});

casper.run(function() {
    test.done();
});
});

```

Now run the tests suite:

```
$ casperjs test googletesting.js
```

You'll probably get something like this:

```

2. niko@n1k0: ~/Sites/casperjs-docs (zsh)
..casperjs-docs (zsh)
$ casperjs test googletesting.js
Test file: googletesting.js
# Google search retrieves 10 or more results
PASS google homepage title is the one expected
PASS main form is found
PASS google title is ok
PASS search term has been submitted
PASS google search for "casperjs" retrieves 10 or more results
PASS 5 tests executed in 1.249s, 5 passed, 0 failed.
niko@n1k0 ~/Sites/casperjs-docs (master*)
$

```

Setting Casper options in the test environment

As you must use a preconfigured `casper` instance within the test environment, updating its *options* can be achieved this way:

```
casper.options.optionName = optionValue; // where optionName is obviously the desired_
↪option name

casper.options.clientScripts.push("new-script.js");
```

Advanced techniques

The `Tester#begin()` accepts either a function or an object to describe a suite; the object option allows to set up `setUp()` and `tearDown()` functions:

```
// cow-test.js
casper.test.begin('Cow can moo', 2, {
  setUp: function(test) {
    this.cow = new Cow();
  },

  tearDown: function(test) {
    this.cow.destroy();
  },

  test: function(test) {
    test.assertEquals(this.cow.moo(), 'moo!');
    test.assert(this.cow.mowed);
    test.done();
  }
});
```

Test command args and options

Arguments

The `casperjs test` command will treat every passed argument as file or directory paths containing tests. It will recursively scan any passed directory to search for `*.js` or `*.coffee` files and add them to the stack.

Warning: There are two important conditions when writing tests:

- You **must not** create a new Casper instance in a test file;
- You **must** call `Tester.done()` when all the tests contained in a suite (or in a file) have been executed.

Options

Options are prefixed with a double-dash (`--`):

- `--xunit=<filename>` will export test suite results in a *XUnit XML file*

- `--direct` or `--verbose` will print *log messages* directly to the console
- `--log-level=<logLevel>` sets the logging level (see the *related section*)
- `--auto-exit=no` prevents the test runner to exit when all the tests have been executed; this usually allows performing supplementary operations, though implies to exit casper manually listening to the `exit` tester event:

```
// $ casperjs test --auto-exit=no
casper.test.on("exit", function() {
  someTediousAsyncProcess(function() {
    casper.exit();
  });
});
```

New in version 1.0.

- `--includes=foo.js,bar.js` will include the `foo.js` and `bar.js` files before each test file execution.
- `--pre=pre-test.js` will add the tests contained in `pre-test.js` **before** executing the whole test suite.
- `--post=post-test.js` will add the tests contained in `post-test.js` **after** having executed the whole test suite.
- `--fail-fast` will terminate the current test suite as soon as a first failure is encountered.
- `--concise` will create a more concise output of the test suite.
- `--no-colors` will create an output without (beautiful) colors from casperjs.

Sample custom command:

```
$ casperjs test --includes=foo.js,bar.js \
  --pre=pre-test.js \
  --post=post-test.js \
  --direct \
  --log-level=debug \
  --fail-fast \
  test1.js test2.js /path/to/some/test/dir
```

Warning: Deprecated since version 1.1.

`--direct` option has been renamed to `--verbose`, though `--direct` will still works, while is to be considered deprecated.

Hint: A [demo gist](#) is also available in order to get you started with a sample suite involving some of these options.

Exporting results in XUnit format

CasperJS can export the results of the test suite to an XUnit XML file, which is compatible with continuous integration tools such as [Jenkins](#). To save the XUnit log of your test suite, use the `--xunit` option:

```
$ casperjs test googletesting.js --xunit=log.xml
```

You should get a pretty XUnit XML report like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites duration="1.249">
  <testsuite errors="0" failures="0" name="Google search retrieves 10 or more
↳ results" package="googletesting" tests="5" time="1.249" timestamp="2012-12-
↳ 30T21:27:26.320Z">
    <testcase classname="googletesting" name="google homepage title is the one
↳ expected" time="0.813"/>
    <testcase classname="googletesting" name="main form is found" time="0.002"/>
    <testcase classname="googletesting" name="google title is ok" time="0.416"/>
    <testcase classname="googletesting" name="search term has been submitted"
↳ time="0.017"/>
    <testcase classname="googletesting" name="google search for &quot;casperjs&
↳ quot; retrieves 10 or more results" time="0.001"/>
    <system-out/>
  </testsuite>
</testsuites>
```

You can customize the value for the *name* property by passing an object to *casper.test.fail()* like:

```
casper.test.fail('google search for "casperjs" retrieves 10 or more results', {name:
↳ 'result count is 10+'});
```

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites duration="1.249">
  <testsuite errors="0" failures="0" name="Google search retrieves 10 or more
↳ results" package="googletesting" tests="5" time="1.249" timestamp="2012-12-
↳ 30T21:27:26.320Z">
    <testcase classname="googletesting" name="google homepage title is the one
↳ expected" time="0.813"/>
    <testcase classname="googletesting" name="main form is found" time="0.002"/>
    <testcase classname="googletesting" name="google title is ok" time="0.416"/>
    <testcase classname="googletesting" name="search term has been submitted"
↳ time="0.017"/>
    <testcase classname="googletesting" name="results count is 10+" time="0.001"/>
      <failure type="fail">google search for "casperjs" retrieves 10 or more
↳ results</failure>
    <system-out/>
  </testsuite>
</testsuites>
```

CasperJS own tests

CasperJS has its own unit and functional test suite, located in the `tests` subfolder. To run this test suite:

```
$ casperjs selftest
```

Note: Running this test suite is a great way to find any bug on your platform. If it fails, feel free to [file an issue](#) or to ask on the [CasperJS mailing-list](#).

Extending Casper for Testing

This command:

```
$ casperjs test [path]
```

is just a shortcut for this one:

```
$ casperjs /path/to/casperjs/tests/run.js [path]
```

So if you want to extend Casper capabilities for your tests, your best bet is to write your own runner and extend the casper object instance from there.

Hint: You can find the default runner code in [run.js](#).

Here you'll find a quite complete reference of the CasperJS API. If something is erroneous or missing, please [file an issue](#).

The casper module

The Casper class

The easiest way to get a casper instance is to use the module's `create()` method:

```
var casper = require('casper').create();
```

But you can also retrieve the main Function and instantiate it by yourself:

```
var casper = new require('casper').Casper();
```

Hint: Also, check out *[how to extend Casper with your own methods](#)*.

Both the `Casper` constructor and the `create()` function accept a single `options` argument which is a standard javascript object:

```
var casper = require('casper').create({  
  verbose: true,  
  logLevel: "debug"  
});
```

Casper.options

An `options` object can be passed to the `Casper` constructor, eg.:

```
var casper = require('casper').create({
  clientScripts: [
    'includes/jquery.js',      // These two scripts will be injected in remote
    'includes/underscore.js'   // DOM on every request
  ],
  pageSettings: {
    loadImages: false,        // The WebPage instance used by Casper will
    loadPlugins: false        // use these settings
  },
  logLevel: "info",           // Only "info" level messages will be logged
  verbose: true                // log messages will be printed out to the console
});
```

You can also alter options at runtime:

```
var casper = require('casper').create();
casper.options.waitForTimeout = 1000;
```

The whole list of available options is detailed below.

clientScripts

Type: Array

Default: []

A collection of script filepaths to include in every page loaded.

exitOnError

Type: Boolean

Default: true

Sets if CasperJS must exit when an uncaught error has been thrown by the script.

httpStatusHandlers

Type: Object

Default: {}

A javascript Object containing functions to call when a requested resource has a given HTTP status code. A dedicated sample is provided as an example.

logLevel

Type: String

Default: "error"

Logging level (see the logging section for more information)

onAlert**Type:** Function**Default:** null**Signature:** onAlert(Object Casper, String message)

A function to be called when a javascript alert() is triggered

onDie**Type:** Function**Default:** null**Signature:** onDie(Object Casper, String message, String status)

A function to be called when Casper#die() is called

onError**Type:** Function**Default:** null**Signature:** onError(Object Casper, String msg, Array backtrace)

A function to be called when an “error” level event occurs

onLoadError**Type:** Function**Default:** null**Signature:** onLoadError(Object Casper, String casper.requestUrl, String status)

A function to be called when a requested resource cannot be loaded

onPageInitialized**Type:** Function**Default:** null**Signature:** onPageInitialized(Object page)

A function to be called after WebPage instance has been initialized

onResourceReceived**Type:** Function**Default:** null**Signature:** onResourceReceived(Object Casper, Object resource)

Proxy method for PhantomJS’ WebPage#onResourceReceived() callback, but the current Casper instance is passed as first argument.

`onResourceRequested`

Type: Function

Default: null

Signature: `onResourceRequested(Object Casper, Object resource)`

Proxy method for PhantomJS' `WebPage#onResourceRequested()` callback, but the current Casper instance is passed as first argument.

`onStepComplete`

Type: Function

Default: null

Signature: `onStepComplete(Object Casper, stepResult)`

A function to be executed when a step function execution is finished.

`onStepTimeout`

Type: Function

Default: Function

Signature: `onStepTimeout(Integer timeout, Integer stepNum)`

A function to be executed when a step function execution time exceeds the value of the `stepTimeout` option, if any has been set.

By default, on timeout the script will exit displaying an error, except in test environment where it will just add a failure to the suite results.

`onTimeout`

Type: Function

Default: Function

Signature: `onTimeout(Integer timeout)`

A function to be executed when script execution time exceeds the value of the `timeout` option, if any has been set.

By default, on timeout the script will exit displaying an error, except in test environment where it will just add a failure to the suite results.

`onWaitTimeout`

Type: Function

Default: Function

Signature: `onWaitTimeout(Integer timeout)`

A function to be executed when a `waitFor*` function execution time exceeds the value of the `waitTimeout` option, if any has been set.

By default, on timeout the script will exit displaying an error, except in test environment where it will just add a failure to the suite results.

page

Type: `WebPage`

Default: `null`

An existing PhantomJS `WebPage` instance

Warning: Overriding the `page` properties can cause some of the casper features **may not work**. For example, overriding the `onUrlChanged` property will cause the `waitForUrl` feature not work.

pageSettings

Type: `Object`

Default: `{}`

PhantomJS's `WebPage` settings object. Available settings are:

- `javascriptEnabled` defines whether to execute the script in the page or not (default to `true`)
- `loadImages` defines whether to load the inlined images or not
- `localToRemoteUrlAccessEnabled` defines whether local resource (e.g. from file) can access remote URLs or not (default to `false`)
- `userAgent` defines the user agent sent to server when the web page requests resources
- `userName` sets the user name used for HTTP authentication
- `password` sets the password used for HTTP authentication
- `resourceTimeout` (in milli-secs) defines the timeout after which any resource requested will stop trying and proceed with other parts of the page. `onResourceTimeout` callback will be called on timeout. `undefined` (default value) means default gecko parameters.

PhantomJS specific settings :

- `XSSAuditingEnabled` defines whether load requests should be monitored for cross-site scripting attempts (default to `false`)
- `webSecurityEnabled` defines whether web security should be enabled or not (defaults to `true`)

SlimerJS specific settings :

- `allowMedia` `false` to deactivate the loading of media (audio / video). Default: `true`. (SlimerJS only)
- `maxAuthAttempts` indicate the maximum of attempts of HTTP authentication. (SlimerJS 0.9)
- `plainTextAllContent` `true` to indicate that `webpage.plainText` returns everything, even content of script elements, invisible elements etc.. Default: `false`.

`remoteScripts`

Type: `Array`

Default: `[]`

New in version 1.0.

A collection of remote script urls to include in every page loaded

`safeLogs`

Type: `Boolean`

Default: `true`

New in version 1.0.

When this option is set to true — which is the default, any password information entered in `<input type="password">` will be obfuscated in log messages. Set `safeLogs` to false to disclose passwords in plain text (not recommended).

`silentErrors`

Type: `Boolean`

Default: `false`

When this option is enabled, caught step errors are not thrown (though related events are still emitted). Mostly used internally in a testing context.

`stepTimeout`

Type: `Number`

Default: `null`

Max step timeout in milliseconds; when set, every defined step function will have to execute before this timeout value has been reached. You can define the `onStepTimeout()` callback to catch such a case. By default, the script will die() with an error message.

`timeout`

Type: `Number`

Default: `null`

Max timeout in milliseconds

`verbose`

Type: `Boolean`

Default: `false`

Realtime output of log messages

viewportSize**Type:** Object**Default:** null

Viewport size, eg. {width: 800, height: 600}

Note: PhantomJS ships with a default viewport of 400x300, and CasperJS won't override it by default.

retryTimeout**Type:** Number**Default:** 100Default delay between attempts, for `wait*` family functions.**waitTimeout****Type:** Number**Default:** 5000Default wait timeout, for `wait*` family functions.

Casper prototype

back()**Signature:** `back()`

Moves back a step in browser's history:

```
casper.start('http://foo.bar/1')
casper.thenOpen('http://foo.bar/2');
casper.thenOpen('http://foo.bar/3');
casper.back();
casper.run(function() {
  console.log(this.getCurrentUrl()); // 'http://foo.bar/2'
});
```

Also have a look at *forward()*.**base64encode()****Signature:** `base64encode(String url [, String method, Object data])`

Encodes a resource using the base64 algorithm synchronously using client-side XMLHttpRequest.

Note: We cannot use `window.btoa()` because it fails miserably in the version of WebKit shipping with PhantomJS.

Example: retrieving google logo image encoded in base64:

```
var base64logo = null;
casper.start('http://www.google.fr/', function() {
    base64logo = this.base64encode('http://www.google.fr/images/srpr/logo3w.png');
});

casper.run(function() {
    this.echo(base64logo).exit();
});
```

You can also perform an HTTP POST request to retrieve the contents to encode:

```
var base64contents = null;
casper.start('http://domain.tld/download.html', function() {
    base64contents = this.base64encode('http://domain.tld/', 'POST', {
        param1: 'foo',
        param2: 'bar'
    });
});

casper.run(function() {
    this.echo(base64contents).exit();
});
```

bypass()

Signature: `bypass(Numbr nb)`

New in version 1.1.

Bypasses a given number of defined navigation steps:

```
casper.start();
casper.then(function() {
    // This step will be executed
});
casper.then(function() {
    this.bypass(2);
});
casper.then(function() {
    // This test won't be executed
});
casper.then(function() {
    // Nor this one
});
casper.run();
```

click()

Signature: `click(String selector, [Number|String X, Number|String Y])`

Performs a click on the element matching the provided *selector expression*. The method tries two strategies sequentially:

1. trying to trigger a MouseEvent in Javascript

2. using native QtWebKit event if the previous attempt failed

Example:

```
casper.start('http://google.fr/');

casper.thenEvaluate(function(term) {
    document.querySelector('input[name="q"]').setAttribute('value', term);
    document.querySelector('form[name="f"]').submit();
}, 'CasperJS');

casper.then(function() {
    // Click on 1st result link
    this.click('h3.r a');
});

casper.then(function() {
    // Click on 1st result link
    this.click('h3.r a', 10, 10);
});

casper.then(function() {
    // Click on 1st result link
    this.click('h3.r a', "50%", "50%");
});

casper.then(function() {
    console.log('clicked ok, new location is ' + this.getCurrentUrl());
});

casper.run();
```

clickLabel()

Signature: clickLabel(String label[, String tag])

New in version 0.6.1.

Clicks on the first DOM element found containing `label` text. Optionally ensures that the element node name is `tag`:

```
// <a href="...">My link is beautiful</a>
casper.then(function() {
    this.clickLabel('My link is beautiful', 'a');
});

// <button type="submit">But my button is sexier</button>
casper.then(function() {
    this.clickLabel('But my button is sexier', 'button');
});
```

capture()

Signature: capture(String targetFilepath, [Object clipRect, Object imgOptions])

Proxy method for PhantomJS' `WebPage#render`. Adds a `clipRect` parameter for automatically setting page `clipRect` setting and reverts it back once done:

```
casper.start('http://www.google.fr/', function() {
  this.capture('google.png', {
    top: 100,
    left: 100,
    width: 500,
    height: 400
  });
});

casper.run();
```

New in version 1.1.

The `imgOptions` object allows to specify two options:

- `format` to set the image format manually, avoiding relying on the filename
- `quality` to set the image quality, from 1 to 100

Example:

```
casper.start('http://foo', function() {
  this.capture('foo', undefined, {
    format: 'jpg',
    quality: 75
  });
});
```

`captureBase64()`

Signature: `captureBase64(String format[, Mixed area])`

New in version 0.6.5.

Computes the [Base64](#) representation of a binary image capture of the current page, or an area within the page, in a given format.

Supported image formats are `bmp`, `jpg`, `jpeg`, `png`, `ppm`, `tiff`, `xbm` and `xpm`.

The `area` argument can be either of the following types:

- **String:** `area` is a CSS3 selector string, eg. `div#plop form[name="form"] input[type="submit"]`
- **clipRect:** `area` is a `clipRect` object, eg. `{ "top":0, "left":0, "width":320, "height":200 }`
- **Object:** `area` is a *selector object*, eg. an XPath selector

Example:

```
casper.start('http://google.com', function() {
  // selector capture
  console.log(this.captureBase64('png', '#lga'));
  // clipRect capture
  console.log(this.captureBase64('png', {
    top: 0,
    left: 0,
    width: 320,
    height: 200
  }));
  // whole page capture
```



```

    console.log(this.captureBase64('png'));
  });

  casper.run();

```

captureSelector()

Signature: captureSelector(String targetFile, String selector [, Object imgOptions])

Captures the page area containing the provided selector and saves it to targetFile:

```

casper.start('http://www.weather.com/', function() {
  this.captureSelector('weather.png', '#wx-main');
});

casper.run();

```

New in version 1.1.

The `imgOptions` object allows to specify two options:

- `format` to set the image format manually, avoiding relying on the target filename
- `quality` to set the image quality, from 1 to 100

clear()

Signature: clear()

New in version 0.6.5.

Clears the current page execution environment context. Useful to avoid having previously loaded DOM contents being still active.

Think of it as a way to stop javascript execution within the remote DOM environment:

```

casper.start('http://www.google.fr/', function() {
  this.clear(); // javascript execution in this page has been stopped
});

casper.then(function() {
  // ...
});

casper.run();

```

debugHTML()

Signature: debugHTML([String selector, Boolean outer])

Outputs the results of `getHTML()` directly to the console. It takes the same arguments as `getHTML()`.

debugPage()

Signature: debugPage()

Logs the textual contents of the current page directly to the standard output, for debugging purpose:

```
casper.start('http://www.google.fr/', function() {
    this.debugPage();
});

casper.run();
```

die()

Signature: die(String message[, int status])

Exits phantom with a logged error message and an optional exit status code:

```
casper.start('http://www.google.fr/', function() {
    this.die("Fail.", 1);
});

casper.run();
```

download()

Signature: download(String url, String target[, String method, Object data])

Saves a remote resource onto the filesystem. You can optionally set the HTTP method using the method argument, and pass request arguments through the data object (see [base64encode\(\)](#)):

```
casper.start('http://www.google.fr/', function() {
    var url = 'http://www.google.fr/intl/fr/about/corporate/company/';
    this.download(url, 'google_company.html');
});

casper.run(function() {
    this.echo('Done.').exit();
});
```

Note: If you have some troubles downloading files, try to [disable web security](#).

each()

Signature: each(Array array, Function fn)

Iterates over provided array items and execute a callback:

```
var links = [
    'http://google.com/',
    'http://yahoo.com/',
    'http://bing.com/'
];
```

```
casper.start().each(links, function(self, link) {
  self.thenOpen(link, function() {
    this.echo(this.getTitle());
  });
});

casper.run();
```

Hint: Have a look at the [googlematch.js](#) sample script for a concrete use case.

eachThen()

Signature: eachThen(Array array, Function then)

New in version 1.1.

Iterates over provided array items and adds a step to the stack with current data attached to it:

```
casper.start().eachThen([1, 2, 3], function(response) {
  this.echo(response.data);
}).run();
```

Here's an example for opening an array of urls:

```
var casper = require('casper').create();
var urls = ['http://google.com/', 'http://yahoo.com/'];

casper.start().eachThen(urls, function(response) {
  this.thenOpen(response.data, function(response) {
    console.log('Opened', response.url);
  });
});

casper.run();
```

Note: Current item will be stored in the `response.data` property.

echo()

Signature: echo(String message[, String style])

Prints something to stdout, optionally with some fancy color (see the [colorizer module](#) for more information):

```
casper.start('http://www.google.fr/', function() {
  this.echo('Page title is: ' + this.evaluate(function() {
    return document.title;
  })), 'INFO'); // Will be printed in green on the console
});

casper.run();
```

`evaluate()`

Signature: `evaluate(Function fn[, arg1[, arg2[, ...]]])`

Basically [PhantomJS' WebPage#evaluate](#) equivalent. Evaluates an expression in the current page DOM context:

```
casper.evaluate(function(username, password) {
  document.querySelector('#username').value = username;
  document.querySelector('#password').value = password;
  document.querySelector('#submit').click();
}, 'sheldon.cooper', 'b4z1ng4');
```

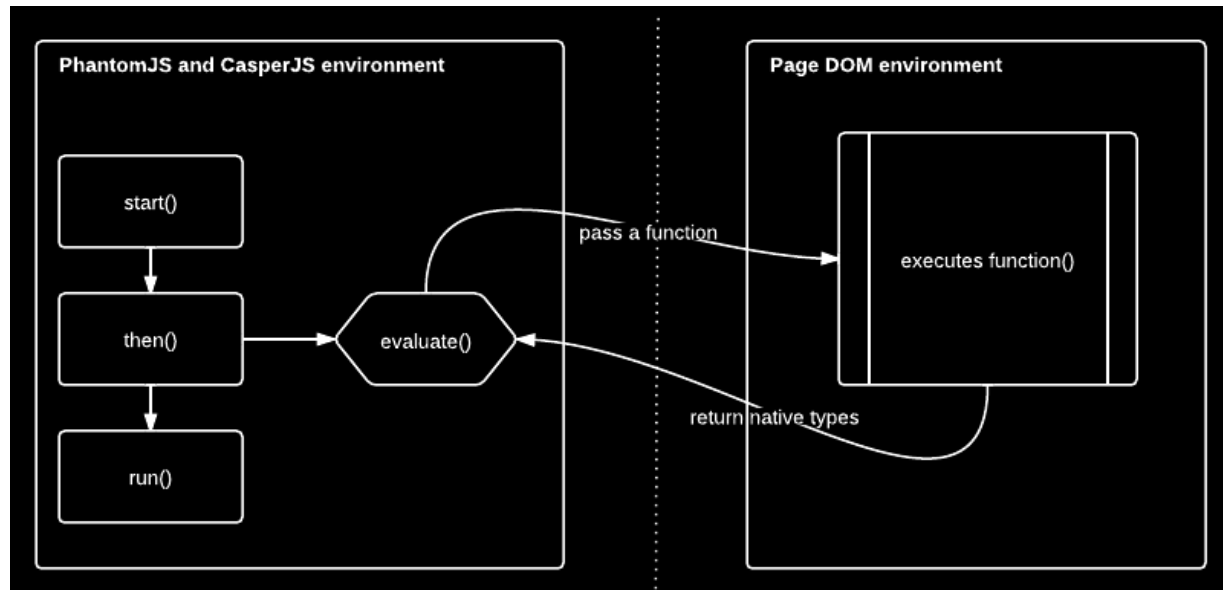
Note: For filling and submitting forms, rather use the [fill\(\)](#) method.

Warning: The pre-1.0 way of passing arguments using an object has been kept for BC purpose, though it may not work in some case; so you're encouraged to use the method described above.

Understanding `evaluate()`

The concept behind this method is probably the most difficult to understand when discovering CasperJS. As a reminder, think of the `evaluate()` method as a *gate* between the CasperJS environment and the one of the page you have opened; everytime you pass a closure to `evaluate()`, you're entering the page and execute code as if you were using the browser console.

Here's a quickly drafted diagram trying to basically explain the separation of concerns:



`evaluateOrDie()`

Signature: `evaluateOrDie(Function fn[, String message, int status])`

Evaluates an expression within the current page DOM and `die()` if it returns anything but `true`:

```
casper.start('http://foo.bar/home', function() {
  this.evaluateOrDie(function() {
    return /logged in/.match(document.title);
  }, 'not authenticated');
});

casper.run();
```

exit()**Signature:** exit([int status])

Exits PhantomJS with an optional exit status code.

Note: You can not rely on the fact that your script will be turned off immediately, because this method works asynchronously. It means that your script may continue to be executed after the call of this method. More info [here](#).

exists()**Signature:** exists(String selector)Checks if any element within remote DOM matches the provided *selector*:

```
casper.start('http://foo.bar/home', function() {
  if (this.exists('#my_super_id')) {
    this.echo('found #my_super_id', 'INFO');
  } else {
    this.echo('#my_super_id not found', 'ERROR');
  }
});

casper.run();
```

fetchText()**Signature:** fetchText(String selector)

Retrieves text contents matching a given *selector expression*. If you provide one matching more than one element, their textual contents will be concatenated:

```
casper.start('http://google.com/search?q=foo', function() {
  this.echo(this.fetchText('h3'));
}).run();
```

forward()**Signature:** forward()

Moves a step forward in browser's history:

```
casper.start('http://foo.bar/1')
casper.thenOpen('http://foo.bar/2');
casper.thenOpen('http://foo.bar/3');
casper.back(); // http://foo.bar/2
```

```
casper.back();    // http://foo.bar/1
casper.forward(); // http://foo.bar/2
casper.run();
```

Also have a look at *back()*.

log()

Signature: `log(String message[, String level, String space])`

Logs a message with an optional level in an optional space. Available levels are `debug`, `info`, `warning` and `error`. A space is a kind of namespace you can set for filtering your logs. By default, Casper logs messages in two distinct spaces: `phantom` and `remote`, to distinguish what happens in the PhantomJS environment from the remote one:

```
casper.start('http://www.google.fr/', function() {
    this.log("I'm logging an error", "error");
});

casper.run();
```

fill()

Signature: `fill(String selector, Object values[, Boolean submit])`

Fills the fields of a form with given values and optionally submits it. Fields are referenced by their `name` attribute.

Changed in version 1.1: To use *CSS3* or *XPath selectors* instead, check the *fillSelectors()* and *fillXPath()* methods.

Example with this sample html form:

```
<form action="/contact" id="contact-form" enctype="multipart/form-data">
  <input type="text" name="subject"/>
  <textarea name="content"></textarea>
  <input type="radio" name="civility" value="Mr"/> Mr
  <input type="radio" name="civility" value="Mrs"/> Mrs
  <input type="text" name="name"/>
  <input type="email" name="email"/>
  <input type="file" name="attachment"/>
  <input type="checkbox" name="cc"/> Receive a copy
  <input type="submit"/>
</form>
```

A script to fill and submit this form:

```
casper.start('http://some.tld/contact.form', function() {
    this.fill('form#contact-form', {
        'subject':    'I am watching you',
        'content':    'So be careful.',
        'civility':   'Mr',
        'name':       'Chuck Norris',
        'email':      'chuck@norris.com',
        'cc':         true,
        'attachment': '/Users/chuck/roundhousekick.doc'
    }, true);
});
```

```
casper.then(function() {
  this.evaluateOrDie(function() {
    return /message sent/.test(document.body.innerText);
  }, 'sending message failed');
});

casper.run(function() {
  this.echo('message sent').exit();
});
```

The `fill()` method supports single selects in the same way as text input. For multiple selects, supply an array of values to match against:

```
<form action="/contact" id="contact-form" enctype="multipart/form-data">
  <select multiple name="category">
    <option value="0">Friends</option>
    <option value="1">Family</option>
    <option value="2">Acquaintances</option>
    <option value="3">Colleagues</option>
  </select>
</form>
```

A script to select multiple options for category in this form:

```
casper.then(function() {
  this.fill('form#contact-form', {
    'categories': ['0', '1'] // Friends and Family
  });
});
```

Warning:

1. The `fill()` method currently can't fill **file fields using XPath selectors**; PhantomJS natively only allows the use of CSS3 selectors in its `uploadFile()` method, hence this limitation.
2. Please Don't use CasperJS nor PhantomJS to send spam, or I'll be calling the Chuck. More seriously, please just don't.

fillSelectors()

Signature: `fillSelectors(String selector, Object values[, Boolean submit])`

New in version 1.1.

Fills form fields with given values and optionally submits it. Fields are referenced by CSS3 selectors:

```
casper.start('http://some.tld/contact.form', function() {
  this.fillSelectors('form#contact-form', {
    'input[name="subject"]': 'I am watching you',
    'input[name="content"]': 'So be careful.',
    'input[name="civility"]': 'Mr',
    'input[name="name"]': 'Chuck Norris',
    'input[name="email"]': 'chuck@norris.com',
    'input[name="cc"]': true,
    'input[name="attachment"]': '/Users/chuck/roundhousekick.doc'
  });
});
```

```
    }, true);  
});
```

fillLabels()

Signature: fillLabels(String selector, Object values[, Boolean submit])

New in version 1.1.

Fills a form with provided field values using associated label text. Fields are referenced by label content values:

```
casper.start('http://some.tld/contact.form', function() {  
    this.fillLabels('form#contact-form', {  
        Email:      'chuck@norris.com',  
        Password:   'chuck',  
        Content:    'Am watching thou',  
        Check:      true,  
        No:         true,  
        Topic:      'bar',  
        Multitopic: ['bar', 'car'],  
        File:       fpath,  
        "1":        true,  
        "3":        true,  
        Strange:    "very"  
    }, true);  
});
```

fillXPath()

Signature: fillXPath(String selector, Object values[, Boolean submit])

New in version 1.1.

Fills form fields with given values and optionally submits it. While the `form` element is always referenced by a CSS3 selector, fields are referenced by XPath selectors:

```
casper.start('http://some.tld/contact.form', function() {  
    this.fillXPath('form#contact-form', {  
        '//input[@name="subject"]': 'I am watching you',  
        '//input[@name="content"]': 'So be careful.',  
        '//input[@name="civility"]': 'Mr',  
        '//input[@name="name"]': 'Chuck Norris',  
        '//input[@name="email"]': 'chuck@norris.com',  
        '//input[@name="cc"]': true,  
    }, true);  
});
```

Warning: The `fillXPath()` method currently can't fill **file fields using XPath selectors**; PhantomJS natively only allows the use of CSS3 selectors in its `uploadFile()` method, hence this limitation.

getCurrentUrl()

Signature: getCurrentUrl()

Retrieves current page URL. Note that the url will be url-decoded:

```
casper.start('http://www.google.fr/', function() {
  this.echo(this.getCurrentUrl()); // "http://www.google.fr/"
});

casper.run();
```

getElementAttribute()

Signature: getElementAttribute(String selector, String attribute)

New in version 1.0.

Retrieves the value of an attribute on the first element matching the provided *selector*:

```
var casper = require('casper').create();

casper.start('http://www.google.fr/', function() {
  require('utils').dump(this.getElementAttribute('div[title="Google"]', 'title')); /
  ↪ // "Google"
});

casper.run();
```

getElementsAttribute()

Signature: getElementsAttribute(String selector, String attribute)

New in version 1.1.

Retrieves the values of an attribute on each element matching the provided *selector*:

```
var casper = require('casper').create();

casper.start('http://www.google.fr/', function() {
  require('utils').dump(this.getElementsAttribute('div[title="Google"]', 'title')); ↵
  ↪ // ["Google"]
});

casper.run();
```

getElementBounds()

Signature: getElementBounds(String selector, Boolean page)

Retrieves boundaries for a DOM element matching the provided *selector*. If you have frames or/and iframes, set ‘true’ to the page parameter.

It returns an Object with four keys: top, left, width and height, or null if the selector doesn’t exist:

```
var casper = require('casper').create();

casper.start('http://www.google.fr/', function() {
  require('utils').dump(this.getElementBounds('div[title="Google"]'));
});
```

```
casper.run();
```

This will output something like:

```
{
  "height": 95,
  "left": 352,
  "top": 16,
  "width": 275
}
```

`getElementsBounds()`

Signature: `getElementsBounds(String selector)`

New in version 1.0.

Retrieves a list of boundaries for all DOM elements matching the provided *selector*.

It returns an array of objects with four keys: `top`, `left`, `width` and `height` (see *getElementBounds()*).

`getElementInfo()`

Signature: `getElementInfo(String selector)`

New in version 1.0.

Retrieves information about the first element matching the provided *selector*:

```
casper.start('http://google.fr/', function() {
  require('utils').dump(this.getElementInfo('#hplogo'));
});
```

Gives something like:

```
{
  "attributes": {
    "align": "left",
    "dir": "ltr",
    "id": "hplogo",
    "onload": "window.lol&&lol()",
    "style": "height:110px;width:276px;background:url(/images/srpr/logo1w.png) no-repeat",
    "title": "Google"
  },
  "height": 110,
  "html": "<div nowrap=\"nowrap\" style=\"color:#777;font-size:16px;font-weight:bold;position:relative;left:214px;top:70px\">France</div>",
  "nodeName": "div",
  "tag": "<div dir=\"ltr\" title=\"Google\" align=\"left\" id=\"hplogo\" onload=\"window.lol&&lol()\" style=\"height:110px;width:276px;background:url(/images/srpr/logo1w.png) no-repeat\"><div nowrap=\"nowrap\" style=\"color:#777;font-size:16px;font-weight:bold;position:relative;left:214px;top:70px\">France</div></div>",
  "text": "France\n",
  "visible": true,
```

```

    "width": 276,
    "x": 62,
    "y": 76
  }

```

Note: This method **does not** return a DOM element, only a simple object representation of it; this is because the casper environment has no direct access to the scraped page one.

getElementsInfo()

Signature: `getElementsInfo(String selector)`

New in version 1.1.

Retrieves information about all elements matching the provided *selector*:

```

casper.start('http://google.fr/', function() {
    require('utils').dump(this.getElementsInfo('#hplogo'));
});

```

Gives something like:

```

[
  {
    "attributes": {
      "align": "left",
      "dir": "ltr",
      "id": "hplogo",
      "onload": "window.lol&&lol()",
      "style": "height:110px;width:276px;background:url(/images/srpr/logo1w.
↪png) no-repeat",
      "title": "Google"
    },
    "height": 110,
    "html": "<div nowrap=\"nowrap\" style=\"color:#777;font-size:16px;font-
↪weight:bold;position:relative;left:214px;top:70px\">France</div>",
    "nodeName": "div",
    "tag": "<div dir=\"ltr\" title=\"Google\" align=\"left\" id=\"hplogo\"
↪onload=\"window.lol&&lol()\" style=\"height:110px;width:276px;
↪background:url(/images/srpr/logo1w.png) no-repeat\"><div nowrap=\"nowrap\" style=
↪\"color:#777;font-size:16px;font-weight:bold;position:relative;left:214px;top:70px\">
↪France</div></div>",
    "text": "France\n",
    "visible": true,
    "width": 276,
    "x": 62,
    "y": 76
  }
]

```

Note: This method **does not** return a NodeList, only a simple array of object representations of matching elements; this is because the casper environment has no direct access to the scraped page one.

getFormValues()

Signature: getFormValues(String selector)

New in version 1.0.

Retrieves a given form all of its field values:

```
casper.start('http://www.google.fr/', function() {
  this.fill('form', {q: 'plop'}, false);
  this.echo(this.getFormValues('form').q); // 'plop'
});

casper.run();
```

getGlobal()

Signature: getGlobal(String name)

Retrieves a global variable value within the remote DOM environment by its name. Basically, `getGlobal('foo')` will retrieve the value of `window.foo` from the page:

```
casper.start('http://www.google.fr/', function() {
  this.echo(this.getGlobal('innerWidth')); // 1024
});

casper.run();
```

getHTML()

Signature: getHTML([String selector, Boolean outer])

New in version 1.0.

Retrieves HTML code from the current page. By default, it outputs the whole page HTML contents:

```
casper.start('http://www.google.fr/', function() {
  this.echo(this.getHTML());
});

casper.run();
```

The `getHTML()` method can also dump HTML contents matching a given *selector*; for example with this HTML code:

```
<html>
  <body>
    <h1 id="foobar">Plop</h1>
  </body>
</html>
```

You can fetch those contents using:

```
casper.start('http://www.site.tld/', function() {
  this.echo(this.getHTML('h1#foobar')); // => 'Plop'
});
```

The `outer` argument allows to retrieve the outer HTML contents of the matching element:

```
casper.start('http://www.site.tld/', function() {
  this.echo(this.getHTML('h1#foobar', true)); // => '<h1 id="foobar">Plop</h1>'
});
```

getPageContent()

Signature: `getPageContent()`

New in version 1.0.

Retrieves current page contents, dealing with exotic other content types than HTML:

```
var casper = require('casper').create();

casper.start().then(function() {
  this.open('http://search.twitter.com/search.json?q=casperjs', {
    method: 'get',
    headers: {
      'Accept': 'application/json'
    }
  });
});

casper.run(function() {
  require('utils').dump(JSON.parse(this.getPageContent()));
  this.exit();
});
```

getTitle()

Signature: `getTitle()`

Retrieves current page title:

```
casper.start('http://www.google.fr/', function() {
  this.echo(this.getTitle()); // "Google"
});

casper.run();
```

mouseEvent()

Signature: `mouseEvent(String type, String selector, [Number|String X, Number|String Y])`

New in version 0.6.9.

Triggers a mouse event on the first element found matching the provided selector.

Supported events are `mouseup`, `mousedown`, `click`, `dblclick`, `mousemove`, `mouseover`, `mouseout` and for phantomjs `>= 1.9.8` `mouseenter`, `mouseleave` and `contextmenu`:

```
.. warning::
```

The list of supported events depends on the version of the engine in use. Older engines only provide partial support. For best support use recent builds of PhantomJS or SlimerJS.”

```
casper.start('http://www.google.fr/', function() { this.mouseEvent('click', 'h2 a', "20%", "50%");
});
casper.run();
```

newPage ()

Signature: newPage ()

New in version 1.1.

Only available since version 1.1.0.

Creates a new WebPage instance:

```
casper.start('http://google.com', function() {
    // ...
});

casper.then(function() {
    casper.page = casper.newPage();
    casper.open('http://yahoo.com').then(function() {
        // ....
    });
});

casper.run();
```

open ()

Signature: open(String location, Object Settings)

Performs an HTTP request for opening a given location. You can forge GET, POST, PUT, DELETE and HEAD requests.

Example for a standard GET request:

```
casper.start();

casper.open('http://www.google.com/').then(function() {
    this.echo('GOT it.');
```

```
});

casper.run();
```

Example for a POST request:

```
casper.start();

casper.open('http://some.testserver.com/post.php', {
    method: 'post',
    data: {
        'title': 'Plop',
        'body': 'Wow.'
    }
});
```

```
casper.then(function() {
    this.echo('POSTED it.');
```

```
});

casper.run();
```

To pass nested parameters arrays:

```
casper.open('http://some.testserver.com/post.php', {
    method: 'post',
    data: {
        'standard_param': 'foo',
        'nested_param[]': [           // please note the use of square brackets!
            'Something',
            'Something else'
        ]
    }
});
```

New in version 1.0.

To POST some data with utf-8 encoding:

```
casper.open('http://some.testserver.com/post.php', {
    method: 'post',
    headers: {
        'Content-Type': 'application/json; charset=utf-8'
    },
    encoding: 'utf8', // not enforced by default
    data: {
        'table_flip': '(\^^\ ',
    }
});
```

New in version 1.1.

You can also set custom request headers to send when performing an outgoing request, passing the `headers` option:

```
casper.open('http://some.testserver.com/post.php', {
    method: 'post',
    data: {
        'title': 'Plop',
        'body': 'Wow.'
    },
    headers: {
        'Accept-Language': 'fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3'
    }
});
```

reload()

Signature: `reload([Function then])`

New in version 1.0.

Reloads current page location:

```
casper.start('http://google.com', function() {
  this.echo("loaded");
  this.reload(function() {
    this.echo("loaded again");
  });
});

casper.run();
```

repeat()

Signature: repeat(int times, Function then)

Repeats a navigation step a given number of times:

```
casper.start().repeat(3, function() {
  this.echo("Badger");
});

casper.run();
```

resourceExists()

Signature: resourceExists(String|Function|RegExp test)

Checks if a resource has been loaded. You can pass either a function, a string or a RegExp instance to perform the test:

```
casper.start('http://www.google.com/', function() {
  if (this.resourceExists('logo3w.png')) {
    this.echo('Google logo loaded');
  } else {
    this.echo('Google logo not loaded', 'ERROR');
  }
});

casper.run();
```

Note: If you want to wait for a resource to be loaded, use the [waitForResource\(\)](#) method.

run()

Signature: run(fn onComplete[, int time])

Runs the whole suite of steps and optionally executes a callback when they've all been done. Obviously, **calling this method is mandatory** in order to run the Casper navigation suite.

Casper suite **won't run**:

```
casper.start('http://foo.bar/home', function() {
  // ...
});
```



```
// hey, it's missing .run() here!
```

Casper suite **will** run:

```
casper.start('http://foo.bar/home', function() {  
    // ...  
});  
  
casper.run();
```

`Casper.run()` also accepts an `onComplete` callback, which you can consider as a custom final step to perform when all the other steps have been executed. Just don't forget to `exit()` Casper if you define one!:

```
casper.start('http://foo.bar/home', function() {  
    // ...  
});  
  
casper.then(function() {  
    // ...  
});  
  
casper.run(function() {  
    this.echo('So the whole suite ended.');    this.exit(); // <--- don't forget me!  
});
```

Binding a callback to `complete.error` will trigger when the `onComplete` callback fails.

`scrollTo()`

Signature: `scrollTo(Number x, Number y)`

New in version 1.1-beta3.

Scrolls current document to the coordinates defined by the value of `x` and `y`:

```
casper.start('http://foo.bar/home', function() {  
    this.scrollTo(500, 300);  
});
```

Note: This operation is synchronous.

`scrollToBottom()`

Signature: `scrollToBottom()`

New in version 1.1-beta3.

Scrolls current document to its bottom:

```
casper.start('http://foo.bar/home', function() {  
    this.scrollToBottom();  
});
```

Note: This operation is synchronous.

sendKeys()

Signature: `sendKeys(Selector selector, String keys[, Object options])`

New in version 1.0.

Sends native keyboard events to the element matching the provided *selector*:

```
casper.then(function() {
  this.sendKeys('form.contact input#name', 'Duke');
  this.sendKeys('form.contact textarea#message', "Damn, I'm looking good.");
  this.click('form.contact input[type="submit"]');
});
```

New in version 1.1.

The currently supported HTML elements that can receive keyboard events from `sendKeys` are `<input>`, `<textarea>`, and any HTML element with attribute `contenteditable="true"`.

Options

- (Boolean) `reset`:

New in version 1.1-beta3.

When set to `true`, this option will first empty the current field value. By default, it's set to `false` and `sendKeys()` will just append string to the current field value.

- (Boolean) `keepFocus`:

`sendKeys()` by default will remove the focus on text input fields, which will typically close autocomplete widgets. If you want to maintain focus, use the `keepFocus` option. For example, if using jQuery-UI, you can click on the first autocomplete suggestion using:

```
casper.then(function() {
  this.sendKeys('form.contact input#name', 'action', {keepFocus: true});
  this.click('form.contact ul.ui-autocomplete li.ui-menu-item:first-child a');
});
```

- (String) `modifiers`:

`sendKeys()` accepts a `modifiers` option to support key modifiers. The option is a string representing the composition of modifiers to use, separated by the `+` character:

```
casper.then(function() {
  this.sendKeys('document', 's', {modifiers: 'ctrl+alt+shift'});
});
```

Available modifiers are:

- `ctrl`
- `alt`
- `shift`

- meta
- keypad

setHttpAuth()

Signature: setHttpAuth(String username, String password)

Sets HTTP_AUTH_USER and HTTP_AUTH_PW values for HTTP based authentication systems:

```
casper.start();

casper.setHttpAuth('sheldon.cooper', 'b4z1ng4');

casper.thenOpen('http://password-protected.domain.tld/', function() {
    this.echo("I'm in. Bazinga.");
})

casper.run();
```

Of course you can directly pass the auth string in the url to open:

```
var url = 'http://sheldon.cooper:b4z1ng4@password-protected.domain.tld/';

casper.start(url, function() {
    this.echo("I'm in. Bazinga.");
})

casper.run();
```

start()

Signature: start(String url[, Function then])

Configures and starts Casper, then opens the provided url and optionally adds the step provided by the then argument:

```
casper.start('http://google.fr/', function() {
    this.echo("I'm loaded.");
});

casper.run();
```

Alternatively:

```
casper.start('http://google.fr/');

casper.then(function() {
    this.echo("I'm loaded.");
});

casper.run();
```

Or alternatively:

```
casper.start('http://google.fr/');

casper.then(function() {
```

```
casper.echo("I'm loaded.");
});

casper.run();
```

Matter of taste!

Note: You must call the `start()` method in order to be able to add navigation steps and run the suite. If you don't you'll get an error message inviting you to do so anyway.

`status()`

Signature: `status(Boolean asString)`

New in version 1.0.

Returns the status of current Casper instance:

```
casper.start('http://google.fr/', function() {
    this.echo(this.status(true));
});

casper.run();
```

`switchToFrame()`

Signature: `switchToFrame(String|Number frameInfo)`

New in version 1.1.5.

Switches the main page to the frame having the name or frame index number matching the passed argument. Inject local scripts, remote scripts and client utils into this frame.

`switchToMainFrame()`

Signature: `switchToMainFrame()`

New in version 1.1.5.

Switch the main page to the parent frame of the currently active one.

`switchToParentFrame()`

Signature: `switchToParentFrame()`

New in version 1.1.5.

Switch the main page to the main frame.

then()**Signature:** then(Function then)

This method is the standard way to add a new navigation step to the stack, by providing a simple function:

```
casper.start('http://google.fr/');

casper.then(function() {
  this.echo("I'm in your google.");
});

casper.then(function() {
  this.echo('Now, let me write something');
});

casper.then(function() {
  this.echo('Oh well.');
});

casper.run();
```

You can add as many steps as you need. Note that the current Casper instance automatically binds the `this` keyword for you within step functions.

To run all the steps you defined, call the `run()` method, and voila.

Note: You must `start()` the casper instance in order to use the `then()` method.

Accessing the current HTTP response

New in version 1.0.

You can access the current HTTP response object using the first parameter of your step callback:

```
casper.start('http://www.google.fr/', function(response) {
  require('utils').dump(response);
});
```

That gives:

```
$ casperjs dump-headers.js
{
  "contentType": "text/html; charset=UTF-8",
  "headers": [
    {
      "name": "Date",
      "value": "Thu, 18 Oct 2012 08:17:29 GMT"
    },
    {
      "name": "Expires",
      "value": "-1"
    },
    // ... lots of other headers
  ],
  "id": 1,
  "redirectURL": null,
  "stage": "end",
  "status": 200,
  "statusText": "OK",
  "time": "2012-10-18T08:17:37.068Z",
  "url": "http://www.google.fr/"
}
```

So to fetch a particular header by its name:

```
casper.start('http://www.google.fr/', function(response) {  
    this.echo(response.headers.get('Date'));  
});
```

That gives:

```
$ casperjs dump-headers.js  
Thu, 18 Oct 2012 08:26:34 GMT
```

Warning: Step functions added to *then()* are processed in two different cases:

1. when the previous step function has been executed,
2. when the previous main HTTP request has been executed and the page *loaded*;

Note that there's no single definition of *page loaded*; is it when the DOMReady event has been triggered? Is it "all requests being finished"? Is it **all* application logic being performed"? Or "all elements being rendered"? The answer always depends on the context. Hence why you're encouraged to always use the *waitFor()* family methods to keep explicit control on what you actually expect.

A common trick is to use *waitForSelector()*:

```
casper.start('http://my.website.com/');  
  
casper.waitForSelector("#plop", function() {  
    this.echo("I'm sure #plop is available in the DOM");  
});  
  
casper.run();
```

thenBypass()

Signature: thenBypass(Number nb)

New in version 1.1.

Adds a navigation step which will bypass a given number of following steps:

```
casper.start('http://foo.bar/');  
casper.thenBypass(2);  
casper.then(function() {  
    // This test won't be executed  
});  
casper.then(function() {  
    // Nor this one  
});  
casper.then(function() {  
    // While this one will  
});  
casper.run();
```

thenBypassIf()**Signature:** thenBypassIf(Mixed condition, Number nb)

New in version 1.1.

Bypass a given number of navigation steps if the provided condition is truthy or is a function that returns a truthy value:

```
var universe = {
  answer: 42
};
casper.start('http://foo.bar/');
casper.thenBypassIf(function() {
  return universe && universe.answer === 42;
}, 2);
casper.then(function() {
  // This step won't be executed as universe.answer is 42
});
casper.then(function() {
  // Nor this one
});
casper.then(function() {
  // While this one will
});
casper.run();
```

thenBypassUnless()**Signature:** thenBypassUnless(Mixed condition, Number nb)

New in version 1.1.

Opposite of *thenBypassIf()*.**thenClick()****Signature:** thenClick(String selector[, Function then])

Adds a new navigation step to click a given selector and optionally add a new navigation step in a single operation:

```
// Click the first link in the casperJS page
casper.start('http://casperjs.org/').thenClick('a', function() {
  this.echo("I clicked on first link found, the page is now loaded.");
});

casper.run();
```

This method is basically a convenient a shortcut for chaining a *then()* and an *click()* calls.

thenEvaluate()**Signature:** thenEvaluate(Function fn[, arg1[, arg2[, ...]])

Adds a new navigation step to perform code evaluation within the current retrieved page DOM:

```
// Querying for "Chuck Norris" on Google
casper.start('http://google.fr/').thenEvaluate(function(term) {
    document.querySelector('input[name="q"]').setAttribute('value', term);
    document.querySelector('form[name="f"]').submit();
}, 'Chuck Norris');

casper.run();
```

This method is a convenient shortcut for chaining *then()* and *evaluate()* calls.

thenOpen()

Signature: `thenOpen(String location[, mixed options])`

Adds a new navigation step for opening a new location, and optionally add a next step when its loaded:

```
casper.start('http://google.fr/').then(function() {
    this.echo("I'm in your google.");
});

casper.thenOpen('http://yahoo.fr/', function() {
    this.echo("Now I'm in your yahoo.");
});

casper.run();
```

New in version 1.0.

You can also specify request settings by passing a setting object (see *open()*) as the second argument:

```
casper.start().thenOpen('http://url.to/some/uri', {
    method: "post",
    data: {
        username: 'chuck',
        password: 'n0rr15'
    }
}, function() {
    this.echo("POST request has been sent.");
});

casper.run();
```

thenOpenAndEvaluate()

Signature: `thenOpenAndEvaluate(String location[, Function then[, arg1[, arg2[, ...]]]])`

Basically a shortcut for opening an url and evaluate code against remote DOM environment:

```
casper.start('http://google.fr/').then(function() {
    this.echo("I'm in your google.");
});

casper.thenOpenAndEvaluate('http://yahoo.fr/', function() {
    var f = document.querySelector('form');
    f.querySelector('input[name=q]').value = 'chuck norris';
});
```



```
f.submit();
});

casper.run(function() {
  this.debugPage();
  this.exit();
});
```

toString()

Signature: toString()

New in version 1.0.

Returns a string representation of current Casper instance:

```
casper.start('http://google.fr/', function() {
  this.echo(this); // [object Casper], currently at http://google.fr/
});

casper.run();
```

unwait()

Signature: unwait()

New in version 1.1.

Abort all current waiting processes, if any.

userAgent()

Signature: userAgent(String agent)

New in version 1.0.

Sets the [User-Agent string](#) to send through headers when performing requests:

```
casper.start();

casper.userAgent('Mozilla/5.0 (Macintosh; Intel Mac OS X)');

casper.thenOpen('http://google.com/', function() {
  this.echo("I'm a Mac.");
  this.userAgent('Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)');
});

casper.thenOpen('http://google.com/', function() {
  this.echo("I'm a PC.");
});

casper.run();
```

viewport()

Signature: viewport(Number width, Number height[, Function then])

Changes current viewport size:

```
casper.viewport(1024, 768);
```

To be sure page reflowing has occurred, you have to use it asynchronously:

```
casper.viewport(1024, 768).then(function() {  
    // new view port is now effective  
});
```

New in version 1.1.

As of 1.1 you can pass a *then* step function directly to viewport():

```
casper.viewport(1024, 768, function() {  
    // new view port is effective  
});
```

Note: PhantomJS comes with a default viewport size of 400x300, and CasperJS doesn't override it by default.

visible()

Signature: visible(String selector)

Checks if the DOM element matching the provided *selector expression* is visible in remote page:

```
casper.start('http://google.com/', function() {  
    if (this.visible('#hplogo')) {  
        this.echo("I can see the logo");  
    } else {  
        this.echo("I can't see the logo");  
    }  
});
```

wait()

Signature: wait(Number timeout[, Function then])

Pause steps suite execution for a given amount of time, and optionally execute a step on done:

```
casper.start('http://yoursite.tld/', function() {  
    this.wait(1000, function() {  
        this.echo("I've waited for a second.");  
    });  
});  
  
casper.run();
```

You can also write the same thing like this:

```
casper.start('http://yoursite.tld/');

casper.wait(1000, function() {
    this.echo("I've waited for a second.");
});

casper.run();
```

waitFor()

Signature: waitFor(Function testFx[, Function then, Function onTimeout, Number timeout, Object details])

Waits until a function returns true to process any next step.

You can also set a callback on timeout using the `onTimeout` argument, and set the timeout using the `timeout` one, in milliseconds. The default timeout is set to 5000ms:

```
casper.start('http://yoursite.tld/');

casper.waitFor(function check() {
    return this.evaluate(function() {
        return document.querySelectorAll('ul.your-list li').length > 2;
    });
}, function then() {
    this.captureSelector('yoursitelist.png', 'ul.your-list');
});

casper.run();
```

Example using the `onTimeout` callback:

```
casper.start('http://yoursite.tld/');

casper.waitFor(function check() {
    return this.evaluate(function() {
        return document.querySelectorAll('ul.your-list li').length > 2;
    });
}, function then() {    // step to execute when check() is ok
    this.captureSelector('yoursitelist.png', 'ul.your-list');
}, function timeout() { // step to execute if check has failed
    this.echo("I can't haz my screenshot.").exit();
});

casper.run();
```

`details` is a property bag of various information that will be passed to the `waitFor.timeout` event, if it is emitted. This can be used for better error messages or to conditionally ignore some timeout events.

New in version 1.1.5.

As of 1.1.5 does a last run of check function after timeout if check function is still false.:

```
.. warning::
```

Please note, that all *waitFor* methods are not chainable. Consider wrapping each of them in a *casper.then* in order to achieve this functionality.

waitForAlert()

Signature: `waitForAlert(Function then[, Function onTimeout, Number timeout])`

New in version 1.1-beta4.

Waits until a [JavaScript alert](#) is triggered. The step function will be passed the alert message in the `response.data` property:

```
casper.waitForAlert(function(response) {  
    this.echo("Alert received: " + response.data);  
});
```

waitForExec()

Signature: `waitForExec(command, parameters[, Function then, Function onTimeout, timeout])`

New in version 1.1.5.

Waits until `command` runs with `parameters` and exits. The `command`, `parameters`, `pid`, `stdout`, `stderr`, `elapsedTime` and `exitCode` will be in the `response.data` property. `command` must be a string or `parameters` must be an array. `command` can be a string of an executable or a string of an executable and its arguments separated by spaces. If `command` is falsy or is not a string, system shell (environment variable `SHELL` or `ComSpec`) is used. The arguments separated by spaces are concatenated with the `parameters` array to be sent to executable. If `parameters` is falsy or is not an array, an empty array is used. `timeout` can be a number or an array of two numbers, the first is the timeout of `wait*` family functions and the second is the timeout between `TERM` and `KILL` signals on timeout. If not declared, it assumes the same value of the first element or the default timeout of `wait*` family functions.:

```
// merge captured PDFs with default system shell (bash on Linux) calling /usr/bin/gs,   
↳ and runs a small script to remove files  
casper.waitForExec(null, ['-c', '{ /usr/bin/gs -dPDFSETTINGS=/ebook -dBATCH -dNOPAUSE -  
↳ -q -sDEVICE=pdfwrite -sOutputFile=/my_merged_captures.pdf /my_captures*.pdf && /bin/  
↳ rm /my_captures*.pdf; } || { /bin/rm /my_merged_captures.pdf && exit 1; }'],  
    function(response) {  
        this.echo("Program finished by itself:" + JSON.stringify(response.data));  
    }, function(timeout, response) {  
        this.echo("Program finished by casper:" + JSON.stringify(response.data));  
    });  
  
// merge captured PDFs with bash calling /usr/bin/gs, and runs a small script to   
↳ remove files  
casper.waitForExec('/bin/bash -c', [{ /usr/bin/gs -dPDFSETTINGS=/ebook -dBATCH -  
↳ dNOPAUSE -q -sDEVICE=pdfwrite -sOutputFile=/my_merged_captures.pdf /my_captures*.  
↳ pdf && /bin/rm /my_captures*.pdf; } || { /bin/rm /my_merged_captures.pdf && exit 1;   
↳ }'],  
    function(response) {  
        this.echo("Program finished by itself:" + JSON.stringify(response.data));  
    }, function(timeout, response) {  
        this.echo("Program finished by casper:" + JSON.stringify(response.data));  
    });  
  
// merge captured PDFs calling /usr/bin/gs  
casper.waitForExec('/usr/bin/gs', ['-dPDFSETTINGS=/ebook', '-dBATCH', '-dNOPAUSE', '-q', '-  
↳ sDEVICE=pdfwrite', '-sOutputFile=/my_merged_captures_pdfs.pdf', '/my_captures_1.pdf',  
↳ '/my_captures_2.pdf', '/my_captures_3.pdf'],
```

```

    function(response) {
        this.echo("Program finished by itself:" + JSON.stringify(response.data));
    }, function(timeout, response) {
        this.echo("Program finished by casper:" + JSON.stringify(response.data));
    });

// merge captured PDFs calling /usr/bin/gs
casper.waitForExec('/usr/bin/gs -dPDFSETTINGS=/ebook -dBATC -dNOPAUSE -q -
↪sDEVICE=pdfwrite -sOutputFile=/my_merged_captures_pdfs.pdf /my_captures_1.pdf /my_
↪captures_2.pdf /my_captures_3.pdf', null,
    function(response) {
        this.echo("Program finished by itself:" + JSON.stringify(response.data));
    }, function(timeout, response) {
        this.echo("Program finished by casper:" + JSON.stringify(response.data));
    });

```

Warning: `waitForExec()` only kills the called program on timeout. If the called program calls other processes, they won't be killed when `waitForExec()` times out.

`waitForPopup()`

Signature: `waitForPopup(String|RegExp|Object urlPattern[, Function then, Function onTimeout, Number timeout])`

New in version 1.0.

Waits for a popup having its url matching the provided pattern to be opened and loaded.

The currently loaded popups are available in the `Casper.popups` array-like property:

```

casper.start('http://foo.bar/').then(function() {
    this.test.assertTitle('Main page title');
    this.clickLabel('Open me a popup');
});

// this will wait for the popup to be opened and loaded
casper.waitForPopup(/popup\.html$/, function() {
    this.test.assertEquals(this.popups.length, 1);
});

// this will wait for the first popup to be opened and loaded
casper.waitForPopup(0, function() {
    this.test.assertEquals(this.popups.length, 1);
});

// this will wait for the popup's named to be opened and loaded
casper.waitForPopup({windowName: "mainPopup"}, function() {
    this.test.assertEquals(this.popups.length, 1);
});

// this will wait for the popup's title to be opened and loaded
casper.waitForPopup({title: "Popup Title"}, function() {
    this.test.assertEquals(this.popups.length, 1);
});

```

```
// this will wait for the popup's url to be opened and loaded
casper.waitForPopup({url: 'http://foo.bar/'}, function() {
  this.test.assertEquals(this.poppers.length, 1);
});

// this will set the popup DOM as the main active one only for time the
// step closure being executed
casper.withPopup(/popup\.html$/, function() {
  this.test.assertTitle('Popup title');
});

// next step will automatically revert the current page to the initial one
casper.then(function() {
  this.test.assertTitle('Main page title');
});
```

waitForResource()

Signature: `waitForResource(String|Function|RegExp testFx[, Function then, Function onTimeout, Number timeout])`

Wait until a resource that matches a resource matching constraints defined by `testFx` are satisfied to process a next step.

The `testFx` argument can be either a string, a function or a `RegExp` instance:

```
casper.waitForResource("foobar.png", function() {
  this.echo('foobar.png has been loaded.');
```

Using a regexp:

```
casper.waitForResource(/foo(bar|baz)\.png$/, function() {
  this.echo('foobar.png or foobaz.png has been loaded.');
```

Using a function:

```
casper.waitForResource(function testResource(resource) {
  return resource.url.indexOf("https") === 0;
}, function onReceived() {
  this.echo('a secure resource has been loaded.');
```

waitForUrl()

Signature: `waitForUrl(String|RegExp url[, Function then, Function onTimeout, Number timeout])`

New in version 1.1.

Waits for the current page url to match the provided argument (String or `RegExp`):

```
casper.start('http://foo/').waitForUrl(/login\.html$/, function() {
  this.echo('redirected to login.html');
```

```
casper.run();
```

waitForSelector()

Signature: `waitForSelector(String selector[, Function then, Function onTimeout, Number timeout])`

Waits until an element matching the provided *selector expression* exists in remote DOM to process any next step. Uses *waitFor()*:

```
casper.start('https://twitter.com/#!/n1k0');

casper.waitForSelector('.tweet-row', function() {
    this.captureSelector('twitter.png', 'html');
});

casper.run();
```

waitWhileSelector()

Signature: `waitWhileSelector(String selector[, Function then, Function onTimeout, Number timeout])`

Waits until an element matching the provided *selector expression* does not exist in remote DOM to process a next step. Uses *waitFor()*:

```
casper.start('http://foo.bar/');

casper.waitWhileSelector('.selector', function() {
    this.echo('.selector is no more!');
});

casper.run();
```

waitForSelectorTextChange()

Signature: `waitForSelectorTextChange(String selectors[, Function then, Function onTimeout, Number timeout])`

Waits until the text on an element matching the provided *selector expression* is changed to a different value before processing the next step. Uses *waitFor()*:

```
casper.start('http://foo.bar/');

casper.waitForSelectorTextChange('.selector', function() {
    this.echo('The text on .selector has been changed.');
```

`waitForText()`

Signature: `waitForText(String text[, Function then, Function onTimeout, Number timeout])`

New in version 1.0.

Waits until the passed text is present in the page contents before processing the immediate next step. Uses `waitFor()`:

```
casper.start('http://why.univer.se/').waitForText("42", function() {
  this.echo('Found the answer.');
```

```
});

casper.run();
```

`waitUntilVisible()`

Signature: `waitUntilVisible(String selector[, Function then, Function onTimeout, Number timeout])`

Waits until an element matching the provided *selector expression* is visible in the remote DOM to process a next step. Uses `waitFor()`.

`waitWhileVisible()`

Signature: `waitWhileVisible(String selector[, Function then, Function onTimeout, Number timeout])`

Waits until an element matching the provided *selector expression* is no longer visible in remote DOM to process a next step. Uses `waitFor()`:

```
var casper = require('casper').create();

casper.start('https://www.example.com/').thenClick('html body div p a', function () {
  this.waitWhileVisible('body > div:nth-child(1) > p:nth-child(2)', function () {
    this.echo("The selected element existed in previous page but doesn't exist in ↵
    this page.");
  })
}).run();
```

`warn()`

Signature: `warn(String message)`

Logs and prints a warning message to the standard output:

```
casper.warn("I'm a warning message.");
```

Note: Calling `warn()` will trigger the `warn event`.

withFrame()**Signature:** withFrame(String|Number frameInfo, Function then)

New in version 1.0.

Switches the main page to the frame having the name or frame index number matching the passed argument, and processes a step.

The page context switch only lasts until the step execution is finished:

```
casper.start('tests/site/frames.html', function() {
    this.test.assertTitle('FRAMESET TITLE');
});

casper.withFrame('frame1', function() {
    this.test.assertTitle('FRAME TITLE');
});

casper.withFrame(0, function() {
    this.test.assertTitle('FRAME TITLE');
});

casper.then(function() {
    this.test.assertTitle('FRAMESET TITLE');
});
```

withPopup()**Signature:** withPopup(Mixed popupInfo, Function then)

New in version 1.0.

Switches the main page to a popup matching the information passed as argument, and processes a step. The page context switch only lasts until the step execution is finished:

```
casper.start('http://foo.bar/').then(function() {
    this.test.assertTitle('Main page title');
    this.clickLabel('Open me a popup');
});

// this will wait for the popup to be opened and loaded
casper.waitForPopup(/popup\.html$/, function() {
    this.test.assertEquals(this.pops.length, 1);
});

// this will set the popup DOM as the main active one only for time the
// step closure being executed
casper.withPopup(/popup\.html$/, function() {
    this.test.assertTitle('Popup title');
});

// this will set the popup DOM as the main active one only for time the
// step closure being executed
casper.withPopup(0, function() {
    this.test.assertTitle('Popup title');
});

// this will set the popup DOM as the main active one only for time the
```

```
// step closure being executed
casper.withPopup({windowName: "mainPopup", title: 'Popup title', url: 'http://foo.bar/'}
→, function() {
    this.test.assertTitle('Popup title');
});

// next step will automatically revert the current page to the initial one
casper.then(function() {
    this.test.assertTitle('Main page title');
});
```

Note: The currently loaded popups are available in the `Casper.popups` array-like property.

`withSelectorScope()`

Signature: `withSelectorScope(String selector, Function then)`

New in version 1.1.5.

Switches the main DOM scope to a specific scope the information passed as argument, and processes a step. The scope context switch only lasts until the step execution is finished:

```
.. index:: Zoom
```

`zoom()`

Signature: `zoom(Number factor)`

New in version 1.0.

Sets the current page zoom factor:

```
var casper = require('casper').create();

casper.start().zoom(2).thenOpen('http://google.com', function() {
    this.capture('big-google.png');
});

casper.run();
```

The `clientutils` module

Casper ships with a few client-side utilities which are injected in the remote DOM environment, and accessible from there through the `__utils__` object instance of the `ClientUtils` class from the `clientutils` module:

```
casper.evaluate(function() {
    __utils__.echo("Hello World!");
});
```

Note: These tools are provided to avoid coupling CasperJS to any third-party library like jQuery, Mootools or something; but you can always include these and have them available client-side using the *Casper.options.clientScripts* option.

Bookmarklet

A bookmarklet is also available to help injecting Casper's client-side utilities in the DOM of your favorite browser.

Just drag the following link onto your favorites toolbar; when clicking it, a `__utils__` object will be available within the console of your browser:

Note: CasperJS and PhantomJS being based on [Webkit](#), you're strongly encouraged to use a recent Webkit compatible browser to use this bookmarklet (Chrome, Safari, etc...)

ClientUtils prototype

`echo()`

Signature: `echo(String message)`

New in version 1.0.

Print a message out to the casper console from the remote page DOM environment:

```
casper.start('http://foo.ner/').thenEvaluate(function() {
  __utils__.echo('plop'); // this will be printed to your shell at runtime
});
```

`encode()`

Signature: `encode(String contents)`

Encodes a string using the [base64 algorithm](#). For the records, CasperJS doesn't use builtin `window.btoa()` function because it can't deal efficiently with strings encoded using >8b characters:

```
var base64;
casper.start('http://foo.bar/', function() {
  base64 = this.evaluate(function() {
    return __utils__.encode("I've been a bit cryptic recently");
  });
});

casper.run(function() {
  this.echo(base64).exit();
});
```

`exists()`

Signature: `exists(String selector)`

Checks if a DOM element matching a given *selector expression* exists:

```
var exists;
casper.start('http://foo.bar/', function() {
  exists = this.evaluate(function() {
    return __utils__.exists('#some_id');
  });
});

casper.run(function() {
  this.echo(exists).exit();
});
```

findAll()

Signature: findAll(String selector)

Retrieves all DOM elements matching a given *selector expression*:

```
var links;
casper.start('http://foo.bar/', function() {
  links = this.evaluate(function() {
    var elements = __utils__.findAll('a.menu');
    return elements.map(function(e) {
      return e.getAttribute('href');
    });
  });
});

casper.run(function() {
  this.echo(JSON.stringify(links)).exit();
});
```

findOne()

Signature: findOne(String selector)

Retrieves a single DOM element by a *selector expression*:

```
var href;
casper.start('http://foo.bar/', function() {
  href = this.evaluate(function() {
    return __utils__.findOne('#my_id').getAttribute('href');
  });
});

casper.run(function() {
  this.echo(href).exit();
});
```

forceTarget()

Signature: forceTarget(String selector, String target)

Force the engine to use another target instead of the one provided. Very useful to limit the number of open windows and reduce memory consumption:

```
casper.start('http://foo.bar/', function() {
    var href = this.evaluate(function() {
        return __utils__.forceTarget('#my_id', '_self').click();
    });
    this.echo(href);
});

casper.run(function() {
    this.exit();
});
```

getBase64()

Signature: getBase64(String url[, String method, Object data])

This method will retrieve a base64 encoded version of any resource behind a url. For example, let's imagine we want to retrieve the base64 representation of some website's logo:

```
var logo = null;
casper.start('http://foo.bar/', function() {
    logo = this.evaluate(function() {
        var imgUrl = document.querySelector('img.logo').getAttribute('src');
        return __utils__.getBase64(imgUrl);
    });
});

casper.run(function() {
    this.echo(logo).exit();
});
```

getBinary()

Signature: getBinary(String url[, String method, Object data])

This method will retrieve the raw contents of a given binary resource; unfortunately though, PhantomJS cannot process these data directly so you'll have to process them within the remote DOM environment. If you intend to download the resource, use [getBase64\(\)](#) or [Casper.base64encode\(\)](#) instead:

```
casper.start('http://foo.bar/', function() {
    this.evaluate(function() {
        var imgUrl = document.querySelector('img.logo').getAttribute('src');
        console.log(__utils__.getBinary(imgUrl));
    });
});

casper.run();
```

getDocumentHeight()

Signature: getDocumentHeight()

New in version 1.0.

Retrieves current document height:

```
var documentHeight;

casper.start('http://google.com/', function() {
  documentHeight = this.evaluate(function() {
    return __utils__.getDocumentHeight();
  });
  this.echo('Document height is ' + documentHeight + 'px');
});

casper.run();
```

getDocumentWidth()

Signature: getDocumentWidth()

New in version 1.0.

Retrieves current document width:

```
var documentHeight;

casper.start('http://google.com/', function() {
  documentWidth = this.evaluate(function() {
    return __utils__.getDocumentWidth();
  });
  this.echo('Document width is ' + documentWidth + 'px');
});

casper.run();
```

getElementBounds()

Signature: getElementBounds(String selector)

Retrieves boundaries for a DOM elements matching the provided *selector*.

It returns an Object with four keys: top, left, width and height, or null if the selector doesn't exist.

getElementsBounds()

Signature: getElementsBounds(String selector)

Retrieves boundaries for all DOM element matching the provided *selector*.

It returns an array of objects each having four keys: top, left, width and height.

getElementByXPath()

Signature: getElementByXPath(String expression [, HTMLElement scope])

Retrieves a single DOM element matching a given *XPath expression*.

New in version 1.0.

The *scope* argument allow to set the context for executing the XPath query:

```
// will be performed against the whole document
__utils__.getElementByXPath('//*[a]');

// will be performed against a given DOM element
__utils__.getElementByXPath('//*[a]', __utils__.findOne('div.main'));
```

getElementsByXPath()

Signature: `getElementsByXPath(String expression [, HTMLElement scope])`

Retrieves all DOM elements matching a given *XPath expression*, if any.

New in version 1.0.

The scope argument allows to set the context for executing the XPath query.

getFieldValue()

Signature: `getFieldValue(String selector[, HTMLElement scope])`

New in version 1.0.

Retrieves the value from the field named against the `inputNamed` argument:

```
<form>
  <input type="text" name="plop" value="42">
</form>
```

Using the `getFieldValue()` method for plop:

```
__utils__.getFieldValue('[name="plop"]'); // 42
```

Options:

getFormValues()

Signature: `getFormValues(String selector)`

New in version 1.0.

Retrieves a given form and all of its field values:

```
<form id="login" action="/login">
  <input type="text" name="username" value="foo">
  <input type="text" name="password" value="bar">
  <input type="submit">
</form>
```

To get the form values:

```
__utils__.getFormValues('form#login'); // {username: 'foo', password: 'bar'}
```

log()

Signature: log(String message[, String level])

Logs a message with an optional level. Will format the message a way CasperJS will be able to log phantomjs side. Default level is debug:

```
casper.start('http://foo.ner/').thenEvaluate(function() {
  __utils__.log("We've got a problem on client side", 'error');
});
```

makeSelector()

Signature: makeSelector(String selector [, String type])

New in version 1.1-beta5.

Makes selector by defined type XPath, Name or Label. Function has same result as selectXPath in Casper module for XPath type - it makes XPath object. Function also accepts name attribute of the form field or can select element by its label text.

Parameter type values:

- 'css'
CSS3 selector - selector is returned transparently
- 'xpath' || null
XPath selector - return XPath object
- 'name' || 'names'
select input of specific name, internally covert to CSS3 selector
- 'label' || 'labels'
select input of specific label, internally converted into XPath selector. As selector is label's text used

Examples:

```
__utils__.makeSelector('//li[text()="blah"]', 'xpath'); // return {type: 'xpath',  
↪ path: '//li[text()="blah"]'}
```

```
__utils__.makeSelector('parameter', 'name'); // return '[name="parameter"]'
```

```
__utils__.makeSelector('My label', 'label'); // return {type: 'xpath', path: '//  
↪ *[@id=string(//label[text()="My label"]/@for)]'}
```

mouseEvent()

Signature: mouseEvent(String type, String selector, [Number|String X, Number|String Y])

Dispatches a mouse event to the DOM element behind the provided selector.

Supported events are mouseup, mousedown, click, dblclick, mousemove, mouseover, mouseout, mouseenter, mouseleave and contextmenu:


```
.. index:: XPath
```

removeElementsByXPath()

Signature: `removeElementsByXPath(String expression)`

Removes all DOM elements matching a given *XPath expression*.

sendAJAX()

Signature: `sendAJAX(String url[, String method, Object data, Boolean async, Object settings])`

New in version 1.0.

Sends an AJAX request, using the following parameters:

- `url`: The url to request.
- `method`: The HTTP method (default: GET).
- `data`: Request parameters (default: null).
- `async`: Flag for an asynchronous request? (default: false)
- `settings`: Custom Headers when perform the AJAX request (default: null). WARNING: an invalid header here may make the request fail silently.

Warning: Don't forget to pass the `--web-security=no` option in your CLI call in order to perform cross-domains requests when needed:

```
var data, wsurl = 'http://api.site.com/search.json';

casper.start('http://my.site.com/', function() {
  data = this.evaluate(function(wsurl) {
    return JSON.parse(__utils__.sendAJAX(wsurl, 'GET', null, false));
  }, {wsurl: wsurl});
});

casper.then(function() {
  require('utils').dump(data);
});
```

setFieldValue()

Signature: `setFieldValue(String|Object selector, Mixed value [, HTMLElement scope])`

New in version 1.1-beta5.

Sets a value to form field by CSS3 or XPath selector. With *makeSelector()* function can be easily used with name or label selector

Options

- (String|Object) scope: selector :
specific form scope

Examples:

```
__utils__.setFieldValue("input[name='email']", 'chuck@norris.com');
__utils__.setFieldValue("input[name='email']", 'chuck@norris.com', {'formSelector': '
↪#myform'});
__utils__.setFieldValue(__utils__.makeSelector('email', 'name'), 'chuck@norris.com');
```

visible()

Signature: visible(String selector)

Checks if an element is visible:

```
var logoIsVisible = casper.evaluate(function() {
  return __utils__.visible('h1');
});
```

The colorizer module

The colorizer module contains a Colorizer class which can generate ANSI colored strings:

```
var colorizer = require('colorizer').create('Colorizer');
console.log(colorizer.colorize("Hello World", "INFO"));
```

Though most of the times you will use it transparently using the *Casper.echo()* method:

```
casper.echo('an informative message', 'INFO'); // printed in green
casper.echo('an error message', 'ERROR');      // printed in red
```

Skipping CasperJS styling operations

If you wish to skip the whole coloration operation and get uncolored plain text, just set the `colorizerType` casper option to `Dummy`:

```
var casper = require('casper').create({
  colorizerType: 'Dummy'
});

casper.echo("Hello", "INFO");
```

Note: That's especially useful if you're using CasperJS on the Windows platform, as there's no support for colored output on this platform.

Available predefined styles

Available predefined styles are:

- ERROR: white text on red background
- INFO: green text
- TRACE: green text
- PARAMETER: cyan text
- COMMENT: yellow text
- WARNING: red text
- GREEN_BAR: white text on green background
- RED_BAR: white text on red background
- INFO_BAR: cyan text
- WARN_BAR: white text on orange background

Here's a sample output of what it can look like:

```
niko@NikoBook:~/Sites/casperjs$ phantomjs tests/run.js
# logging
PASS log() adds a log entry
# navigating
PASS start() can add a new navigation step
PASS then() adds a new navigation step
PASS start() casper can start itself an open an url
PASS start() injects ClientUtils instance within remote DOM
# encoding
PASS base64encode() can retrieve base64 contents
# clicking
PASS click() casper can click on a text link and react when it is loaded 1/2
PASS click() casper can click on a text link and react when it is loaded 2/2
# filling a form
PASS fill() can fill an input[type=text] form field
PASS fill() can fill a textarea form field
PASS fill() can pick a value from a select form field
PASS fill() can check a form checkbox
PASS fill() can check a form radio button 1/2
PASS fill() can check a form radio button 2/2
PASS fill() can select a file to upload
PASS click() casper can click on a submit button
PASS fill() input[type=email] field was submitted
PASS fill() textarea field was submitted
PASS fill() input[type=checkbox] field was submitted
PASS fill() input[type=radio] field was submitted
PASS fill() select field was submitted
PASS log() logged messages
PASS 22 tests executed, 22 passed, 0 failed.
niko@NikoBook:~/Sites/casperjs$
```

colorize()

Signature: `colorize(String text, String styleName)`

Computes a colored version of the provided text string using a given predefined style:

```
var colorizer = require('colorizer').create();
console.log(colorizer.colorize("I'm a red error", "ERROR"));
```

Note: Most of the time you won't have to use a `Colorizer` instance directly as CasperJS provides all the necessary methods.

See the list of the *predefined styles available*.

format()

Signature: `format(String text, Object style)`

Formats a text string using the provided style definition. A style definition is a standard javascript `Object` instance which can define the following properties:

- String `bg`: background color name
- String `fg`: foreground color name
- Boolean `bold`: apply bold formatting
- Boolean `underline`: apply underline formatting
- Boolean `blink`: apply blink formatting
- Boolean `reverse`: apply reverse formatting
- Boolean `conceal`: apply conceal formatting

Note: Available color names are black, red, green, yellow, blue, magenta, cyan and white:

```
var colorizer = require('colorizer').create();
colorizer.format("We all live in a yellow submarine", {
  bg: 'yellow',
  fg: 'blue',
  bold: true
});
```

The mouse module

The Mouse class

The `Mouse` class is an abstraction on top of various mouse operations like moving, clicking, double-clicking, rollovers, etc. It requires a `Casper` instance as a dependency for accessing the DOM. A mouse object can be created that way:

```
var casper = require("casper").create();
var mouse = require("mouse").create(casper);
```

Note: A casper instance has a `mouse` property already defined, so you usually don't have to create one by hand in your casper scripts:

```
casper.then(function() {  
  this.mouse.click(400, 300); // clicks at coordinates x=400; y=300  
});
```

`click()`

Signature:

- `click(Number x, Number y)`
- `click(String selector)`
- `click(String selector, Number x, Number y)`

Performs a click on the first element found matching the provided *selector expression* or at given coordinates if two numbers are passed:

```
casper.then(function() {  
  this.mouse.click("#my-link"); // clicks <a id="my-link">hey</a>  
  this.mouse.click(400, 300);   // clicks at coordinates x=400; y=300  
});
```

Note: You may want to directly use *Casper#click* instead.

`doubleclick()`

Signature:

- `doubleclick(Number x, Number y)`
- `doubleclick(String selector)`
- `doubleclick(String selector, Number x, Number y)`

Sends a `doubleclick` mouse event onto the element matching the provided arguments:

```
casper.then(function() {  
  this.mouse.doubleclick("#my-link"); // doubleclicks <a id="my-link">hey</a>  
  this.mouse.doubleclick(400, 300);   // doubleclicks at coordinates x=400; y=300  
});
```

`rightclick()`

Signature:

- `rightclick(Number x, Number y)`
- `rightclick(String selector)`
- `rightclick(String selector, Number x, Number y)`

Sends a `contextmenu` mouse event onto the element matching the provided arguments:

```
casper.then(function() {  
  this.mouse.rightclick("#my-link"); // doubleclicks <a id="my-link">hey</a>  
  this.mouse.rightclick(400, 300); // doubleclicks at coordinates x=400; y=300  
});
```

down()

Signature:

- `down(Number x, Number y)`
- `down(String selector)`
- `down(String selector, Number x, Number y)`

Sends a `mousedown` mouse event onto the element matching the provided arguments:

```
casper.then(function() {  
  this.mouse.down("#my-link"); // press left button down <a id="my-link">hey</a>  
  this.mouse.down(400, 300); // press left button down at coordinates x=400; y=300  
});
```

move()

Signature:

- `move(Number x, Number y)`
- `move(String selector)`
- `move(String selector, Number x, Number y)`

Moves the mouse cursor onto the element matching the provided arguments:

```
casper.then(function() {  
  this.mouse.move("#my-link"); // moves cursor over <a id="my-link">hey</a>  
  this.mouse.move(400, 300); // moves cursor over coordinates x=400; y=300  
});
```

up()

Signature:

- `up(Number x, Number y)`
- `up(String selector)`
- `up(String selector, Number x, Number y)`

Sends a `mouseup` mouse event onto the element matching the provided arguments:

```
casper.then(function() {  
  this.mouse.up("#my-link"); // release left button over <a id="my-link">hey</a>  
  this.mouse.up(400, 300); // release left button over coordinates x=400; y=300  
});
```

The tester module

Casper ships with a `tester` module and a `Tester` class providing an API for unit & functional testing purpose. By default you can access an instance of this class through the `test` property of any `Casper` class instance.

Note: The best way to learn how to use the `Tester` API and see it in action is probably to have a look at [CasperJS' own test suites](#).

The `Tester` prototype

`assert()`

Signature: `assert(Boolean condition[, String message])`

Asserts that the provided condition strictly resolves to a boolean `true`:

```
casper.test.assert(true, "true's true");
casper.test.assert(!false, "truth is out");
```

See also:

[`assertNot\(\)`](#)

`assertDoesntExist()`

Signature: `assertDoesntExist(String selector[, String message])`

Asserts that an element matching the provided *selector expression* doesn't exists within the remote DOM environment:

```
casper.test.begin('assertDoesntExist() tests', 1, function(test) {
  casper.start().then(function() {
    this.setContent('<div class="heaven"></div>');
    test.assertDoesntExist('.taxes');
  }).run(function() {
    test.done();
  });
});
```

See also:

[`assertExists\(\)`](#)

`assertEquals()`

Signature: `assertEquals(mixed testValue, mixed expected[, String message])`

Asserts that two values are strictly equivalent:

```
casper.test.begin('assertEquals() tests', 3, function(test) {
  test.assertEquals(1 + 1, 2);
  test.assertEquals([1, 2, 3], [1, 2, 3]);
  test.assertEquals({a: 1, b: 2}, {a: 1, b: 2});
  test.done();
});
```

See also:

[*assertNotEquals\(\)*](#)

assertEval()

Signature: `assertEval(Function fn[, String message, Mixed arguments])`

Asserts that a *code evaluation in remote DOM* strictly resolves to a boolean true:

```
casper.test.begin('assertEval() tests', 1, function(test) {
  casper.start().then(function() {
    this.setContent('<div class="heaven">beer</div>');
    test.assertEval(function() {
      return __utils__.findOne('.heaven').textContent === 'beer';
    });
  }).run(function() {
    test.done();
  });
});
```

assertEvalEquals()

Signature: `assertEvalEquals(Function fn, mixed expected[, String message, Mixed arguments])`

Asserts that the result of a *code evaluation in remote DOM* strictly equals to the expected value:

```
casper.test.begin('assertEvalEquals() tests', 1, function(test) {
  casper.start().then(function() {
    this.setContent('<div class="heaven">beer</div>');
    test.assertEvalEquals(function() {
      return __utils__.findOne('.heaven').textContent;
    }, 'beer');
  }).run(function() {
    test.done();
  });
});
```

assertElementCount()

Signature: `assertElementCount(String selector, Number count[, String message])`

Asserts that a *selector expression* matches a given number of elements:

```
casper.test.begin('assertElementCount() tests', 3, function(test) {
  casper.start().then(function() {
    this.page.content = '<ul><li>1</li><li>2</li><li>3</li></ul>';
    test.assertElementCount('ul', 1);
    test.assertElementCount('li', 3);
    test.assertElementCount('address', 0);
  }).run(function() {
    test.done();
  });
});
```


assertExists()**Signature:** `assertExists(String selector[, String message])`Asserts that an element matching the provided *selector expression* exists in remote DOM environment:

```
casper.test.begin('assertExists() tests', 1, function(test) {
  casper.start().then(function() {
    this.setContent('<div class="heaven">beer</div>');
    test.assertExists('.heaven');
  }).run(function() {
    test.done();
  });
});
```

See also:*assertDoesntExist()***assertFalsy()****Signature:** `assertFalsy(Mixed subject[, String message])`

New in version 1.0.

Asserts that a given subject is falsy.

See also:*assertTruthy()***assertField()****Signature:** `assertField(String|Object input, String expected[, String message, Object options])`Asserts that a given form field has the provided value with input name or *selector expression*:

```
casper.test.begin('assertField() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    this.fill('form[name="gs"]', { q: 'plop' }, false);
    test.assertField('q', 'plop');
  }).run(function() {
    test.done();
  });
});

// Path usage with type 'css'
casper.test.begin('assertField() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    this.fill('form[name="gs"]', { q: 'plop' }, false);
    test.assertField({type: 'css', path: '.q.foo'}, 'plop');
  }).run(function() {
    test.done();
  });
});
```

New in version 1.0.

This also works with any input type: `select`, `textarea`, etc.

New in version 1.1.

The *options* parameter allows to set the options to use with *ClientUtils#getFieldValue()*.

input parameter introspects whether or not a *type* key is passed in with *xpath* or *css* and a property *path* specified along with it.

assertFieldName()

Signature: `assertFieldName(String inputName, String expected[, String message, Object options])`

New in version 1.1-beta3.

Asserts that a given form field has the provided value:

```
casper.test.begin('assertFieldName() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    this.fill('form[name="gs"]', { q: 'plop' }, false);
    test.assertFieldName('q', 'plop', 'did not plop', {formSelector: 'plopper'});
  }).run(function() {
    test.done();
  });
});
```

assertFieldCSS()

Signature: `assertFieldCSS(String cssSelector, String expected, String message)`

New in version 1.1.

Asserts that a given form field has the provided value given a CSS selector:

```
casper.test.begin('assertFieldCSS() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    this.fill('form[name="gs"]', { q: 'plop' }, false);
    test.assertFieldCSS('q', 'plop', 'did not plop', 'input.plop');
  }).run(function() {
    test.done();
  });
});
```

assertFieldXPath()

Signature: `assertFieldXPath(String xpathSelector, String expected, String message)`

New in version 1.1.

Asserts that a given form field has the provided value given a XPath selector:

```
casper.test.begin('assertFieldXPath() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    this.fill('form[name="gs"]', { q: 'plop' }, false);
    test.assertFieldXPath('q', 'plop', 'did not plop', '/html/body/form[0]/
    ↪input[1]');
  }).run(function() {
    test.done();
  });
});
```

assertHttpStatus()

Signature: `assertHttpStatus(Number status[, String message])`

Asserts that current [HTTP status code](#) is the same as the one passed as argument:

```
casper.test.begin('casperjs.org is up and running', 1, function(test) {
  casper.start('http://casperjs.org/', function() {
    test.assertHttpStatus(200);
  }).run(function() {
    test.done();
  });
});
```

assertMatch()

Signature: `assertMatch(mixed subject, RegExp pattern[, String message])`

Asserts that a provided string matches a provided javascript RegExp pattern:

```
casper.test.assertMatch('Chuck Norris', /^chuck/i, 'Chuck Norris\' first name is Chuck
    ↪');
```

See also:

- [assertUrlMatch\(\)](#)
- [assertTitleMatch\(\)](#)

assertNot()

Signature: `assertNot(mixed subject[, String message])`

Asserts that the passed subject resolves to some falsy value:

```
casper.test.assertNot(false, "Universe is still operational");
```

See also:

[assert\(\)](#)

assertNotEquals()

Signature: `assertNotEquals(mixed testValue, mixed expected[, String message])`

New in version 0.6.7.

Asserts that two values are **not** strictly equals:

```
casper.test.assertNotEquals(true, "true");
```

See also:

[*assertEquals\(\)*](#)

assertNotVisible()

Signature: `assertNotVisible(String selector[, String message])`

Asserts that the element matching the provided *selector expression* is not visible:

```
casper.test.begin('assertNotVisible() tests', 1, function(test) {
  casper.start().then(function() {
    this.setContent('<div class="foo" style="display:none>boo</div>');
    test.assertNotVisible('.foo');
  }).run(function() {
    test.done();
  });
});
```

See also:

- [*assertVisible\(\)*](#)
- [*assertAllVisible\(\)*](#)

assertRaises()

Signature: `assertRaises(Function fn, Array args[, String message])`

Asserts that the provided function called with the given parameters raises a javascript Error:

```
casper.test.assertRaises(function(throwIt) {
  if (throwIt) {
    throw new Error('thrown');
  }
}, [true], 'Error has been raised.');
```

```
casper.test.assertRaises(function(throwIt) {
  if (throwIt) {
    throw new Error('thrown');
  }
}, [false], 'Error has been raised.');// fails
```

assertSelectorDoesntHaveText()

Signature: `assertSelectorDoesntHaveText(String selector, String text[, String message])`

Asserts that given text does not exist in all the elements matching the provided *selector expression*:

```
casper.test.begin('assertSelectorDoesntHaveText() tests', 1, function(test) {
  casper.start('http://google.com/', function() {
    test.assertSelectorDoesntHaveText('title', 'Yahoo!');
  }).run(function() {
    test.done();
  });
});
```

See also:

assertSelectorHasText()

assertSelectorHasText()

Signature: `assertSelectorHasText(String selector, String text[, String message])`

Asserts that given text exists in elements matching the provided *selector expression*:

```
casper.test.begin('assertSelectorHasText() tests', 1, function(test) {
  casper.start('http://google.com/', function() {
    test.assertSelectorHasText('title', 'Google');
  }).run(function() {
    test.done();
  });
});
```

See also:

assertSelectorDoesntHaveText()

assertResourceExists()

Signature: `assertResourceExists(Function testFx[, String message])`

The `testFx` function is executed against all loaded assets and the test passes when at least one resource matches:

```
casper.test.begin('assertResourceExists() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    test.assertResourceExists(function(resource) {
      return resource.url.match('logo3w.png');
    });
  }).run(function() {
    test.done();
  });
});
```

Shorter:

```
casper.test.begin('assertResourceExists() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    test.assertResourceExists('logo3w.png');
  }).run(function() {
    test.done();
  });
});
```

Hint: Check the documentation for *Casper.resourceExists()*.

assertTextExists()

Signature: `assertTextExists(String expected[, String message])`

Asserts that body **plain text content** contains the given string:

```
casper.test.begin('assertTextExists() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    test.assertTextExists('google', 'page body contains "google"');
  }).run(function() {
    test.done();
  });
});
```

See also:

assertTextDoesntExist()

assertTextDoesntExist()

Signature: `assertTextDoesntExist(String unexpected[, String message])`

New in version 1.0.

Asserts that body **plain text content** doesn't contain the given string:

```
casper.test.begin('assertTextDoesntExist() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    test.assertTextDoesntExist('bing', 'page body does not contain "bing"');
  }).run(function() {
    test.done();
  });
});
```

See also:

assertTextExists()

assertTitle()

Signature: `assertTitle(String expected[, String message])`

Asserts that title of the remote page equals to the expected one:

```
casper.test.begin('assertTitle() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    test.assertTitle('Google', 'google.fr has the correct title');
  }).run(function() {
    test.done();
  });
});
```

See also:

assertTitleMatch()

assertTitleMatch()

Signature: `assertTitleMatch(RegExp pattern[, String message])`

Asserts that title of the remote page matches the provided RegExp pattern:

```
casper.test.begin('assertTitleMatch() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    test.assertTitleMatch(/Google/, 'google.fr has a quite predictable title');
  }).run(function() {
    test.done();
  });
});
```

See also:

assertTitle()

assertTruthy()

Signature: `assertTruthy(Mixed subject[, String message])`

New in version 1.0.

Asserts that a given subject is `truthy`.

See also:

assertFalsy()

assertType()

Signature: `assertType(mixed input, String type[, String message])`

Asserts that the provided input is of the given type:

```
casper.test.begin('assertType() tests', 1, function suite(test) {
  test.assertType(42, "number", "Okay, 42 is a number");
  test.assertType([1, 2, 3], "array", "We can test for arrays too!");
  test.done();
});
```

Note: Type names are always expressed in lower case.

assertInstanceOf()

Signature: `assertInstanceOf(mixed input, Function constructor[, String message])`

New in version 1.1.

Asserts that the provided input is of the given constructor:

```
function Cow() {
  this.moo = function moo() {
    return 'moo!';
  };
}
casper.test.begin('assertInstanceOf() tests', 2, function suite(test) {
  var daisy = new Cow();
  test.assertInstanceOf(daisy, Cow, "Ok, daisy is a cow.");
  test.assertInstanceOf(["moo", "boo"], Array, "We can test for arrays too!");
  test.done();
});
```

assertUrlMatch()

Signature: `assertUrlMatch(RegExp pattern[, String message])`

Asserts that the current page url matches the provided `RegExp` pattern:

```
casper.test.begin('assertUrlMatch() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    test.assertUrlMatch(/^http:\/\/\//, 'google.fr is served in http://');
  }).run(function() {
    test.done();
  });
});
```

assertVisible()

Signature: `assertVisible(String selector[, String message])`

Asserts that at least one element matching the provided *selector expression* is visible:

```
casper.test.begin('assertVisible() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    test.assertVisible('h1');
  }).run(function() {
    test.done();
  });
});
```

See also:

- [*assertAllVisible\(\)*](#)
- [*assertNotVisible\(\)*](#)

assertAllVisible()

Signature: `assertAllVisible(String selector[, String message])`

Asserts that all elements matching the provided *selector expression* are visible:

```
casper.test.begin('assertAllVisible() tests', 1, function(test) {
  casper.start('http://www.google.fr/', function() {
    test.assertAllVisible('input[type="submit"]');
  }).run(function() {
```



```

        test.done();
    });
});

```

See also:

- [assertVisible\(\)](#)
- [assertNotVisible\(\)](#)

begin()

Signatures:

- `begin(String description, Number planned, Function suite)`
- `begin(String description, Function suite)`
- `begin(String description, Number planned, Object config)`
- `begin(String description, Object config)`

New in version 1.1.

Starts a suite of <planned> tests (if defined). The suite callback will get the current `Tester` instance as its first argument:

```

function Cow() {
    this.mowed = false;
    this.moo = function moo() {
        this.mowed = true; // mootable state: don't do that
        return 'moo!';
    };
}

// unit style synchronous test case
casper.test.begin('Cow can moo', 2, function suite(test) {
    var cow = new Cow();
    test.assertEquals(cow.moo(), 'moo!');
    test.assert(cow.mowed);
    test.done();
});

```

Note: The planned argument is especially useful in case a given test script is abruptly interrupted leaving you with no obvious way to know it and an erroneously successful status.

A more asynchronous example:

```

casper.test.begin('Casperjs.org is navigable', 2, function suite(test) {
    casper.start('http://casperjs.org/', function() {
        test.assertTitleMatches(/casperjs/i);
        this.clickLabel('Testing');
    });

    casper.then(function() {
        test.assertUrlMatches(/testing\.html$/);
    });
});

```

```
casper.run(function() {
    test.done();
});
```

Important: `done()` must be called in order to terminate the suite. This is specially important when doing asynchronous tests so ensure it's called when everything has actually been performed.

See also:

`done()`

`Tester#begin()` also accepts a test configuration object, so you can add `setUp()` and `tearDown()` methods:

```
// cow-test.js
casper.test.begin('Cow can moo', 2, {
    setUp: function(test) {
        this.cow = new Cow();
    },

    tearDown: function(test) {
        this.cow.destroy();
    },

    test: function(test) {
        test.assertEquals(this.cow.moo(), 'moo!');
        test.assert(this.cow.mowed);
        test.done();
    }
});
```

colorize()

Signature: `colorize(String message, String style)`

Render a colored output. Basically a proxy method for `Casper.Colorizer#colorize()`.

comment()

Signature: `comment(String message)`

Writes a comment-style formatted message to stdout:

```
casper.test.comment("Hi, I'm a comment");
```

done()

Signature: `done()`

Changed in version 1.1: `planned` parameter is deprecated

Flag a test suite started with `begin()` as processed:

```
casper.test.begin('my test suite', 2, function(test) {
  test.assert(true);
  test.assertNot(false);
  test.done();
});
```

More asynchronously:

```
casper.test.begin('Casperjs.org is navigable', 2, function suite(test) {
  casper.start('http://casperjs.org/', function() {
    test.assertTitleMatches(/casperjs/i);
    this.clickLabel('Testing');
  });

  casper.then(function() {
    test.assertUrlMatches(/testing\.html$/);
  });

  casper.run(function() {
    test.done();
  });
});
```

See also:

begin()

error()

Signature: error(String message)

Writes an error-style formatted message to stdout:

```
casper.test.error("Hi, I'm an error");
```

fail()

Signature: fail(String message [, Object option])

Adds a failed test entry to the stack:

```
casper.test.fail("Georges W. Bush");
casper.test.fail("Here goes a really long and expressive message", {name:'shortfacts'}
↪);
```

See also:

pass()

formatMessage()

Signature: formatMessage(String message, String style)

Formats a message to highlight some parts of it. Only used internally by the tester.

getFailures()

Signature: getFailures()

New in version 1.0.

Deprecated since version 1.1.

Retrieves failures for current test suite:

```
casper.test.assertEquals(true, false);
require('utils').dump(casper.test.getFailures());
casper.test.done();
```

That will give something like this:

```
$ casperjs test test-getFailures.js
Test file: test-getFailures.js
FAIL Subject equals the expected value
#   type: assertEquals
#   subject: true
#   expected: false
{
  "length": 1,
  "cases": [
    {
      "success": false,
      "type": "assertEquals",
      "standard": "Subject equals the expected value",
      "file": "test-getFailures.js",
      "values": {
        "subject": true,
        "expected": false
      }
    }
  ]
}
FAIL 1 tests executed, 0 passed, 1 failed.

Details for the 1 failed test:

In c.js:0
  assertEquals: Subject equals the expected value
```

Note: In CasperJS 1.1, you can record test failures by listening to the `tester fail` event:

```
var failures = [];

casper.test.on("fail", function(failure) {
  failures.push(failure);
});
```

getPasses()

Signature: getPasses()

New in version 1.0.

Deprecated since version 1.1.

Retrieves a report for successful test cases in the current test suite:

```
casper.test.assertEquals(true, true);
require('utils').dump(casper.test.getPasses());
casper.test.done();
```

That will give something like this:

```
$ casperjs test test-getPasses.js
Test file: test-getPasses.js
PASS Subject equals the expected value
{
  "length": 1,
  "cases": [
    {
      "success": true,
      "type": "assertEquals",
      "standard": "Subject equals the expected value",
      "file": "test-getPasses.js",
      "values": {
        "subject": true,
        "expected": true
      }
    }
  ]
}
PASS 1 tests executed, 1 passed, 0 failed.
```

Note: In CasperJS 1.1, you can record test successes by listening to the tester success event:

```
var successes = [];

casper.test.on("success", function(success) {
  successes.push(success);
});
```

info()

Signature: info(String message)

Writes an info-style formatted message to stdout:

```
casper.test.info("Hi, I'm an informative message.");
```

pass()

Signature: pass(String message)

Adds a successful test entry to the stack:

```
casper.test.pass("Barrack Obama");
```

See also:

fail()

renderResults()

Signature: renderResults(Boolean exit, Number status, String save)

Render test results, save results in an XUnit formatted file, and optionally exits phantomjs:

```
casper.test.renderResults(true, 0, 'test-results.xml');
```

Note: This method is not to be called when using the `casperjs test` command (see documentation for *testing*), where it's done automatically for you.

setUp()

Signature: setUp([Function fn])

Defines a function which will be executed before every test defined using *begin()*:

```
casper.test.setUp(function() {  
    casper.start().userAgent('Mosaic 0.1');  
});
```

To perform asynchronous operations, use the `done` argument:

```
casper.test.setUp(function(done) {  
    casper.start('http://foo').then(function() {  
        // ...  
    }).run(done);  
});
```

Warning: Don't specify the `done` argument if you don't intend to use the method asynchronously.

See also:

tearDown()

skip()

Signature: skip(Number nb, String message)

Skips a given number of planned tests:

```
casper.test.begin('Skip tests', 4, function(test) {  
    test.assert(true, 'First test executed');  
    test.assert(true, 'Second test executed');  
    test.skip(2, 'Two tests skipped');
```

```
test.done();
});
```

`tearDown()`

Signature: `tearDown([Function fn])`

Defines a function which will be executed after every test defined using *begin()*:

```
casper.test.tearDown(function() {
  casper.echo('See ya');
});
```

To perform asynchronous operations, use the `done` argument:

```
casper.test.tearDown(function(done) {
  casper.start('http://foo/goodbye').then(function() {
    // ...
  }).run(done);
});
```

Warning: Don't specify the `done` argument if you don't intend to use the method asynchronously.

See also:

setUp()

The `utils` module

This module provides simple helper functions, some of them being very specific to CasperJS though.

Functions reference

Usage is pretty much straightforward:

```
var utils = require('utils');
utils.dump({plop: 42});
```

`betterTypeOf()`

Signature: `betterTypeOf(input)`

Provides a better `typeof` operator equivalent, eg. able to retrieve the `Array` type.

`betterInstanceOf()`

New in version 1.1.

Signature: `betterInstanceOf(input, constructor)`

Provides a `better instanceof` operator equivalent, is able to retrieve the `Array` instance or to deal with inheritance.

`dump()`

Signature: `dump(value)`

Dumps a `JSON` representation of passed argument to the standard output. Useful for *debugging your scripts*.

`fileExt()`

Signature: `fileExt(file)`

Retrieves the extension of passed filename.

`fillBlanks()`

Signature: `fillBlanks(text, pad)`

Fills a string with trailing spaces to match `pad` length.

`format()`

Signature: `format(f)`

Formats a string against passed args. `sprintf` equivalent.

Note: This is a port of `nodejs util.format()`.

`getPropertyPath()`

Signature: `getPropertyPath(Object obj, String path)`

New in version 1.0.

Retrieves the value of an `Object` foreign property using a dot-separated path string:

```
var account = {
  username: 'chuck',
  skills: {
    kick: {
      roundhouse: true
    }
  }
}
utils.getPropertyPath(account, 'skills.kick.roundhouse'); // true
```


Warning: This function doesn't handle object key names containing a dot.

`inherits()`

Signature: `inherits(ctor, superCtor)`

Makes a constructor inheriting from another. Useful for subclassing and *extending*.

Note: This is a port of `nodejs util.inherits()`.

`isArray()`

Signature: `isArray(value)`

Checks if passed argument is an instance of `Array`.

`isCasperObject()`

Signature: `isCasperObject(value)`

Checks if passed argument is an instance of `Casper`.

`isClipRect()`

Signature: `isClipRect(value)`

Checks if passed argument is a `cliprect` object.

`isFalsy()`

Signature: `isFalsy(subject)`

New in version 1.0.

Returns subject *falsiness*.

`isFunction()`

Signature: `isFunction(value)`

Checks if passed argument is a function.

`isJsFile()`

Signature: `isJsFile(file)`

Checks if passed filename is a Javascript one (by checking if it has a `.js` or `.coffee` file extension).

`isNull()`

Signature: `isNull(value)`

Checks if passed argument is a `null`.

`isNumber()`

Signature: `isNumber(value)`

Checks if passed argument is an instance of `Number`.

`isObject()`

Signature: `isObject(value)`

Checks if passed argument is an object.

`isString()`

Signature: `isString(value)`

Checks if passed argument is an instance of `String`.

`isTruthy()`

Signature: `isTruthy(subject)`

New in version 1.0.

Returns subject [truthiness](#).

`isType()`

Signature: `isType(what, type)`

Checks if passed argument has its type matching the `type` argument.

`isUndefined()`

Signature: `isUndefined(value)`

Checks if passed argument is undefined.

`isWebPage()`

Signature: `isWebPage(what)`

Checks if passed argument is an instance of native PhantomJS' `WebPage` object.

mergeObjects()

Signature: `mergeObjects(origin, add[, Object opts])`

Merges two objects recursively.

Add `opts.keepReferences` if cloning of internal objects is not needed.

node()

Signature: `node(name, attributes)`

Creates an (HT|X)ML element, having optional `attributes` added.

serialize()

Signature: `serialize(value)`

Serializes a value using **JSON** format. Will serialize functions as strings. Useful for *debugging* and comparing objects.

unique()

Signature: `unique(array)`

Retrieves unique values from within a given `Array`.

Writing CasperJS modules

As of 1.1, CasperJS relies on PhantomJS' native `require()` function internally though it had to be patched in order to allow requiring casper modules using their full name, eg. `require('casper')`.

So if you plan to write your own modules and use casperjs' ones from them, be sure to call the `patchRequire()` function:

```
// my module, stored in universe.js
// patching phantomjs' require()
var require = patchRequire(require);

// now you're ready to go
var utils = require('utils');
var magic = 42;
exports.answer = function() {
    return utils.format("it's %d", magic);
};
```

Warning: When using CoffeeScript `global.require` must be passed to `patchRequire()` instead of just `require`:

```
require = patchRequire global.require

utils = require 'utils'
magic = 42
exports.answer = ->
    utils.format "it's ${magic}"
```

From your root casper script:

```
var universe = require('./universe');
console.log(universe.answer()); // prints "It's 42"
```

New in version 1.1..

Hint: CasperJS allows using nodejs modules installed through [npm](#). Note that since CasperJS uses it's own JavaScript environment, npm modules that use node-specific features will not work under CasperJS.

As an example, let's install the [underscore](#) library:

```
$ npm install underscore
```

Then, require it like you would with any other nodejs compliant module:

```
//npm-underscore-test.js
var _ = require('underscore');
var casper = require('casper').create();
var urls = _.uniq([
  'http://google.com/',
  'http://docs.casperjs.org/',
  'http://google.com/'
]);

casper.start().eachThen(urls, function(response) {
  this.thenOpen(response.data, function(response) {
    this.echo(this.getTitle());
  });
});

casper.run();
```

Finally, you'll probably get something like this:

```
$ casperjs npm-underscore-test.js
Google
CasperJS documentation | CasperJS 1.1.0-DEV documentation
```

CasperJS provides an *event handler* very similar to the `nodejs`' one; actually it borrows most of its codebase. CasperJS also adds *filters*, which are basically ways to alter values asynchronously.

Events

Using events is pretty much straightforward if you're a node developer, or if you worked with any evented system before:

```
var casper = require('casper').create();

casper.on('resource.received', function(resource) {
    casper.echo(resource.url);
});
```

Emitting you own events

Of course you can emit your own events, using the `Casper.emit()` method:

```
var casper = require('casper').create();

// listening to a custom event
casper.on('google.loaded', function() {
    this.echo('Google page title is ' + this.getTitle());
});

casper.start('http://google.com/', function() {
    // emitting a custom event
    this.emit('google.loaded');
});

casper.run();
```

Removing events

You can also remove events. This is particularly useful when running a lot of tests where you might need to add and remove different events for different tests:

```
var casper = require('casper').create();

// listener function for requested resources
var listener = function(resource, request) {
    this.echo(resource.url);
};

// listening to all resources requests
casper.on("resource.requested", listener);

// load the google homepage
casper.start('http://google.com/', function() {
    this.echo(this.getTitle());
});

casper.run().then(function() {
    // remove the event listener
    this.removeListener("resource.requested", listener);
});
```

Here is an example of how to use this in a casperjs test within the `tearDown` function.:

```
var currentRequest;

//Resource listener
function onResourceRequested(requestData, request) {
    if (/\/jquery\.min\.js/.test(requestData.url)) {
        currentRequest = requestData;
    }
}

casper.test.begin('jQuery Test', 1, {
    setUp: function() {
        // Attach the resource listener
        casper.on('resource.requested', onResourceRequested);
    },

    tearDown: function() {
        // Remove the resource listener
        casper.removeListener('resource.requested', onResourceRequested);
        currentRequest = undefined;
    },

    test: function(test) {
        casper.start('http://casperjs.org/', function() {
            test.assert(currentRequest !== undefined, "jQuery Exists");
        });

        casper.run(function() {
            test.done();
        });
    }
});
```


Events reference

`back`

Arguments: None

Emitted when the embedded browser is asked to go back a step in its history.

`capture.saved`

Arguments: `targetFile`

Emitted when a screenshot image has been captured.

`click`

Arguments: `selector`

Emitted when the `Casper.click()` method has been called.

`complete.error`

Arguments: `error`

New in version 1.1.

Emitted when a complete callback has errored.

By default, CasperJS doesn't listen to this event, you have to declare your own listeners by hand:

```
casper.on('complete.error', function(err) {  
  this.die("Complete callback has failed: " + err);  
});
```

`die`

Arguments: `message`, `status`

Emitted when the `Casper.die()` method has been called.

`downloaded.file`

Arguments: `targetPath`

Emitted when a file has been downloaded by `Casper.download()`; `target` will contain the path to the downloaded file.

`downloaded.error`

Arguments: `url`

Emitted when a file has encountered an error when downloaded by `Casper.download()`; `url` will contain the url of the downloaded file.

`error`

Arguments: `msg`, `backtrace`

New in version 0.6.9.

Emitted when an error hasn't been explicitly caught within the CasperJS/PhantomJS environment. Do basically what PhantomJS' `onError()` native handler does.

`exit`

Arguments: `status`

Emitted when the `Casper.exit()` method has been called.

`fill`

Arguments: `selector`, `vals`, `submit`

Emitted when a form is filled using the `Casper.fill()` method.

`forward`

Arguments: `None`

Emitted when the embedded browser is asked to go forward a step in its history.

`frame.changed`

Arguments: `name`, `status`

Emitted when the current frame is changed with `Casper.withPopup`, `Casper.switchToFrame()`

`http.auth`

Arguments: `username`, `password`

Emitted when http authentication parameters are set.

`http.status.[code]`

Arguments: `resource`

Emitted when any given HTTP response is received with the status code specified by `[code]`, eg.:

```
casper.on('http.status.404', function(resource) {
  casper.echo(resource.url + ' is 404');
})
```

`load.started`

Arguments: `None`

Emitted when PhantomJS' `WebPage.onLoadStarted` event callback is called.

load.failed**Arguments:** Object

Emitted when PhantomJS' `WebPage.onLoadFinished` event callback has been called and failed.

load.finished**Arguments:** status

Emitted when PhantomJS' `WebPage.onLoadFinished` event callback is called.

log**Arguments:** entry

Emitted when the `Casper.log()` method has been called. The `entry` parameter is an Object like this:

```
{
  level:    "debug",
  space:    "phantom",
  message:  "A message",
  date:     "a javascript Date instance"
}
```

mouse.click**Arguments:** args

Emitted when the mouse left-click something or somewhere.

mouse.down**Arguments:** args

Emitted when the mouse presses on something or somewhere with the left button.

mouse.move**Arguments:** args

Emitted when the mouse moves onto something or somewhere.

mouse.up**Arguments:** args

Emitted when the mouse releases the left button over something or somewhere.

`navigation.requested`

Arguments: url, navigationType, navigationLocked, isMainFrame

New in version 1.0.

Emitted each time a navigation operation has been requested. Available navigation types are: `LinkClicked`, `FormSubmitted`, `BackOrForward`, `Reload`, `FormResubmitted` and `Other`.

`open`

location, settings

Emitted when an HTTP request is sent. First callback arg is the location, second one is a request settings Object of the form:

```
{
  method: "post",
  data:   "foo=42&chuck=norris"
}
```

`page.created`

Arguments: page

Emitted when PhantomJS' `WebPage` object used by CasperJS has been created.

`page.error`

Arguments: message, trace

Emitted when retrieved page leaves a Javascript error uncaught:

```
casper.on("page.error", function(msg, trace) {
  this.echo("Error: " + msg, "ERROR");
});
```

`page.initialized`

Arguments: `WebPage`

Emitted when PhantomJS' `WebPage` object used by CasperJS has been initialized.

`page.resource.received`

Arguments: response

Emitted when the HTTP response corresponding to current required url has been received.

`page.resource.requested`

Arguments: request

Emitted when a new HTTP request is performed to open the required url.

New in version 1.1.

Arguments: requestData, request

You can also abort requests:

```
casper.on('page.resource.requested', function(requestData, request) {  
    if (requestData.url.indexOf('http://adserver.com') === 0) {  
        request.abort();  
    }  
});
```

`popup.created`

Arguments: WebPage

Emitted when a new window has been opened.

`popup.loaded`

Arguments: WebPage

Emitted when a new window has been loaded.

`popup.closed`

Arguments: WebPage

Emitted when a new opened window has been closed.

`remote.alert`

Arguments: message

Emitted when a remote `alert()` call has been performed.

`remote.callback`

Arguments: data

Emitted when a remote `window.callPhantom(data)` call has been performed.

`remote.longRunningScript`

Arguments: WebPage

Emitted when any remote `longRunningScript` call has been performed.

You have to call `stopJavaScript` method

```
casper.on('remote.longRunningScript', function stopLongScript (webpage) {  
    webpage.stopJavaScript();  
    return true;  
});
```

`remote.message`

Arguments: msg

Emitted when any remote console logging call has been performed.

`resource.error`

Arguments: resourceError

Emitted when any requested resource fails to load properly. The received `resourceError` object has the following properties:

- `errorCode`: error code
- `errorString`: error description
- `url`: resource url
- `id`: resource id

`resource.received`

Arguments: resource

Emitted when any resource has been received.

`resource.requested`

Arguments: request

Emitted when any resource has been requested.

`resource.timeout`

Arguments: request

Emitted when the execution time of any resource has exceeded the value of `settings.resourceTimeout`. you can configure timeout with `settings.resourceTimeout` parameter.

`run.complete`

Arguments: None

Emitted when the whole series of steps in the stack have been executed.

run.start

Arguments: None

Emitted when `Casper.run()` is called.

starting

Arguments: None

Emitted when `Casper.start()` is called.

started

Arguments: None

Emitted when Casper has been started using `Casper.start()`.

step.added

Arguments: `step`

Emitted when a new navigation step has been added to the stack.

step.bypassed

Arguments: `step`, `step`

Emitted when a new navigation step has been reached by bypass (destination, origin).

step.complete

Arguments: `stepResult`

Emitted when a navigation step has been executed.

step.created

Arguments: `fn`

Emitted when a new navigation step has been created.

step.error

Arguments: `error`

New in version 1.1.

Emitted when a step function has errored.

By default, CasperJS doesn't listen to this event, you have to declare your own listeners by hand:

```
casper.on('step.error', function(err) {  
  this.die("Step has failed: " + err);  
});
```

`step.start`

Arguments: `step`

Emitted when a navigation step has been started.

`step.timeout`

Arguments: [`step`, `timeout`]

Emitted when a navigation step has timed out.

`timeout`

Arguments: None

Emitted when the execution time of the script has reached the `Casper.options.timeout` value.

`url.changed`

Arguments: `url`

New in version 1.0.

Emitted each time the current page url changes.

`viewport.changed`

Arguments: [`width`, `height`]

Emitted when the viewport has been changed.

`wait.done`

Arguments: None

Emitted when a `Casper.wait()` operation ends.

`wait.start`

Arguments: None

Emitted when a `Casper.wait()` operation starts.

`waitFor.timeout`

Arguments: [timeout, details]

Emitted when the execution time of a `Casper.wait*()` operation has exceeded the value of `timeout`.

`details` is a property bag describing what was being waited on. For example, if `waitForSelector` timed out, `details` will have a `selector` string property that was the selector that did not show up in time.

Filters

Filters allow you to alter some values asynchronously. Sounds obscure? Let's take a simple example and imagine you would like to alter every single url opened by CasperJS to append a `foo=42` query string parameter:

```
var casper = require('casper').create();

casper.setFilter('open.location', function(location) {
  return /\?+/.test(location) ? location += "&foo=42" : location += "?foo=42";
});
```

There you have it, every single requested url will have this appended. Let me bet you'll find far more interesting use cases than my silly one :)

Every filter methods called emit an identical event. For instance, "page.confirm" filter sends "page.confirm" event.

Here's a the list of all available filters with their expected return value:

Filters reference

`capture.target_filename`

Arguments: args

Return type: String

Allows to alter the value of the filename where a screen capture should be stored.

`echo.message`

Arguments: message

Return type: String

Allows to alter every message written onto stdout.

`log.message`

Arguments: message

Return type: String

Allows to alter every log message.

`open.location`

Arguments: args

Return type: String

Allows to alter every url before it being opened.

`page.confirm`

Arguments: message

Return type: Boolean

New in version 1.0.

Allows to react on a javascript `confirm()` call:

```
casper.setFilter("page.confirm", function(msg) {  
    return msg === "Do you like vbscript?" ? false : true;  
});
```

`page.filePicker`

Arguments: oldFile

Return type: String

New in version 1.4.

Allows to react on a webpage.onFilePicker call:

```
casper.setFilter("page.filePicker", function(oldFile) {  
    if (system.os.name === 'windows') {  
        return 'C:\\Windows\\System32\\drivers\\etc\\hosts';  
    }  
    return '/etc/hosts';  
});
```

`page.prompt`

Arguments: message, value

Return type: String

New in version 1.0.

Allows to react on a javascript `prompt()` call:

```
casper.setFilter("page.prompt", function(msg, value) {  
    if (msg === "What's your name?") {  
        return "Chuck";  
    }  
});
```

CasperJS allows logging using the `casper.log()` method and these standard event levels:

- debug
- info
- warning
- error

Sample use:

```
var casper = require('casper').create();
casper.log('plop', 'debug');
casper.log('plip', 'warning');
```

Now, there are two things to distinguish: log *storage* and log *display*; by default CasperJS won't print the logs to the standard output. In order to do so, you must enable the verbose Casper option:

```
var casper = require('casper').create({
  verbose: true
});
```

Also, by default Casper is configured to filter logging which is under the `error` level; you can override this setting by configuring the `logLevel` option:

```
var casper = require('casper').create({
  verbose: true,
  logLevel: 'debug'
});
```

You can also dump a JSON log of your Casper suite just by rendering the contents of the `Casper.result.log` property:

```
var casper = require('casper').create({
  // ...
```

```
casper.run(function() {  
    require('utils').dump(this.result.log);  
    this.exit();  
});
```

Last, if you print log messages to the standard output using the `verbose` option, you'll get some fancy colors:

```
var casper = require('casper').create({  
    verbose: true,  
    logLevel: 'debug'  
});  
casper.log('this is a debug message', 'debug');  
casper.log('and an informative one', 'info');  
casper.log('and a warning', 'warning');  
casper.log('and an error', 'error');  
casper.exit();
```

This will give the following output:

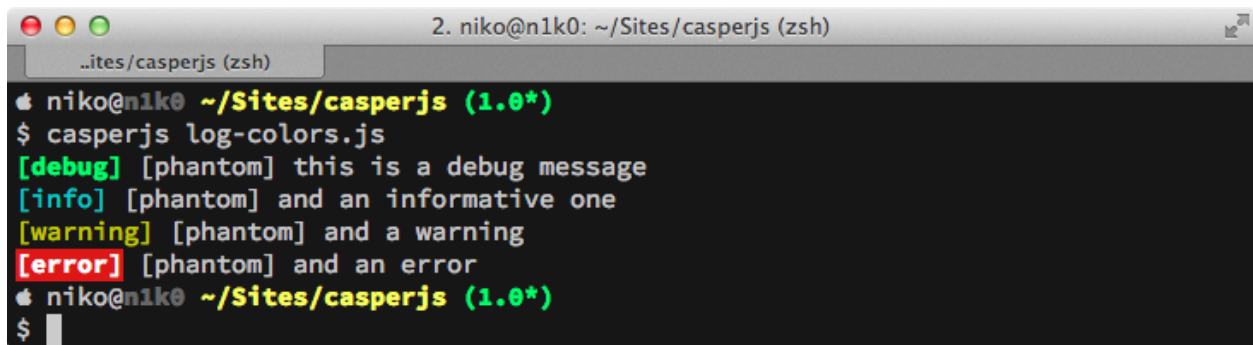


Fig. 9.1: image

Hint: CasperJS doesn't write logs on the filesystem. You have to implement this by yourself if needed.

CHAPTER 10

Extending

Sometimes it can be convenient to add your own methods to a Casper object instance; you can easily do so as illustrated in the example below:

```
var casper = require('casper').create({
  verbose: true,
  logLevel: "debug"
});

var links = {
  'http://edition.cnn.com/': 0,
  'http://www.nytimes.com/': 0,
  'http://www.bbc.co.uk/': 0,
  'http://www.guardian.co.uk/': 0
};

casper.countLinks = function() {
  return this.evaluate(function() {
    return __utils__.findAll('a[href]').length;
  });
};

casper.renderJSON = function(what) {
  return this.echo(JSON.stringify(what, null, '  '));
};

casper.start();

casper.each(Object.keys(links), function(casper, link) {
  this.thenOpen(link, function() {
    links[link] = this.countLinks();
  });
});

casper.run(function() {
  this.renderJSON(links).exit();
});
```

```
});
```

But that’s just plain old *monkey-patching* the `casper` object, and you may probably want a more OO approach... That’s where the `inherits()` function from the `utils` module and ported from `nodejs` comes handy:

```
var Casper = require('casper').Casper;
var utils = require('utils');
var links = {
  'http://edition.cnn.com/': 0,
  'http://www.nytimes.com/': 0,
  'http://www.bbc.co.uk/': 0,
  'http://www.guardian.co.uk/': 0
};

function Fantomas() {
  Fantomas.super_.apply(this, arguments);
}

// Let's make our Fantomas class extending the Casper one
// please note that at this point, CHILD CLASS PROTOTYPE WILL BE OVERRIDEN
utils.inherits(Fantomas, Casper);

Fantomas.prototype.countLinks = function() {
  return this.evaluate(function() {
    return __utils__.findAll('a[href]').length;
  });
};

Fantomas.prototype.renderJSON = function(what) {
  return this.echo(JSON.stringify(what, null, '  '));
};

var fantomas = new Fantomas({
  verbose: true,
  logLevel: "debug"
});

fantomas.start();

Object.keys(links).forEach(function(url) {
  fantomas.thenOpen(url, function() {
    links[url] = this.countLinks();
  });
});

fantomas.run(function() {
  this.renderJSON(links).exit();
});
```

Note: The use of the `super_` child class property which becomes available once its parent has been defined using `inherits()`; it contains a reference to the parent constructor.

Don’t forget to call “Casper”’s parent constructor!

Of course this approach is bit more verbose than the easy *monkey-patching* one, so please ensure you’re not just overengineering stuff by subclassing the `Casper` class.

Using CoffeeScript

If you're writing your casper scripts using [CoffeeScript](#), extending casper is getting a bit more straightforward:

```
links =
  'http://edition.cnn.com/': 0
  'http://www.nytimes.com/': 0
  'http://www.bbc.co.uk/': 0
  'http://www.guardian.co.uk/': 0

class Fantomas extends require('casper').Casper
  countLinks: ->
    @evaluate ->
      __utils__.findAll('a').length

  renderJSON: (what) ->
    @echo JSON.stringify what, null, '  '

fantomas = new Fantomas
  loadImages: false
  logLevel:   "debug"
  verbose:    true

fantomas.start()

for url of links
  do (url) ->
    fantomas.thenOpen url, ->
      links[url] = @countLinks()

fantomas.run ->
  @renderJSON links
  @exit()
```


A few tips for debugging your casper scripts:

- *Use the verbose mode*
- *Hook in the deep using events*
- *Dump serialized values to the console*
- *Localize yourself in modules*
- *Name your closures*

Use the verbose mode

By default & by design, a Casper instance won't print anything to the console. This can be very limiting & frustrating when creating or debugging scripts, so a good practice is to always start coding a script using these settings:

```
var casper = require('casper').create({
  verbose: true,
  logLevel: "debug"
});
```

The `verbose` setting will tell Casper to write every logged message at the `logLevel` logging level onto the standard output, so you'll be able to trace every step made.

Warning: Output will then be pretty verbose, and will potentially display sensitive informations onto the console. Use with care on production.

Hook in the deep using events

Events are a very powerful features of CasperJS, and you should probably give it a look if you haven't already.

Some interesting events you may eventually use to debug your scripts:

- The `http.status.XXX` event will be emitted everytime a resource is sent with the [HTTP code](#) corresponding to XXX;
- The `remote.alert` everytime an `alert()` call is performed client-side;
- `remote.message` everytime a message is sent to the client-side console;
- `step.added` everytime a step is added to the stack;
- etc...

Listening to an event is dead easy:

```
casper.on('http.status.404', function(resource) {  
    this.log('Hey, this one is 404: ' + resource.url, 'warning');  
});
```

Ensure to check the [full list](#) of all the other available events.

Dump serialized values to the console

Sometimes it's helpful to inspect a variable, especially Object contents. The *utils_dump()* function can achieve just that:

```
require('utils').dump({  
    foo: {  
        bar: 42  
    },  
});
```

Note: *utils_dump()* won't be able to serialize function nor complex cyclic structures though.

Localize yourself in modules

Warning: Deprecated since version 1.1.

As of 1.1, CasperJS uses PhantomJS' builtin *require* and won't expose the `__file__` variable anymore.

If you're creating Casper modules, a cool thing to know is that there's a special built-in variable available in every module, `__file__`, which contains the absolute path to current javascript file (the module file).

Name your closures

Probably one of the most easy but effective best practice, always name your closures:

Hard to track:

```
casper.start('http://foo.bar/', function() {  
  this.evaluate(function() {  
    // ...  
  });  
});
```

Easier:

```
casper.start('http://foo.bar/', function afterStart() {  
  this.evaluate(function evaluateStuffAfterStart() {  
    // ...  
  });  
});
```

That way, everytime one is failing, its name will be printed out in the stack trace, **so you can more easily locate it within your code.**

Note: This one also applies for all your other Javascript works, of course ;)

Here's a selection of the most frequently asked questions by CasperJS newcomers:

- *Is CasperJS a node.js library?*
- *I'm stuck! I think there's a bug! What can I do?*
- *The `casper.test` property is undefined, I can't write any test!*
- *I keep copy and pasting stuff in my test scripts, that's boring*
- *What is the versioning policy of CasperJS?*
- *Can I use jQuery with CasperJS?*
- *Can I use CasperJS without using the `casperjs` executable?*
- *How can I catch HTTP 404 and other status codes?*
- *Where does CasperJS write its logfile?*
- *What's this mysterious `__utils__` object?*
- *How does `then()` and the step stack work?*
- *I'm having hard times downloading files using `download()`*
- *Is it possible to achieve parallel browsing using CasperJS?*
- *Can I access & manipulate DOM elements directly from the CasperJS environment?*
- *Why can't I create a new casper instance in a test environment?*
- *Okay, honestly, I'm stuck with Javascript.*
- *How do I use PhantomJS page module API in `casperjs`?*
- *How do I provide my implementation of a remote resource?*
- *I'm getting intermittent test failure, what can I do to fix them?*

Is CasperJS a node.js library?

No. CasperJS is written on top of [PhantomJS](#), which is a node-independent [Qt/WebKit](#) based library. If you try to run your CasperJS script with node, it just won't work out of the box.

Hint: If you want to drive CasperJS from node, try [SpookyJS](#).

I'm stuck! I think there's a bug! What can I do?

Before rage-tweeting:

1. Read the [docs](#)
2. Check if an [issue](#) has been open about your problem already
3. Check you're running the [latest stable tag](#)
4. Check you're running the [latest version](#) of PhantomJS
5. Ask on the [project mailing list](#):
 - (a) try to post a reproducible, minimal test case
 - (b) compare casperjs results with native phantomjs ones
 - (c) if the problem also occurs with native phantomjs, ask on [phantomjs mailing list](#)
6. Eventually, [file an issue](#).

The `casper.test` property is undefined, I can't write any test!

That's because as of 1.1, the `casper.test` property is only set to a [Tester](#) instance when using the `casperjs test` subcommand.

You may want to read the [testing documentation](#) for more information.

I keep copy and pasting stuff in my test scripts, that's boring

Have a look at [this gist](#), it might help.

Also, don't forget that CasperJS supports a [CommonJS-compliant module pattern](#) implementation.

Note: CasperJS' implementation of `require()` differs a bit from the one provided by [PhantomJS](#), but I personally never encountered any functional difference.

What is the versioning policy of CasperJS?

Releases will follow the [SemVer standard](#); they will be numbered with the follow format:

```
<major>.<minor>.<patch>[-<identifier>]
```

And constructed with the following guidelines:

- Breaking backwards compatibility bumps the major
- New additions without breaking backwards compatibility bumps the minor
- Bug fixes and misc changes bump the patch
- Unstable, special and trunk versions will have a proper identifier

Can I use jQuery with CasperJS?

Sure, you can use [jQuery](#), as every single other javascript library on Earth.

A first solution is to inject it into the remote DOM environment by hand using the standard `WebPage.injectJs()` method:

```
casper.page.injectJs('/path/to/jquery.js');
```

In the event that you require jQuery being available on every page, you can make use of the `clientScripts` option of CasperJS:

```
var casper = require('casper').create({
  clientScripts: ["includes/jquery.min.js"]
});
```

Note: You can't *inject* scripts using the HTTP protocol, you actually have to use a relative/absolute filesystem path to the script resource.

Can I use CasperJS without using the casperjs executable?

Yes, you can call a CasperJS script directly with the `phantomjs` executable, but if you do so, you must set the `phantom.casperPath` property to the path where the library root is located on your system:

```
// casperscript.js
phantom.casperPath = '/path/to/casperjs';
phantom.injectJs(phantom.casperPath + '/bin/bootstrap.js');

var casper = require('casper').create();
// ...
```

You can run such a script like any other standard [PhantomJS](#) script:

```
$ phantomjs casperscript.js
```

If you're on Windows, this is the way you may manage to get casper working the most easily:

```
phantom.casperPath = 'C:\\path\\to\\your\\repo\\lib\\casperjs-0.6.X';
phantom.injectJs(phantom.casperPath + '\\bin\\bootstrap.js');

var casper = require('casper').create();
```

```
// do stuff
```

How can I catch HTTP 404 and other status codes?

You can define your own [HTTP status code](#) handlers by using the `httpStatusHandlers` option of the Casper object. You can also catch other HTTP status codes as well, as demoed below:

```
var casper = require('casper').create();

casper.on('http.status.404', function(resource) {
  this.echo('wait, this url is 404: ' + resource.url);
});

casper.on('http.status.500', function(resource) {
  this.echo('woops, 500 error: ' + resource.url);
});

casper.start('http://mywebsite/404', function() {
  this.echo('We suppose this url return an HTTP 404');
});

casper.thenOpen('http://mywebsite/500', function() {
  this.echo('We suppose this url return an HTTP 500');
});

casper.run(function() {
  this.echo('Done.').exit();
});
```

Hint: Check out all the other cool [events](#) you may use as well.

Where does CasperJS write its logfile?

Nowhere. CasperJS doesn't write logs on the filesystem. You have to implement this by yourself if needed.

What's this mysterious `__utils__` object?

The `__utils__` object is actually a *ClientUtils object* which have been automatically injected into the page DOM and is therefore always available.

So everytime to perform an *evaluate()* call, you have this instance available to perform common operation like:

- fetching nodes using CSS3 or XPath selectors,
- retrieving information about element properties (attributes, size, bounds, etc.),
- sending AJAX requests,
- triggering DOM events

Check out the [whole API](#). You even have [a bookmarklet](#) to play around with this `__utils__` instance right within your browser console!

Note: You're not obliged at all to use the `__utils__` instance in your scripts. It's just there because it's used by CasperJS internals.

How does `then()` and the step stack work?

Disclaimer This entry is based on an [answer I made on Stack Overflow](#).

The `then()` method basically adds a new navigation step in a stack. A step is a javascript function which can do two different things:

1. waiting for the previous step - if any - being executed
2. waiting for a requested url and related page to load

Let's take a simple navigation scenario:

```
var casper = require('casper').create();

casper.start();

casper.then(function step1() {
    this.echo('this is step one');
});

casper.then(function step2() {
    this.echo('this is step two');
});

casper.thenOpen('http://google.com/', function step3() {
    this.echo('this is step 3 (google.com is loaded)');
});
```

You can print out all the created steps within the stack like this:

```
require('utils').dump(casper.steps.map(function(step) {
    return step.toString();
}));
```

That gives:

```
$ casperjs test-steps.js
[
  "function step1() { this.echo('this is step one'); }",
  "function step2() { this.echo('this is step two'); }",
  "function _step() { this.open(location, settings); }",
  "function step3() { this.echo('this is step 3 (google.com is loaded)'); }"
]
```

Notice the `_step()` function which has been added automatically by CasperJS to load the url for us; when the url is loaded, the next step available in the stack — which is `step3()` — is *then* called.

When you have defined your navigation steps, `run()` executes them one by one sequentially:

```
casper.run();
```

Note: The callback/listener stuff is an implementation of the [Promise pattern](#).

I'm having hard times downloading files using `download()`

You should try to disable *web security*. Using the `--web-security` command line option:

```
$ casperjs --web-security=no myscript.js
```

Within code:

```
var casper = require('casper').create({
  pageSettings: {
    webSecurityEnabled: false
  }
});
```

Or anytime:

```
casper.page.settings.webSecurityEnabled = false;
```

Is it possible to achieve parallel browsing using CasperJS?

Officially **no**, but you may want to try.

Can I access & manipulate DOM elements directly from the CasperJS environment?

No. Like in PhantomJS, you have to use *Casper#evaluate()* to access actual page DOM and manipulate elements.

For example, you **can't** do this:

```
// this won't work
casper.then(function() {
  var titleNode = document.querySelector('h1');
  this.echo('Title is: ' + titleNode.textContent);
  titleNode.textContent = 'New title';
  this.echo('Title is now: ' + titleNode.textContent);
});
```

You have to use the *Casper#evaluate()* method in order to communicate with the page DOM:

```
// this will
casper.then(function() {
  var titleText = this.evaluate(function() {
    return document.querySelector('h1').textContent;
  });
});
```

```

    this.echo('Title is: ' + titleText);
    this.evaluate(function() {
        document.querySelector('h1').textContent = 'New title';
    });
    this.echo('Title is now: ' + this.evaluate(function() {
        return document.querySelector('h1').textContent;
    }));
});

```

Of course, it's a whole lot more verbose, but Casper provides convenient methods to ease accessing elements properties, eg. *Casper#fetchText()* and *Casper#getElementInfo()*:

```

// this will
casper.then(function() {
    this.echo('Title is: ' + this.fetchText('h1'));
    this.evaluate(function() {
        document.querySelector('h1').textContent = 'New title';
    });
    this.echo('Element HTML is now: ' + this.getElementInfo('h1').html);
});

```

Why can't I create a new *casper* instance in a test environment?

The *casperjs test subcommand* is a convenient utility which bootstraps and configures a *test environment* for you, so a preconfigured *casper* object is already available in your test script when using this command.

As of 1.1-beta3, you're prevented from overriding this preconfigured instance as this practice prevents the test runner from working properly. If you try to create a new casper instance in a test script, you'll get an error and CasperJS will exit with an error message with a link pointing to the documentation.

One may argue this is mostly related to some historical bad design decisions, and this might be true. This behavior is not likely to exist anymore in a future 2.0.

Okay, honestly, I'm stuck with Javascript.

Don't worry, you're not alone. Javascript is a great language, but it's far more difficult to master than one might expect at first look.

Here are some great resources to get started efficiently with the language:

- Learn and practice Javascript online at [Code Academy](#)
- [Eloquent Javascript](#)
- [JavaScript Enlightenment \(PDF\)](#)
- last, a [great tutorial on Advanced Javascript Techniques](#) by John Resig, the author of jQuery. If you master this one, you're almost done with the language.

How do I use PhantomJS page module API in casperjs?

After *casperjs.start()*, you have phantomjs page module available in *casper.page* (<http://docs.casperjs.org/en/latest/modules/casper.html#page>)

You can simply do like below:

```
casper.page.nameOfMethod()
```

PhantomJS Web Page API: <http://phantomjs.org/api/webpage/>

How do I provide my implementation of a remote resource?

Using phantomjs native *onResourceRequested* event, you can override remote resource url to your own implementation. Your own implementation file can be provided from local path too:

```
casper.page.onResourceRequested = function(requestData, networkRequest) {
    var match = requestData.url.match(/wordfamily.js/g);
    if (match != null) {
        console.log('Request (' + requestData.id + '): ' + JSON.
↪stringify(requestData));

        // overrides wordfamily.js to local newWordFamily.js
        networkRequest.changeUrl('newWordFamily.js');
    }
};
```

I'm getting intermittent test failure, what can I do to fix them?

This is probably because you are executing a test before the resource or element is available and the page is fully loaded/rendered. This can even happen on things like modals and dynamic content.

You can solve this problem by using the *wait** operations:

```
casper.thenOpen(url, function initialAppearance() {
    casper.waitForText('Text in deep part of page or modal');
});
```

It is good practice to wait for DOM nodes, text, or resources before beginning your tests. It will help make them stable and predictable while still running fast.

This is a collection of scripts and ideas that aim to solve common situations that are encountered by users. This is by no means an exhaustive list, and we encourage you to contribute your recipes on [github](#).

Creating a web service

Warning: It is worth noting that this is probably not the best of ideas. You should be careful of things like memory leaks, lack of long term stability (due to said leaks), and the overall memory hog that headless JS can be.

With the above caveat in mind, a web service would look something like:

```
//filename: server.js

//define ip and port to web service
var ip_server = '127.0.0.1:8585';

//includes web server modules
var server = require('webserver').create();

//start web server
var service = server.listen(ip_server, function(request, response) {
  var links = [];
  var casper = require('casper').create();

  function getLinks() {
    var links = document.querySelectorAll('h3.r a');
    return Array.prototype.map.call(links, function(e) {
      return e.getAttribute('href')
    });
  }
});
```

```
casper.start('http://google.com/', function() {
  // search for 'casperjs' from google form
  this.fill('form[action="/search"]', { q: request.postRow }, true);
});

casper.then(function() {
  // aggregate results for the 'casperjs' search
  links = this.evaluate(getLinks);
});

casper.run(function() {
  response.statusCode = 200;

  //sends results as JSON object
  response.write(JSON.stringify(links, null, null));
  response.close();
});
});
console.log('Server running at http://' + ip_server+'/');
```

You can start the server by executing:

```
casperjs server.js
```

You can then access the results via an HTTP POST request:

```
curl --data "casperjs" http://127.0.0.1:8585/
```

The above command would search for “casperjs” on google and return a JSON array of results. This is a trivial example and can be expanded into something more complex.

Script to automatically check a page for 404 and 500 errors

```
var casper = require("casper").create({
  pageSettings: {
    loadImages: false,
    loadPlugins: false
  }
});
var checked = [];
var currentLink = 0;
var fs = require('fs');
var upTo = ~~casper.cli.get('max-depth') || 100;
var url = casper.cli.get(0);
var baseUrl = url;
var links = [url];
var utils = require('utils');
var f = utils.format;

function absPath(url, base) {
  return new URI(url).resolve(new URI(base)).toString();
}

// Clean links
function cleanLinks(urls, base) {
```

```

    return utils.unique(urls).filter(function(url) {
        return url.indexOf(baseUrl) === 0 || !new RegExp('^(#|ftp|javascript|http)').
↪test(url);
    }).map(function(url) {
        return absPath(url, base);
    }).filter(function(url) {
        return checked.indexOf(url) === -1;
    });
}

// Opens the page, perform tests and fetch next links
function crawl(link) {
    this.start().then(function() {
        this.echo(link, 'COMMENT');
        this.open(link);
        checked.push(link);
    });
    this.then(function() {
        if (this.currentHTTPStatus === 404) {
            this.warn(link + ' is missing (HTTP 404)');
        } else if (this.currentHTTPStatus === 500) {
            this.warn(link + ' is broken (HTTP 500)');
        } else {
            this.echo(link + f(' is okay (HTTP %s)', this.currentHTTPStatus));
        }
    });
    this.then(function() {
        var newLinks = searchLinks.call(this);
        links = links.concat(newLinks).filter(function(url) {
            return checked.indexOf(url) === -1;
        });
        this.echo(newLinks.length + " new links found on " + link);
    });
}

// Fetch all <a> elements from the page and return
// the ones which contains a href starting with 'http://'
function searchLinks() {
    return cleanLinks(this.evaluate(function _fetchInternalLinks() {
        return [].map.call(__utils__.findAll('a[href]'), function(node) {
            return node.getAttribute('href');
        });
    })), this.getCurrentUrl());
}

// As long as it has a next link, and is under the maximum limit, will keep running
function check() {
    if (links[currentLink] && currentLink < upTo) {
        crawl.call(this, links[currentLink]);
        currentLink++;
        this.run(check);
    } else {
        this.echo("All done, " + checked.length + " links checked.");
        this.exit();
    }
}

if (!url) {

```

```
casper.warn('No url passed, aborting.').exit();
}

casper.start('https://js-uri.googlecode.com/svn/trunk/lib/URI.js', function() {
  var scriptCode = this.getPageContent() + '; return URI;';
  window.URI = new Function(scriptCode)();
  if (typeof window.URI === "function") {
    this.echo('URI.js loaded');
  } else {
    this.warn('Could not setup URI.js').exit();
  }
});

casper.run(process);

function process() {
  casper.start().then(function() {
    this.echo("Starting");
  }).run(check);
}
```

Run it with:

```
casperjs 404checker.js http://mysite.tld/ [--max-depth=42]
```

[Reference gist.](#)

Test drag&drop

Assuming a page containing a draggable element like that [one](#), we can test drag&drop that way:

```
casper.test.begin('Test drag&drop', 2, function(test) {
  casper.start('http://localhost:8000/example.html', function() {
    test.assertEval(function() {
      var pos = $('#box').position();
      return (pos.left == 0 && pos.top == 0);
    }, "The box is at the top");
    this.mouse.down(5, 5);
    this.mouse.move(400, 200);
    this.mouse.up(400, 200);
  });
  casper.then(function() {
    test.assertEval(function() {
      var pos = $('#box').position();
      return (pos.left == 395 && pos.top == 195);
    }, "The box has been moved");
  });
  casper.run(function() {
    test.done();
  });
});
```


Passing parameters into your tests

Let's say you want to be able to change the Uri your tests visits depending on what you are testing. To do this, you can add custom `-parameter=value` to your cli.

```
// casperjs test /foo/bar --url=test.html
var url = 'http://localhost:8000'
var cli = casper.cli

if (cli.has('url')) {
  url = cli.get('url')
}
console.log('\n\tUsing url: ' + url + '\n')

casper.test.begin(...)
```

You can find the complete documentation for the cli object in <http://docs.casperjs.org/en/latest/cli.html>

CHAPTER 14

Changelog

The CasperJS changelog is [hosted on github](#).

Upgrading to 1.1

Testing framework refactor

The most visible change is the way you write tests. With 1.0, you were able to access a `.test` property from any casper script and so running a suite using the standard `casperjs` executable:

```
// 1.0 style test script not using the `casperjs test` subcommand
var casper = require('casper').create();

casper.start('http://foo.bar/', function() {
    this.test.assert(true);
});

casper.run(function() {
    this.test.done(1);
    this.test.renderResults(true);
});
```

In 1.1, the test framework has been heavily refactored to decouple the tester from a casper instance as much as possible, so it's no more possible to run a test suite right from the standard `casperjs` command as you would have done with the script shown above.

Instead you now have to use the `casperjs test` subcommand mandatorily to access a tester instance from the `casper.test` property.

Warning: As of 1.1:

- you shouldn't invoke the `renderResults()` method directly anymore
- you shouldn't use the `done()` first argument to set planned test as it's been deprecated
- you can't access the `casper.test` property when not using the `casperjs test` subcommand

If you try, you'll get an error:

```
// test.js
var casper = require('casper').create();
casper.test.assert(true);
```

Will give:

```
$ casperjs test.js
CasperError: casper.test property is only available using the `casperjs test`
↳ command
```

The new `Tester#begin()` method

However, a new `begin()` method has been added to the `Tester` prototype, to ease describing your tests:

```
casper.test.begin('Description of my test', 1, function(test) {
  test.assert(true);
  test.done();
});
```

More asynchronously:

```
casper.test.begin('Description of my test', 1, function(test) {
  casper.start('http://foo.bar/', function() {
    test.assert(true);
  });

  casper.run(function() {
    test.done();
  });
});
```

Note: Please notice `begin()`'s second argument which is now the place to set the number of planned tests.

`require()` in custom modules

CasperJS 1.1 now internally uses PhantomJS' native `require()` function, but it has side effect if you write your own casperjs modules; in any casperjs module, you now have to use the new global `patchRequire()` function first:

```
// casperjs module code
var require = patchRequire(require);
// now you can require casperjs builtins
var utils = require('utils');
exports = {
  // ...
};
```

Note: You don't have to use `patchRequire()` in a standard casperjs script.

`__file__` has been removed

As of 1.1, CasperJS now uses native PhantomJS' `require()` function which doesn't support the `__file__` builtin variable within custom modules like 1.0 allowed.

`Tester#getFailures()` and `Tester#getPasses()` methods removed

These two methods have been removed from the *Tester* API.

You can retrieve test failure and success records by simply accessing `tester.currentSuite.failures` and `tester.currentSuite.passes` instead.

Step and run completion callbacks don't throw anymore

Instead, you should listen to the `step.error` and `complete.error` events; if you really want to keep raising them:

```
casper.on("step.error complete.error", function(error) {  
    throw error;  
});
```


CHAPTER 16

Known Issues

This is a non-exhaustive list of issues that the CasperJS team is aware of and tracking.

PhantomJS

Versions below 2.0.0:

- [phantomjs-issue-10795](#):

There is a known issue while doing clicks within the page that causes execution to halt. It has been fixed in v2.0.0+ in phantomjs.

It is mentioned in the following issues: [#233](#)

```
console.log('START click');
console.log(document.getElementById('foo').toString());
console.log(document.getElementById('foo').click()); // this ends execution
console.log('END click'); // this never gets called
```

Version 2.0.0:

- [phantomjs-issue-12506](#):

Webpage.uploadFile is not working. It has been fixed in v2.0.1+ in phantomjs.

- [phantomjs-issue-12410](#):

Quote from [PhantomJS 2.0 Release Note](#):

“PhantomJS 2 can not run scripts written in CoffeeScript anymore (see issue [12410](#)). As a workaround, CoffeeScript users can still compile their scripts to JavaScript first before executing it with PhantomJS.”

Author

CasperJS is mainly developed by [Nicolas Perriault](#) on its free time.

If you want to thank him and/or sponsor the development of CasperJS, please consider donating (see links in the sidebar).

Contributors

These people have contributed to CasperJS:

- Brikou CARRE
- Thomas Parisot
- Han Yu
- Chris Lorenzo
- Victor Yap
- Rob Barreca
- Tyler Ritchie
- Nick Rabinowitz
- Pascal Borreli
- Dave Lee
- Andrew Childs
- Solomon White
- Reina Sweet

- Jan Schaumann
- Elmar Langholz
- Clochix
- Donovan Hutchinson
- Julien Moulin
- Michael Geers
- Jason Funk
- Vladimir Chizhov
- Jean-Philippe Serafin
- snkashis
- Rafael
- Andrew de Andrade
- Ben Lowery
- Chris Winters
- Christophe Benz
- Harrison Reiser
- Jan Pochyla
- Jan-Martin Fruehwacht
- Julian Gruber
- Justin Slattery
- Justine Tunney
- KaroDidi
- Leandro Boscariol
- Maisons du monde
- Marcel Duran
- Mathieu Agopian
- Mehdi Kabab
- Mikko Peltonen
- Rafael Garcia
- Raphael Benitte
- Tim Bunce

Logo

CasperJS logo designed by [Jeremy Forveille](#)

You can download the logo sources [here](#):

- logo CasperJS (PDF)
- logo CasperJS (EPS)
- logo CasperJS (AI)

These assets are under [MIT license](#)

CHAPTER 18

License

CasperJS is released under the terms of the MIT license.

```
Copyright (c) 2011-{{year}} Nicolas Perriault
Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

You can also search the genindex if you're looking for something particular.

CHAPTER 19

Community

- [get the code](#) and [contribute](#)
- [join the mailing list](#)
- [check out the ecosystem](#)
- follow [@casperjs_org](#) on Twitter
- there's also a [Google+ account](#) (not much updated though)

Symbols

`__utils__`, 68, 130

A

AJAX, 75, 130

alert, 61

arguments, 10

Asynchronicity, 54, 61, 64, 91, 92, 131

auth, 29, 53, 108

B

Base64, 31, 34, 69, 71

Binary, 71

bookmarklet, 69

Browser testing, 18

Bugs, 6, 121, 128

bypass, 32, 56

C

Casper, 25

Casper options, 25

Child Process, 62

CLI, 10

click, 32, 33, 107, 109

Client scripts, 26

Client utils, 68

Code reuse, 128

coffeescript, 9, 120

colorizer, 76

Colors, 76, 92

Command line, 10

Community, 153

Continuous Integration, 21

Contributing, 128, 153

Cookbook, 134

CORS, 132

CSS, 15

CSS3, 15

Custom module, 101

D

Debugging, 35, 46, 69, 98, 121

DOM, 14, 37, 39, 43–45, 47, 60, 68, 69, 81, 82, 87, 90, 101

DOMReady, 56

done(), 92

download, 36, 107, 132

dump, 98

E

echo, 37, 69

error, 26, 27, 86, 107, 128

Error handling, 27, 28

evaluate, 37, 82

events, 47, **105**, 123, 124

Examples, 134

exec File, 62

exit, 26, 39, 108

extending, 22, 118

F

falsiness, 83, 99

FAQ, 125

fill, 108

filters, 115

Form, 40, 45, 52, 73, 83

Frames, 66

Framesets, 66

Functional testing, 18

G

git, 4

Globals, 46

H

Help, 125, 153

Helpers, 97

Homebrew, 3

HTML, 14

HTTP, 26, 27, 48, 50, 64, 85, 87, 108, 110, 130
HTTP Headers, 48
HTTP Method, 48
HTTP Request, 48
HTTP Response, 55
HTTP Status Code, 85

I

Iframes, 66
inheritance, 99, 118
initialization, 53
Installation, 1
InstanceOf, 89

J

Jenkins, 21
jQuery, 69, 129
JSON, 98, 101

K

Known Issues, 145

L

Licensing, 151
log, 73, 109, 130
log levels, 12, 116
Logging, 12, 26, 30, 40, 116, 130

M

Modules, 101
modules, 23
Mouse, 78

N

New window, 63, 67
Node.js, 127

O

options, 10, 20, 25

P

PhantomJS, 3, 10, 29
planned tests, 91
Popups, 63, 67
Printing, 37
Printing styles, 76
prototype, 118
Python, 3, 129

R

Raw values, 13
Remote scripts, 29
REPL, 6

Ruby, 129
run, 50

S

Samples, 134
screenshot, 33–35, 107, 115
Scroll, 51
selector, 14, 65, 87
Serialization, 98
settings, 29, 123
setUp, 20
Shell, 10
sleep, 60
SlimerJS, 3, 12
Spawn, 62
SSL, 29
stack trace, 125
start, 53
Step stack, 28, 30, 32, 50, 54, 56, 131
String formatting, 98
Support, 153

T

Tabs, 63, 67
tearDown, 20
Termination, 91, 92
Test failure, 93
Test success, 95
Test suite, 18, 91, 92
Testing, 16, 80, 128
timeout, 28, 30, 31
truthiness, 89, 100
Type, 89

U

Unit testing, 17
URL, 42, 64, 90
User Agent, 59
Utilities, 97

V

verbose, 30, 117, 123
Versioning, 128
viewport, 30, 59, 114

W

wait, 60
Web security, 132
window, 46
window.open, 63, 67
Windows, 5, 76, 129

X

XML, 21

XPath, [16](#), [72](#), [73](#)
XSS, [29](#)
XUnit, [21](#)