

# Zombie.js



**Insanely fast, full-stack, headless browser  
testing**

## Table of Content

Table of Content	2
Zombie.js	4
The Bite	4
Infection	5
Walking	5
Hunting	7
Feeding	8
Readiness	9
In The Family	10
Reporting Glitches	10
Giving Back	11
Brains	11
See Also	12
The Zombie API	13
The Browser	13
new zombie.Browser(options?) : Browser	13
Browser Options	13
browser.visit(url, callback)	14
browser.visit(url, options, callback)	14
browser.open() : Window	14
browser.window : Window	14
browser.error : Error	14
browser.errors : Array	14
Document Content	14
browser.body : Element	15
browser.document : Document	15
browser.evaluate(expr) : Object	15
browser.html(selector?, context?) : String	15
browser.queryAll(selector, context?) : Array	16
browser.query(selector, context?) : Element	16
browser.text(selector, context?) : String	16
browser.xpath(expression, context?) => XPathResult	16
Navigation	17
browser.clickLink(selector, callback)	17
browser.link(selector) : Element	17
browser.location : Location	17
browser.location = url	18
browser.statusCode : Number	18
browser.success : Boolean	18
browser.visit(url, callback)	18
browser.visit(url, options, callback)	18
browser.redirected : Boolean	19
Forms	19
browser.attach(selector, filename, callback) : this	19
browser.check(field, callback) : this	19
browser.choose(field, callback) : this	19
browser.field(selector) : Element	20
browser.fill(field, value, callback) : this	20
browser.button(selector) : Element	20
browser.pressButton(selector, callback)	20

browser.select(field, value, callback) : this	21
browser.selectOption(option, callback) : this	21
browser.uncheck(field, callback) : this	21
browser.unselect(field, value, callback) : this	21
browser.unselectOption(option, callback) : this	22
<b>State Management</b>	<b>22</b>
browser.cookies(domain, path?) : Cookies	22
browser.fork() : Browser	22
browser.loadCookies(String)	23
browser.loadHistory(String)	23
browser.loadStorage(String)	23
browser.localStorage(host) : Storage	23
browser.saveCookies() : String	23
browser.saveHistory() : String	23
browser.saveStorage() : String	24
browser.sessionStorage(host) : Storage	24
<b>Interaction</b>	<b>24</b>
browser.onalert(fn)	24
browser.onconfirm(question, response)	24
browser.onconfirm(fn)	24
browser.onprompt(message, response)	24
browser.onprompt(fn)	24
browser.prompted(message) => boolean	25
<b>Events</b>	<b>25</b>
browser.fire(name, target, callback?)	25
browser.wait(callback)	26
browser.wait(duration, callback)	26
Event: 'done'	26
Event: 'error'	26
Event: 'loaded'	26
<b>Debugging</b>	<b>26</b>
browser.dump()	27
browser.lastError : Object	27
browser.lastRequest : Object	27
browser.lastResponse : Object	27
browser.log(arguments)	27
browser.log(function)	27
browser.resources : Object	27
browser.viewInBrowser(name?)	27
<b>CSS Selectors</b>	<b>29</b>
<b>Troubleshooting guide</b>	<b>32</b>
The Dump	32
Debugging	33
Request/response	33

# Zombie.js

## Zombie.js

**Insanely fast, headless full-stack testing using Node.js**

### The Bite

If you're going to write an insanely fast, headless browser, how can you not call it Zombie? Zombie it is.

Zombie.js is a lightweight framework for testing client-side JavaScript code in a simulated environment. No browser required.

Let's try to sign up to a page and see what happens:

```
var zombie = require("zombie");
var assert = require("assert");

// Load the page from localhost
zombie.visit("http://localhost:3000/", function (e, browser, status) {

  // Fill email, password and submit form
  browser.
    fill("#email", "zombie@underworld.dead").
    fill("#password", "eat-the-living").
    pressButton("#Sign Me Up!", function(e, browser, status) {

      // Form submitted, new page loaded.
      assert.equal(status, 200);
      assert.equal(browser.text("#title"), "Welcome To Brains Depot");

    })

  });
```

Well, that was easy.

## Infection

To install Zombie.js you need Node.js, NPM, a C++ compiler and Python.

On OS X start by installing XCode, or use the [OSX GCC installer](#) (less to download).

Next, assuming you're using the mighty [Homebrew](#):

```
$ brew install node
$ node --version
v0.6.2
$ curl http://npmjs.org/install.sh | sudo sh
$ npm --version
1.0.106
$ npm install zombie
```

On Ubuntu try these steps:

```
$ sudo apt-get install python-software-properties
$ sudo add-apt-repository ppa:chris-lea/node.js
$ sudo apt-get update
$ sudo apt-get install nodejs nodejs-dev npm
$ node --version
v0.6.2
$ npm --version
1.0.106
$ npm install zombie
```

On Windows you'll need Cygwin to get access to GCC, Python, etc. [Read this](#) for detailed instructions and troubleshooting.

## Walking

To start off we're going to need a browser. A browser maintains state

across requests: history, cookies, HTML 5 local and session storage, etc. A browser has a main window, and typically a document loaded into that window.

You can create a new `zombie.Browser` and point it at a document, either by setting the `location` property or calling its `visit` function. As a shortcut, you can just call the `zombie.visit` function with a URL and callback.

The browser will load the document and if the document includes any scripts, also load and execute these scripts. It will then process some events, for example, anything your scripts do on page load. All of that, just like a real browser, happens asynchronously.

To wait for the page to fully load and process events, you pass `visit` a callback function. Zombie will then call your callback with `null`, the browser object, the status code of the last response, and an array of errors (hopefully empty). This is JavaScript, so you don't need to declare all these arguments, and in fact can access them as `browser.statusCode` and `browser.errors`.

(Why would the first callback argument be `null`? It works great when using asynchronous testing frameworks like [Vows.js](#))

Most errors that occur “resource loading and JavaScript execution” are not fatal, so rather than stopping processing, they are collected in `browser.errors`. As a convenience, you can get the last error by calling `browser.error`, for example:

```
browser.visit("http://localhost:3000", function () {
  assert.equal(browser.success, "Expected status code to be 2xx");
  if (browser.error)
    console.dir("Errors reported:", browser.errors);
})
```

Whenever you want to wait for all events to be processed, just call `browser.wait` with a callback. If you know how long the wait is (e.g. animation or page transition), you can pass a duration (in milliseconds) as the first argument. □

Otherwise, Zombie makes best judgement by waiting up to 5 seconds for the page to load resources (scripts, XHR requests, iframes), process DOM events, and fire timeouts events. It is quite common for pages to fire timeout events as they load, e.g. jQuery's `onready`. Usually these events delay the test by no more than a few milliseconds.

Read more [on the Browser API](#)

## Hunting

There are several ways you can inspect the contents of a document. For starters, there's the [DOM API](#), which you can use to find elements and traverse the document tree.

You can also use CSS selectors to pick a specific element or node list. Zombie.js implements the [DOM Selector API](#). These functions are available from every element, the document, and the **Browser** object itself.

To get the HTML contents of an element, read its `innerHTML` property. If you want to include the element itself with its attributes, read the element's `outerHTML` property instead. Alternatively, you can call the `browser.html` function with a CSS selector and optional context element. If the function selects multiple elements, it will return the combined HTML of them all.

To see the textual contents of an element, read its `textContent` property. Alternatively, you can call the `browser.text` function with a CSS selector and optional context element. If the function selects multiple elements, it will return the combined text contents of them all.

Here are a few examples for checking the contents of a document:

```
// Make sure we have an element with the ID brains.
assert.ok(browser.query("#brains"));

// Make sure body has two elements with the class hand.
assert.lengthOf(browser.body.queryAll(".hand"), 2);

// Check the document title.
```

```
assert.equal(browser.text("title"), "The Living Dead");

// Show me the document contents.
console.log(browser.html());

// Show me the contents of the parts table:
console.log(browser.html("table.parts"));
```

CSS selectors are implemented by Sizzle.js. In addition to CSS 3 selectors you get additional and quite useful extensions, such as `:not(selector)`, `[NAME!=VALUE]`, `:contains(TEXT)`, `:first/:last` and so forth. Check out the [Sizzle.js documentation](#) for more details.

Read more [on the Browser API](#) and [CSS selectors](#)

## Feeding

You're going to want to perform some actions, like clicking links, entering text, submitting forms. You can certainly do that using the [DOM API](#), or several of the convenience functions we're going to cover next.

To click a link on the page, use `clickLink` with selector and callback. The first argument can be a CSS selector (see *Hunting*), the `A` element, or the text contents of the `A` element you want to click.

The second argument is a callback, which much like the `visit` callback gets fired after all events are processed. □

Let's see that in action:

```
// Now go to the shopping cart page and check that we have
// three bodies there.
browser.clickLink("View Cart", function(e, browser, status) {
  assert.lengthOf(browser.queryAll("#cart .body"), 3);
});
```

To submit a form, use `pressButton`. The first argument can be a CSS □



selector, the button/input element. the button name (the value of the `name` argument) or the text that shows on the button. You can press any **BUTTON** element or **INPUT** of type `submit`, `reset` or `button`. The second argument is a callback, just like `clickLink`.

Of course, before submitting a form, you'll need to fill it with values. `fill` For text fields, use the `fill` function, which takes two arguments: selector and the field value. This time the selector can be a **CSS** selector, the input element, the field name (its `name` attribute), or the text that shows on the label associated with that field.

Zombie.js supports text fields, password fields, text areas, and also the new HTML 5 fields types like email, search and url.

The `fill` function returns a reference to the browser, so you can chain several functions together. Its sibling functions `check` and `uncheck` (for check boxes), `choose` (for radio buttons) and `select` (for drop downs) work the same way.

Let's combine all of that into one example:

```
// Fill in the form and submit.
browser.
  fill("#Your Name", "Arm Biter").
  fill("#Profession", "Living dead").
  select("#Born", "1968").
  uncheck("#Send me the newsletter").
  pressButton("#Sign me up", function(e, browser, status) {

    // Make sure we got redirected to thank you page.
    assert.equal(browser.location.pathname, "/thankyou");

  });
```

Read more [on the Browser API](#)

## Readiness

Zombie.js supports the following:

- HTML5 parsing and dealing with tag soups
- [DOM Level 3](#) implementation
- HTML5 form fields (`search`, `url`, etc)
- CSS3 Selectors with [some extensions](#)
- Cookies and [Web Storage](#)
- XMLHttpRequest in all its glory
- `setTimeout/setInterval`
- `pushState`, `popstate` and `hashchange` events
- Scripts that use `document.write`
- `alert`, `confirm` and `prompt`

## In The Family

[capybara-zombie](#) -- Capybara driver for zombie.js running on top of node.

[zombie-jasmine-spike](#) -- Spike project for trying out Zombie.js with Jasmine

[Vows BDD](#) -- A BDD wrapper for Vows, allowing for easy writing of tests in a given-when-then format

[Mink](#) -- PHP 5.3 acceptance test framework for web applications

## Reporting Glitches

**Step 1:** Run Zombie with debugging turned on, the trace will help figure out what it's doing. For example: ☐

```
var browser = new zombie.Browser({ debug: true });
browser.visit("http://thedeat", function(e, browser, status) {
  console.log(status, browser.errors);
  ...
});
```

**Step 2:** Wait for it to finish processing, then dump the current browser ☐ state:

```
browser.dump();
```

**Step 3:** If publicly available, include the URL of the page you're trying to access. Even better, provide a test script I can run from the Node.js console (similar to step 1 above).

Read more [about troubleshooting](#)

## Giving Back

- Find [assaf/zombie on Github](#)
- Fork the project
- Add tests
- Make your changes
- Send a pull request

Read more [about the guts of Zombie.js](#) and check out the outstanding [to-dos](#).

Follow announcements, ask questions on [the Google Group](#)

Get help on IRC: join [zombie.js on Freenode](#) or [web-based IRC](#)

## Brains

Zombie.js is copyright of [Assaf Arkin](#), released under the MIT License

Blood, sweat and tears of joy:

[Damian Janowski aka djanowski](#)

[JosÃ© Valim aka josevalim](#)

[Bob Lail boblail](#)

And all the fine people mentioned in [the changelog](#).

Zombie.js is written in [CoffeeScript](#) for [Node.js](#)

DOM emulation by Elijah Insua's [JSDOM](#)

HTML5 parsing by Aria Stewart's [HTML5](#)

CSS selectors by John Resig's [Sizzle.js](#)

XPath support using Google's [AJAXSLT](#)

Magical Zombie Girl by [Toho Scope](#)

## See Also

**zombie-api**(7), **zombie-troubleshoot**(7), **zombie-selectors**(7),  
**zombie-changelog**(7), **zombie-todo**(7)

Zombie.js brought to you by [very alive people](#).

# Zombie.js

## The Zombie API

### The Browser

#### **new zombie.Browser(options?) : Browser**

Creates and returns a new browser. A browser maintains state across requests: history, cookies, HTML 5 local and session storage. A browser has a main window, and typically a document loaded into that window.

You can pass options when initializing a new browser, for example:

```
browser = new zombie.Browser({ debug: true })
browser.runScripts = false
```

Or on existing browser for the duration of a page load:

```
zombie.visit("http://localhost:3000/", { debug: true, runScripts: false },
  function (e, browser, status) {
    ...
  });
```

You can also set options globally for all browsers to inherit:

```
Zombie.site = "http://localhost:3000"
Zombie.loadCSS = false
```

### Browser Options

You can use the following options:

- **credentials** -- Object containing authorization credentials.
- **debug** -- Have Zombie report what it's doing. Defaults to true if environment variable **DEBUG** is set.

- `loadCSS` -- Loads external stylesheets. Defaults to true.
- `runScripts` -- Run scripts included in or loaded from the page. Defaults to true.
- `userAgent` -- The User-Agent string to send to the server.
- `silent` -- If true, suppress all `console.log` output from scripts. You can still view it with `window.console.output`.
- `site` -- Base URL for all requests. If set, you can call `visit` with relative URL.
- `waitFor` -- Tells `wait` function how long to wait (in milliseconds) while timers fire. Defaults to 5 seconds.

## **browser.visit(url, callback)**

## **browser.visit(url, options, callback)**

Shortcut for creating new browser and calling `browser.visit` on it. If the second argument are options, initializes the browser with these options. See *Navigation* below for more information about the `visit` method.

## **browser.open() : Window**

Opens a new browser window.

## **browser.window : Window**

Returns the main window. A browser always has one window open.

## **browser.error : Error**

Returns the last error reported while loading this window.

## **browser.errors : Array**

Returns all errors reported while loading this window.

## **Document Content**

You can inspect the document content using the [DOM API](#) traversal

methods or the [DOM Selector API](#).

To find an element with ID "item-23":□

```
var item = document.getElementById("item-32");
```

For example, to find out the first input field with the name "email":□

```
var field = document.querySelector("input[name=email]);
```

To find out all the even rows in a table:□

```
var rows = table.querySelectorAll("tr:even");
```

CSS selectors support is provided by [Sizzle.js](#), the same engine used by jQuery. You're probably familiar with it, if not, check the [list of supported selectors](#).

## **browser.body : Element**

Returns the body element of the current document.

## **browser.document : Document**

Returns the main window's document. Only valid after opening a document (see `browser.visit`).

## **browser.evaluate(expr) : Object**

Evaluates a JavaScript expression in the context of the current window and returns the result. For example:

```
browser.evaluate("document.title");
```

## **browser.html(selector?, context?) : String**

Returns the HTML contents of the selected elements.

With no arguments returns the HTML contents of the document. This is one way to find out what the page looks like after executing a bunch of JavaScript.

With one argument, the first argument is a CSS selector evaluated against the document body. With two arguments, the CSS selector is evaluated against the element given as the context.

For example:

```
console.log(browser.html("#main"));
```

### **browser.queryAll(selector, context?) : Array**

Evaluates the CSS selector against the document (or context node) and return array of nodes. (Unlike `document.querySelectorAll` that returns a node list).

### **browser.query(selector, context?) : Element**

Evaluates the CSS selector against the document (or context node) and return an element.

### **browser.text(selector, context?) : String**

Returns the text contents of the selected elements.

With one argument, the first argument is a CSS selector evaluated against the document body. With two arguments, the CSS selector is evaluated against the element given as the context.

For example:

```
console.log(browser.text("title"));
```

### **browser.xpath(expression, context?) => XPathResult**

Evaluates the XPath expression against the document (or context node)



and return the XPath result. Shortcut for `document.evaluate`.

## Navigation

Zombie.js loads pages asynchronously. In addition, a page may require loading additional resources (such as JavaScript files) and executing various event handlers (e.g. `jQuery.onready`).

For that reason, navigating to a new page doesn't land you immediately on that page: you have to wait for the browser to complete processing of all events. You can do that by calling `browser.wait` or passing a callback to methods like `visit` and `clickLink`.

### **browser.clickLink(selector, callback)**

Clicks on a link. The first argument is the link text or CSS selector. Second argument is a callback, invoked after all events are allowed to run their course.

Zombie.js fires a `click` event and has a default event handler that will to the link's `href` value, just like a browser would. However, event handlers may intercept the event and do other things, just like a real browser.

For example:

```
browser.clickLink("View Cart", function(e, browser, status) {  
  assert.lengthOf(browser.queryAll("#cart .body"), 3);  
});
```

### **browser.link(selector) : Element**

Finds and returns a link (A) element. You can use a CSS selector or find a link by its text contents (case sensitive, but ignores leading/trailing spaces).

### **browser.location : Location**

Return the location of the current document (same as

`window.location`).

## **browser.location = url**

Changes document location, loading a new document if necessary (same as setting `window.location`). This will also work if you just need to change the hash (Zombie.js will fire a `hashchange` event), for example:

```
browser.location = "#bang";
browser.wait(function(e, browser) {
  // Fired hashchange event and did something cool.
  ...
});
```

## **browser.statusCode : Number**

Returns the status code returned for this page request (200, 303, etc).

## **browser.success : Boolean**

Returns true if the status code is 2xx.

## **browser.visit(url, callback)**

## **browser.visit(url, options, callback)**

Loads document from the specified URL, processes all events in the queue, and finally invokes the callback.

In the second form, sets the options for the duration of the request, and resets before passing control to the callback. For example:

```
browser.visit("http://localhost:3000", { debug: true },
  function(e, browser, status) {
    console.log("The page:", browser.html());
  }
);
```

## **browser.redirected : Boolean**

Returns true if the page request followed a redirect.

## **Forms**

Methods for interacting with form controls (e.g. `fill`, `check`) take a first argument that tries to identify the form control using a variety of approaches. You can always select the form control using an appropriate [CSS selector](#), or pass the element itself.

Zombie.js can also identify form controls using their name (the value of the `name` attribute) or using the text of the label associated with that control. In both case, the comparison is case sensitive, but to work flawlessly, ignores leading/trailing whitespaces when looking at labels.

If there are no event handlers, Zombie.js will submit the form just like a browser would, process the response (including any redirects) and transfer control to the callback function when done.

If there are event handlers, they will all be run before transferring control to the callback function. Zombie.js can even support jQuery live event handlers.

## **browser.attach(selector, filename, callback) : this**

Attaches a file to the specified input field. The second argument is the file name (you cannot attach streams).

Without callback, returns this.

## **browser.check(field, callback) : this**

Checks a checkbox. The argument can be the field name, label text or a CSS selector.

Without callback, returns this.

## **browser.choose(field, callback) : this**

Selects a radio box option. The argument can be the field name, label text or a CSS selector.

Without callback, returns this.

### **browser.field(selector) : Element**

Find and return an input field (INPUT, TEXTAREA or SELECT) based on a CSS selector, field name (its `name` attribute) or the text value of a label associated with that field (case sensitive, but ignores leading/trailing spaces).

### **browser.fill(field, value, callback) : this**

Fill in a field: input field or text area. The first argument can be the field name, label text or a CSS selector. The second argument is the field value.

For example:

```
browser.fill("Name", "ArmBiter").fill("Password", "Brains...")
```

Without callback, returns this.

### **browser.button(selector) : Element**

Finds a button using CSS selector, button name or button text (BUTTON or INPUT element).

### **browser.pressButton(selector, callback)**

Press a button. Typically this will submit the form, but may also reset the form or simulate a click, depending on the button type.

The first argument is either the button name, text value or CSS selector. Second argument is a callback, invoked after the button is pressed, form submitted and all events allowed to run their course.

For example:

```
browser.fill(email, "zombie@underworld.dead").
  pressButton("Sign me Up", function() {
    // All signed up, now what?
  });
```

Returns nothing.

### **browser.select(field, value, callback) : this**

Selects an option. The first argument can be the field name, label text or a CSS selector. The second value is the option to select, by value or label.

For example:

```
browser.select("Currency", "brains")
```

See also `selectOption`.

Without callback, returns this.

### **browser.selectOption(option, callback) : this**

Selects the option (an `OPTION` element).

Without callback, returns this.

### **browser.uncheck(field, callback) : this**

Unchecks a checkbox. The argument can be the field name, label text or a CSS selector.

Without callback, returns this.

### **browser.unselect(field, value, callback) : this**

Unselects an option. The first argument can be the field name, label text or a CSS selector. The second value is the option to unselect, by value or label.

You can use this (or `unselectOption`) when dealing with multiple selection.

Without callback, returns this.

### **browser.unselectOption(option, callback) : this**

Unselects the option (an `OPTION` element).

Without callback, returns this.

## **State Management**

The browser maintains state as you navigate from one page to another. Zombie.js supports both [cookies](#) and HTML5 [Web Storage](#).

Note that Web storage is specific to a host/port combination. Cookie storage is specific to a domain, typically a host, ignoring the port. □

### **browser.cookies(domain, path?) : Cookies**

Returns all the cookies for this domain/path. Path defaults to `"/"`.

For example:

```
browser.cookies("localhost").set("session", "567");
```

The `Cookies` object has the methods `clear()`, `get(name)`, `set(name, value)`, `remove(name)` and `dump()`.

The `set` method accepts a third argument which may include the options `expires`, `maxAge` and `secure`.

### **browser.fork() : Browser**

Return a new browser using a snapshot of this browser's state. This method clones the forked browser's cookies, history and storage. The two browsers are independent, actions you perform in one browser do not affect the other.

Particularly useful for constructing a state (e.g. sign in, add items to a shopping cart) and using that as the base for multiple tests, and for running parallel tests in Vows.

### **browser.loadCookies(String)**

Load cookies from a text string (e.g. previously created using `browser.saveCookies`).

### **browser.loadHistory(String)**

Load history from a text string (e.g. previously created using `browser.saveHistory`).

### **browser.loadStorage(String)**

Load local/session storage from a text string (e.g. previously created using `browser.saveStorage`).

### **browser.localStorage(host) : Storage**

Returns local Storage based on the document origin (hostname/port).

For example:

```
browser.localStorage("localhost:3000").setItem("session", "567");
```

The `Storage` object has the methods `key(index)`, `getItem(name)`, `setItem(name, value)`, `removeItem(name)`, `clear()` and `dump`. It also has the read-only property `length`.

### **browser.saveCookies() : String**

Save cookies to a text string. You can use this to load them back later on using `browser.loadCookies`.

### **browser.saveHistory() : String**

Save history to a text string. You can use this to load the data later on

using `browser.loadHistory`.

## **browser.saveStorage() : String**

Save local/session storage to a text string. You can use this to load the data later on using `browser.loadStorage`.

## **browser.sessionStorage(host) : Storage**

Returns session Storage based on the document origin (hostname/port). See `localStorage` above.

## **Interaction**

### **browser.onalert(fn)**

Called by `window.alert` with the message. If you just want to know if an alert was shown, you can also use `prompted` (see below).

### **browser.onconfirm(question, response)**

### **browser.onconfirm(fn)**

The first form specifies a canned response to return when `window.confirm` is called with that question. The second form will call the function with the question and use the response of the first function to return a value (true or false).

The response to the question can be true or false, so all canned responses are converted to either value. If no response available, returns false.

For example:

```
browser.onconfirm("Are you sure?", true)
```

### **browser.onprompt(message, response)**

### **browser.onprompt(fn)**



The first form specifies a canned response to return when `window.prompt` is called with that message. The second form will call the function with the message and default value and use the response of the first function to return a value or false.

The response to a prompt can be any value (converted to a string), false to indicate the user cancelled the prompt (returning null), or nothing to have the prompt return the default value or an empty string.

For example:

```
browser.onprompt(function(message) { return Math.random() })
```

## **browser.prompted(message) => boolean**

Returns true if user was prompted with that message by a previous call to `window.alert`, `window.confirm` or `window.prompt`.

## **Events**

Since events may execute asynchronously (e.g. XHR requests, timers), the browser maintains an event queue. Occasionally you will need to let the browser execute all the queued events before proceeding. This is done by calling `wait`, or one of the many methods that accept a callback.

In addition the browser is also an `EventEmitter`. You can register any number of event listeners to any of the emitted events.

## **browser.fire(name, target, callback?)**

Fires a DOM event. You can use this to simulate a DOM event, e.g. clicking a link or clicking the mouse. These events will bubble up and can be cancelled.

The first argument is the event name (e.g. `click`), the second argument is the target element of the event. With a callback, this method will transfer control to the callback after running all events.

## **browser.wait(callback)**

## **browser.wait(duration, callback)**

Waits for the browser to complete loading resources and processing JavaScript events. When done, calls the callback with null and browser.

With **duration** as the first argument, this method waits for the specified time (in milliseconds) and any resource/JavaScript to complete processing.

Without duration, Zombie makes best judgement by waiting up to 5 seconds for the page to load resources (scripts, XHR requests, iframes), process DOM events, and fire timeouts events.

You can also call **wait** with no callback and simply listen to the **done** and **error** events getting fired.

### **Event: 'done'**

```
function (browser) { }
```

Emitted whenever the event queue goes back to empty.

### **Event: 'error'**

```
function (error) { }
```

Emitted if an error occurred loading a page or submitting a form.

### **Event: 'loaded'**

```
function (browser) { }
```

Emitted whenever new page loaded. This event is emitted before DOMContentLoaded.

## **Debugging**

When trouble strikes, refer to these functions and the [troubleshooting](#)

[guide](#).

## **browser.dump()**

Dump information to the console: Zombie version, current URL, history, cookies, event loop, etc. Useful for debugging and submitting error reports.

## **browser.lastError : Object**

Returns the last error received by this browser in lieu of response.

## **browser.lastRequest : Object**

Returns the last request sent by this browser.

## **browser.lastResponse : Object**

Returns the last response received by this browser.

## **browser.log(arguments)**

## **browser.log(function)**

Call with multiple arguments to spit them out to the console when debugging enabled (same as `console.log`). Call with function to spit out the result of that function call when debugging enabled.

## **browser.resources : Object**

Returns a list of resources loaded by the browser.

## **browser.viewInBrowser(name?)**

Views the current document in a real Web browser. Uses the default system browser on OS X, BSD and Linux. Probably errors on Windows.

Zombie.js brought to you by [very alive people](#).



# Zombie.js

## CSS Selectors

Zombie.js uses [Sizzle.js](#) which provides support for most [CSS 3 selectors](#) with a few useful extension.

Sizzle.js is the selector engine used in jQuery, so if you're familiar with jQuery selectors, you're familiar with Sizzle.js.

The following list summarizes which selectors are currently supported:

\* Any element

E An element of type E

E#myid An E element with ID equal to "myid"

E.foo An E element whose class is "foo"

E[foo] An E element with a "foo" attribute

E[foo="bar"] An E element whose "foo" attribute value is exactly equal to "bar"

E[foo!="bar"] An E element whose "foo" attribute value does not equal to "bar"

E[foo~="bar"] An E element whose "foo" attribute value is a list of whitespace-separated values, one of which is exactly equal to "bar"

E[foo^="bar"] An E element whose "foo" attribute value begins exactly with the string "bar"

E[foo\$="bar"] An E element whose "foo" attribute value ends exactly with the string "bar"

E[foo\*="bar"] An E element whose "foo" attribute value contains the substring "bar"

**E[foo="en"]** An E element whose "foo" attribute has a hyphen-separated list of values beginning (from the left) with "en"

**E:nth-child(n)** An E element, the n-th child of its parent

**E:first-child** An E element, first child of its parent□

**E:last-child** An E element, last child of its parent

**E:only-child** An E element, only child of its parent

**E:empty** An E element that has no children (including text nodes)

**E:link** A link

**E:focus** An E element during certain user actions

**E:enabled** A user interface element E which is enabled

**E:disabled** A user interface element E which is disabled

**E:checked** A user interface element E which is checked (for instance a radio-button or checkbox)

**E:input** An E element that is an input element (includes **textarea**, **select** and **button**)

**E:text** An E element that is an input text field or text area□

**E:checkbox** An E element that is an input checkbox

**E:file** An E element that is an input file□

**E:password** An E element that is an input password

**E:submit** An E element that is an input or button of type **submit**

**E:image** An E element that is an input of type **image**

**E:button** An E element that is an input or button of type **button**

**E:reset** An E element that is an input or button of type **reset**

**E:header** An header element, one of h1, h2, h3, h4, h5, h6

**E:parent** A parent element, an element that contains another element

**E:not(s)** An E element that does not match the selector **s** (multiple selectors supported)

**E:contains(t)** An E element whose textual contents contains **t** (case sensitive)

**E:first** An E element whose position on the page is first in document order

**E:last** An E element whose position on the page is last in document order

**E:even** An E element whose position on the page is even numbered (counting starts at 0)

**E:odd** An E element whose position on the page is odd numbered (counting starts at 0)

**E:eq(n)/:nth(n)** An E element whose Nth element on the page (e.g. **:eq(5)**)

**E:lt(n)** An E element whose position on the page is less than **n**

**E:gt(n)** An E element whose position on the page is less than **n**

**E F** An F element descendant of an E element

**E > F** An F element child of an E element

**E + F** An F element immediately preceded by an E element

**E ~ F** An F element preceded by an E element

Zombie.js brought to you by [very alive people](#).

# Zombie.js

## Troubleshooting guide

### The Dump

Get the browser to dump its current state. You'll be able to see the current document URL, history, cookies, local/session storage, and portion of the current page:

```
browser.dump()
```

URL: http://localhost:3003/here/##there

History:

1. http://localhost:3003/here
- 2: http://localhost:3003/here/##there

Cookies:

session=e62ab205; domain=localhost; path=/here

Storage:

localhost:3003 session:  
day = Monday

Document:

```
<html>
  <head>
    <script src="/jquery.js"></script>
    <script src="/sammy.js"></script>
    <script src="/app.js"></script>
  </head>
  <body>
    ...
```



The actual report will have much more information.

## Debugging

When running in debug mode, `Zombie.js` will spit out messages to the console. These could help you see what's going on as your tests execute, especially useful around stuff that happens in the background, like XHR requests.

To turn debugging on/off set `browser.debug` to true/false. You can also set this option when creating a new `Browser` object (the constructor takes an options argument), or for the duration of a single call to `visit` (the second argument being the options).

For example:

```
zombie.visit("http://thedead", { debug: true}, function(err, browser) {  
  console.log(browser.errors);  
  ...  
});
```

If you're working on the code and you want to add more debug statements, call `browser.log` with any sequence of arguments (same as `console.log`), or with a function. In the later case, it will call the function only when debugging is turned on, and spit the value returned from the console.

For example:

```
browser.log("Currently visiting", browser.location);  
browser.log(function() {  
  return "Currently visiting " + browser.location;  
});
```

## Request/response

Each window keeps a trail of every resource request it makes (to load the page itself, scripts, XHR requests, etc). You can inspect these by

obtaining the `window.resources` array and looking into it.

For example:

```
browser.resources.dump()
```

The browser object provides the convenient methods `lastRequest`, `lastResponse` and `lastError` that return, respectively, the request, response and error associated with the last resources loaded by the current window.

Zombie.js brought to you by [very alive people](#).