Zombie.js

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

Note: This page documents the older Zombie 1.4, check here for Zombie 2.0 docs. The new docs are still work in progress, please help make them better.

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

// Load the page from localhost
browser = new Browser()
browser.visit("http://localhost:3000/", function () {

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

      // Form submitted, new page loaded.
      assert.ok(browser.success);
      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, HTML5 local and session stroage, etc. A browser has a main window, and typically a document loaded into that window.

You can create a new `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 `Browser.visit` function with a URL and callback:

```
Browser.visit("http://localhost:3000/", function (e, browser) {
  // The browser argument is an instance of Browser class
  ...
})
```

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 Mocha

Zombie also supports promises. When you call functions like `visit`, `wait` or `clickLink` without a callback, you get a promise. After the browser is done processing, it either fulfills or rejects the promise.

For example:

```
browser.visit("http://localhost:3000/").
  then(function() {
    assert.equal(browser.text("H1"), "Deferred zombies");
  }).
  fail(function(error) {
    console.log("Oops", error);
  });
```

Another way to simplify your code is to catch all errors from one place using events, for example:

```
browser.on("error", function(error) {
  console.error(error);
})
browser.visit("http://localhost:3000/").
```

```
  then(function() {
    assert.equal(browser.text("H1"), "Deferred zombies");
    // Chaining works by returning a promise here
    return browser.clickLink("Hit me");
  }).
  then(function() {
    assert.equal(browser.text("H1"), "Ouch");
  });
```

Most errors that occur â€“ resource loading and JavaScript execution â€“ are not fatal, so rather the stopping processing, they are collected in `browser.errors`. For example:

```
browser.visit("http://localhost:3000/", function () {
  assert.ok(browser.success);
  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. You can also pass a function that would return true when done.

Otherwise, Zombie makes best judgement by waiting for about half a second 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. 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 HTML5 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() {

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

  });
```

Read more on the Browser API

Believing

Here are some guidelines for writing tests using Zombie, promises and Mocha.

Let's start with a simple example:

```
describe("visit", function() {
  before(function(done) {
    this.browser = new Browser();
    this.browser
      .visit("/promises")
      .then(done, done);
  });

  it("should load the promises page", function() {
    assert.equal(this.browser.location.pathname, "/promises");
  });
});
```

The call to `visit` returns a promise. Once the page loads successfully, the promise will resolve and call the first callback (`done`) with no arguments. This will run the test and evaluate the assertion. Success.

If there's an error, the promise fails and calls the second callback (also `done`) with an error. Calling it with an `Error` argument causes Mocha to fail the test.

Now let's chain promises together:

```
browser
  .visit("/promises") // Step 1, open a page
  .then(function() {
    // Step 2, fill-in the form
    browser.fill("Email", "armbiter@example.com");
    browser.fill("Password", "b100d");
  })
  .then(function() {
    // Step 3, resolve the next promise with this value.
    return "OK";
  })
```

```
  .then(function(value) {
    // Step 4, previous step got us to resolve with this value.
    assert.equal(value, "OK");
  })
  .then(function() {
    // Step 5, click the button and wait for something to happen
    // by returning another promise.
    return browser.pressButton("Let me in");
  })
  .then(done, done);
```

The first step is easy, it loads a page and returns a promise. When that promise resolves, it calls the function of the second step which fills the two form fields. That step is itself a promise that resolves with no value.

The third step follows, and here we simply return a value. As a result, the next promise will resolve with that value, as you can see in the fourth step. Another way to think about it is: step four is chained to a new promise with the value "OK".

On to step five where we press the button, which submits the form and loads the response page. All of that happens after `pressButton`, so we want to wait for it before moving to the sixth and last step.

Luckily, `pressButton` - just like `wait` - returns a promise which gets fulfilled after the browser is done processing events and loading resources. By returning this new promise, we cause the next step to wait for this promise to resolve.

In short: the very last step is chained to a new promise returned by `pressButton`. You can use this pattern whenever you need to wait, after `visit`, `reload`, `clickLink`, etc.

Note: In CoffeeScript a function that doesn't end with explicit `return` statement would return the value of the last statement. If you're seeing promises resolved with unexpected values, you may need to end your function with a `return`.

In real life the ability to chain promises helps us structure complex scenarios out of reusable steps. Like so:

```
browser
  .visit("/promises")
  .then( fillInName.bind(browser) )
  .then( fillInAddress.bind(browser) )
  .then( fillInCreditCard.bind(browser) )
```

```
  .then(function() {
    browser.pressButton("Buy it!")
  })
  .then(done, done);
```

Note: This usage of `bind` is one way to allow a function defined in another context to use the `Browser` object available in this context.

Let's talk about error handling. In promise-land, an error causes the promise to be rejected. However, errors are not re-thrown out of the promise, and so this code will fail silently:

```
browser
  .visit("/promises")
  .then(function() {
    // This throws an error, which gets caught and rejects
    // the promise.
    assert(false, "I fail!");
  })
  .then(done);
```

Rejection may not be fun, but you've got to deal with it.

When a promise gets rejected, that rejection travels down the chain, so you only need to catch it at the very end. The examples we started with do that by calling `then` with the same callback for handling the resolved and rejected cases.

If your test case expects an error to happen, write it like this:

```
browser
  .visit("/promises")
  .then(function() {
    assert(false, "I fail!")
  })
  .fail(function(error) {
    // Error happened, test is done.  Otherwise, done never
    // gets called and Mocha will fail this test.
    done();
  });
```

Another way of dealing with errors:

```
before(function(done) {
  this.browser = new Browser();
  this.browser
    .visit("/no-such-page")
    .finally(done);
});

it("should report an error", function() {
  assert(this.browser.error);
})
```

Unlike `then`, the `finally` callback gets called on either success or failure and with no value.

Read more about promises.

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
- `alert`, `confirm` and `prompt`
- WebSockets and Server-Sent Events

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

Mocha -- mocha - simple, flexible, fun javascript test framework for node.js & the browser. (BDD, TDD, QUnit styles via interfaces)

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:

```
Browser.debug = true
var browser = new Browser()
browser.visit("http://thedead", function() {
  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

Blood, sweat and tears of joy:

Bob Lail boblail

Brian McDaniel

Damian Janowski aka djanowski

JosÃ© Valim aka josevalim

Justin Latimer

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

JavaScript execution contexts using Contextify

HTTP(S) requests using Request

Cookie support using Tough Cookie

Promises support via Q

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.