

# Zombie.js

## Browser

### **browser.assert**

Methods for making assertions against the browser, such as `browser.assert.element(".foo")`.

See [Assertions](#) for detailed discussion.

### **browser.console**

Provides access to the browser console (same as `window.console`).

### **browser.referer**

You can use this to set the HTTP Referer header.

### **browser.resources**

Access to history of retrieved resources. Also provides methods for retrieving resources and managing the resource pipeline. When things are not going your way, try calling `browser.resources.dump()`.

See [Resources](#) for detailed discussion.

### **browser.tabs**

Array of all open tabs (windows). Allows you to operate on more than one open window at a time.

See [Tabs](#) for detailed discussion.

### **browser.eventLoop**

### **browser.errors**

## Extending The Browser

```
Browser.extend(function(browser) {
  browser.on("console", function(level, message) {
    logger.log(message);
  });
  browser.on("log", function(level, message) {
    logger.log(message);
  });
});
```

## Tabs

Just like your favorite Web browser, Zombie manages multiple open windows as tabs. New browsers start without any open tabs. As you visit the first page, Zombie will open a tab for it.

All operations against the `browser` object operate on the currently active tab (window) and most of the time you only need to interact with that one tab. You can access it directly via `browser.window`.

Web pages can open additional tabs using the `window.open` method, or whenever a link or form specifies a target (e.g. `target=_blank` or `target=window-name`). You can also open additional tabs by calling `browser.open`. To close the currently active tab, close the window itself.

You can access all open tabs from `browser.tabs`. This property is an associative array, you can access each tab by its index number, and iterate over all open tabs using functions like `forEach` and `map`.

If a window was opened with a name, you can also access it by its name. Since names may conflict with reserved properties/methods, you may need to use `browser.tabs.find`.

The value of a tab is the currently active window. That window changes when you navigate forwards and backwards in history. For example, if you visited the URL `"/foo"` and then the URL `"/bar"`, the first tab (`browser.tabs[0]`) would be a window with the document from `"/bar"`. If you then navigate back in history, the first tab would be the window with the document `"/foo"`.

The following operations are used for managing tabs:

### **browser.close(window)**

Closes the tab with the given window.

### **browser.close()**

Closes the currently open tab.

### **browser.tabs**

Returns an array of all open tabs.

### **browser.tabs[number]**

Returns the tab with that index number.

### **browser.tabs[string]**

### **browser.tabs.find(string)**

Returns the tab with that name.

### **browser.tabs.closeAll()**

Closes all tabs.

### **browser.tabs.current**

This is a read/write property. It returns the currently active tab.

Can also be used to change the currently active tab. You can set it to a window (e.g. as currently returned from `browser.current`), a window name or the tab index number.

### **browser.tabs.dump(output)**

Dump a list of all open tabs to standard output, or the output stream.

### **browser.tabs.index**

Returns the index of the currently active tab.

### **browser.tabs.length**

Returns the number of currently opened tabs.

### **browser.open(url: "http://example.com")**

Opens and returns a new tab. Supported options are: - name - Window name. - url - Load document from this URL.

### **browser.window**

Returns the currently active window, same as `browser.tabs.current`.

## **Assertions**

To make life easier, Zombie introduces a set of convenience assertions that you can access directly from the browser object. For example, to check that a page loaded successfully:

```
browser.assert.success();
browser.assert.text("title", "My Awesome Site");
browser.assert.element("#main");
```

These assertions are available from the `browser` object since they operate on a particular browser instance -- generally dependent on the currently open window, or document loaded in that window.

Many assertions require an `element`/`elements` as the first argument, for example, to compare the text content (`assert.text`), or attribute value (`assert.attribute`). You can pass one of the following values:

- An HTML element or an array of HTML elements
- A CSS selector string (e.g. "h2", ".book", "#first-name")

Many assertions take an expected value and compare it against the actual value. For example, `assert.text` compares the expected value against the text contents of one or more strings. The expected value can be one of:

- A JavaScript primitive value (string, number)
- `undefined` or `null` are used to assert the lack of value
- A regular expression
- A function that is called with the actual value and returns true if the assertion is true
- Any other object will be matched using `assert.deepEqual`

Note that in some cases the DOM specification indicates that lack of value is an empty string, not `null`/`undefined`.

All assertions take an optional last argument that is the message to show if the assertion fails. Better yet, use a testing framework like [Mocha](#) that has good diff support and don't worry about these messages.

## Available Assertions

The following assertions are available:

### **`assert.attribute(selection, name, expected, message)`**

Asserts the named attribute of the selected element(s) has the expected value.

Fails if no element found.

### **`assert.className(selection, className, message)`**

Asserts that selected element(s) has that and only that class name. May also be space-separated list of class names.

Fails if no element found.

### **`assert.cookie(name, expected, message)`**

Asserts that a cookie with the given name has the expected value.

### **`assert.element(selection, message)`**

Asserts that one element matching selection exists.

Fails if no element or more than one matching element are found.

### **assert.elements(selection, count, message)**

Asserts how many elements exist in the selection.

The argument `count` can be a number, or an object with the following properties:

- `atLeast` - Expecting to find at least that many elements
- `atMost` - Expecting to find at most that many elements
- `exactly` - Expecting to find exactly that many elements

### **assert.evaluate(expression, expected, message)**

Evaluates the JavaScript expression in the context of the currently open window.

With one argument, asserts that the value is equal to `true`.

With two/three arguments, asserts that the returned value matches the expected value.

### **assert.global(name, expected, message)**

Asserts that the global (window) property has the expected value.

### **assert.hasClass(selection, className, message)**

Asserts that selected element(s) have the expected class name. Elements may have other class names (unlike `assert.className`).

Fails if no element found.

### **assert.hasFocus(selection, message)**

Asserts that selected element has the focus.

If the first argument is `null`, asserts that no element has the focus.

Otherwise, fails if element not found, or if more than one element found.

### **assert.input(selection, expected, message)**

Asserts that selected input field(s) (`input`, `textarea`, `select` etc) have the expected value.

Fails if no element found.

### **assert.hasNoClass(selection, className, message)**

Asserts that selected element(s) does not have the expected class name. Elements may have other class names (unlike `assert.className`).

Fails if no element found.

### **assert.prompted(messageShown, message)**

Asserts the browser prompted with a given message.

### **assert.redirected(message)**

Asserts the browser was redirected when retrieving the current page.

### **assert.success(message)**

Asserts the current page loaded successfully (status code 2xx or 3xx).

### **assert.status(code, message)**

Asserts the current page loaded with the expected status code.

### **assert.style(selection, style, expected, message)**

Asserts that selected element(s) have the expected value for the named style property. For example:

```
browser.assert.style(".navigation", "opacity", 0.5)
```

Fails if no element found.

### **assert.text(selection, expected, message)**

Asserts that selected element(s) have the expected text content. For example:

```
browser.assert.text("title", "My Awesome Page")
```

Fails if no element found.

### **assert.url(url, message)**

Asserts the current page has the expected URL.

The expected URL can be one of:

- The full URL as a string
- A regular expression
- A function, called with the URL and returns true if the assertion is true
- An object, in which case individual properties are matched against the URL

For example:

```
browser.assert.url("http://localhost/foo/bar")
browser.assert.url({ pathame: "/foo/bar" });
browser.assert.url({ query: { name: "joedoe" } });
```

## **Roll Your Own Assertions**

Not seeing an assertion you want? You can add your own assertions to the prototype of `Browser.Assert`.

For example:

```
// Asserts the browser has the expected number of open tabs.
Browser.Assert.prototype.openTabs = function(expected, message) {
  assert.equal(this.browser.tabs.length, expected, message);
};
```

Or application specific:

```
// Asserts which links is highlighted in the navigation bar
Browser.Assert.navigationOn = function(linkText) {
  this.assert.element(".navigation-bar");
  this.assert.text(".navigation-bar a.highlighted", linkText);
};
```

## Events

Each browser instance is an `EventEmitter`, and will emit a variety of events you can listen to.

Some things you can do with events:

- Trace what the browser is doing, e.g. log every page loaded, every DOM event emitted, every timeout fired
- Wait for something to happen, e.g. form submitted, link clicked, input element getting the focus
- Strip out code from HTML pages, e.g. remove analytics code when running tests
- Add event listeners to the page before any JavaScript executes
- Mess with the browser, e.g. modify loaded resources, capture and change DOM events

### `console (level, message)`

Emitted whenever a message is printed to the console (`console.log`, `console.error`, `console.trace`, etc).

The first argument is the logging level, and the second argument is the message.

The logging levels are: `debug`, `error`, `info`, `log`, `trace` and `warn`.

### `active (window)`

Emitted when this window becomes the active window.

### `closed (window)`

Emitted when this window is closed.

## **done ()**

Emitted when the event loop goes empty.

## **evaluated (code, result, filename)**

Emitted after JavaScript code is evaluated.

The first argument is the JavaScript function or code (string). The second argument is the result. The third argument is the filename.

## **event (event, target)**

Emitted whenever a DOM event is fired on the target element, document or window.

## **focus (element)**

Emitted whenever an element receives the focus.

## **inactive (window)**

Emitted when this window is no longer the active window.

## **interval (function, interval)**

Emitted whenever an interval (`setInterval`) is fired.

The first argument is the function or code to evaluate, the second argument is the interval in milliseconds.

## **link (url, target)**

Emitted when a link is clicked.

The first argument is the URL of the new location, the second argument identifies the target window (`_self`, `_blank`, window name, etc).

## **loaded (document)**

Emitted when a document has been loaded into a window or frame.

This event is emitted after the HTML is parsed, and some scripts executed.

## **loading (document)**

Emitted when a document is about to be loaded into a window or frame.

This event is emitted when the document is still empty, before parsing any HTML.

## **opened (window)**

Emitted when a new window is opened.

## **redirect (request, response)**

Emitted when following a redirect.

The first argument is the request, the second argument is the response that caused the redirect. See [Resources](#) for more details.

The URL of the new resource to retrieve is given by `response.url`.

### **request (request)**

Emitted before making a request to retrieve a resource.

The first argument is the request object. See [Resources](#) for more details.

### **response (request, response)**

Emitted after receiving the response (excluding redirects).

The first argument is the request object, the second argument is the response object. See [Resources](#) for more details.

### **submit (url, target)**

Emitted whenever a form is submitted.

The first argument is the URL of the new location, the second argument identifies the target window (`_self`, `_blank`, window name, etc).

### **timeout (function, delay)**

Emitted whenever a timeout (`setTimeout`) is fired.

The first argument is the function or code to evaluate, the second argument is the delay in milliseconds.

## **Resources**

Zombie can retrieve with resources - HTML pages, scripts, XHR requests - over HTTP, HTTPS and from the file system.

Most work involving resources is done behind the scenes, but there are few notable features that you'll want to know about. Specifically, if you need to do any of the following:

- Inspect the history of retrieved resources, useful for troubleshooting issues related to resource loading
- Simulate a failed server
- Change the order in which resources are retrieved, or otherwise introduce delays to simulate a real world network
- Mock responses from servers you don't have access to, or don't want to access from test environment
- Request resources directly, but have *Zombie* handle cookies, authentication, etc
- Implement new mechanism for retrieving resources, for example, add new protocols or support new headers

## The Resources List

Each browser provides access to its resources list through `browser.resources`.

The resources list is an array of all resources requested by the browser. You can iterate and manipulate it just like any other JavaScript array.

Each resource provides four properties:

- `request` - The request object
- `response` - The resource object (if received)
- `error` - The error received instead of response
- `target` - The target element or document (when loading HTML page, script, etc)

The request object consists of:

- `method` - HTTP method, e.g. "GET"
- `url` - The requested URL
- `headers` - All request headers
- `body` - The request body can be `Buffer` or `string`; only applies to POST and PUT methods
- `multipart` - Used instead of a body to support file upload
- `time` - Timestamp when request was made
- `timeout` - Request timeout (0 for no timeout)

The response object consists of:

- `url` - The actual URL of the resource; different from request URL if there were any redirects
- `statusCode` - HTTP status code, eg 200
- `statusText` - HTTP status code as text, eg "OK"
- `headers` - All response headers
- `body` - The response body, may be `Buffer` or string, depending on the content type encoding
- `redirects` - Number of redirects followed (0 if no redirects)
- `time` - Timestamp when response was completed

Request for loading pages and scripts include the target DOM element or document. This is used internally, and may also give you more insight as to why a request is being made.

## Mocking, Failing and Delaying Responses

To help in testing, `Zombie` includes some convenience methods for mocking, failing and delaying responses.

For example, to mock a response:

```
browser.resources.mock("http://3rd.party.api/v1/request", {
  statusCode: 200,
  headers:    { "ContentType": "application/json" },
  body:      JSON.stringify({ "count": 5 })
})
```

In the real world, servers and networks often fail. You can test for these conditions by asking `Zombie` to simulate a failure. For example:

```
browser.resource.fail("http://3rd.party.api/v1/request");
```

Another issue you'll encounter in real-life applications are network latencies. When running tests, `Zombie` will request resources in the order in which they appear on the page, and likely receive them from the local server in that same order.

Occasionally you'll need to force the server to return resources in a different order, for example, to check what happens when script A loads after script B. You can introduce a delay into any response as simple as:

```
browser.resources.delay("http://3d.party.api/v1/request", 50);
```

## The Pipeline

`Zombie` uses a pipeline to operate on resources. You can extend that pipeline with your own set of handlers, for example, to support additional protocols, content types, special handlers, better resource mocking, etc.

The pipeline consists of a set of handlers. There are two types of handlers:

Functions with two arguments deal with requests. They are called with the request object and a callback, and must call that callback with one of:

- No arguments to pass control to the next handler
- An error to stop processing and return that error
- `null` and the response object to return that response

Functions with three arguments deal with responses. They are called with the request object, response object and a callback, and must call that callback with one of:

- No arguments to pass control to the next handler
- An error to stop processing and return that error

To add a new handle to the end of the pipeline:

```
browser.resources.addHandler(function(request, next) {
  // Let's delay this request by 1/10th second
  setTimeout(function() {
    Resources.httpRequest(request, next);
  }, Math.random() * 100);
});
```

If you need anything more complicated, you can access the pipeline directly via `browser.resources.pipeline`.

You can add handlers to all browsers via `Browser.Resources.addHandler`. These handlers are automatically added to every new `browser.resources` instance.

```
Browser.Resources.addHandler(function(request, response, next) {
  // Log the response body
  console.log("Response body: " + response.body);
  next();
});
```

When handlers are executed, `this` is set to the browser instance.

## Operating On Resources

If you need to retrieve or operate on resources directly, you can do that as well, using all the same features available to `Zombie`, including mocks, cookies, authentication, etc.

### `resources.addHandler(handler)`

Adds a handler to the pipeline of this browser instance. To add a handler to the pipeline of every browser instance, use `Browser.Resources.addHandler`.

### `resources.delay(url, delay)`

Retrieve the resource with the given URL, but only after a delay.

### **resources.dump(output)**

Dumps the resources list to the output stream (defaults to standard output stream).

### **resources.fail(url, error)**

Do not attempt to retrieve the resource with the given URL, but act as if the request failed with the given message.

This is used to simulate network failures (can't resolve hostname, can't make connection, etc). To simulate server failures (status codes 5xx), use `resources.mock`.

### **resources.pipeline**

Returns the current pipeline (array of handlers) for this browser instance.

### **resources.get(url, callback)**

Retrieves a resource with the given URL and passes response to the callback.

For example:

```
browser.resources.get("http://some.service", function(error, response) {
  console.log(response.statusText);
  console.log(response.body);
});
```

### **resources.mock(url, response)**

Do not attempt to retrieve the resource with the given URL, but return the response object instead.

### **resources.post(url, options, callback)**

Posts a document to the resource with the given URL and passes the response to the callback.

Supported options are:

- `body` - Request document body
- `headers` - Headers to include in the request
- `params` - Parameters to pass in the document body
- `timeout` - Request timeout in milliseconds (0 or `null` for no timeout)

For example:

```
var params = { "count": 5 };
browser.resources.post("http://some.service", { params: params },
    . . .
});

var headers = { "Content-Type": "application/x-www-form-urlencoded" };
browser.resources.post("http://some.service", { headers: headers,
    . . .
});
```

### **resources.request(method, url, options, callback)**

Makes an HTTP request to the resource and passes the response to the callback.

Supported options are:

- `body` - Request document body
- `headers` - Headers to include in the request
- `params` - Parameters to pass in the query string (GET, DELETE) or document body (POST, PUT)
- `timeout` - Request timeout in milliseconds (0 or null for no timeout)

For example:

```
browser.resources.request("DELETE", "http://some.service", function
    . . .
});
```

### **resources.restore(url)**

Reset any special resource handling from a previous call to `delay`, `fail` or `mock`.