# 

jsdom

jsdom is a pure-JavaScript implementation of many web standards, notably the WHATWG [DOM](https://dom.spec.whatwg.org/) and [HTML](https://html.spec.whatwg.org/multipage/) Standards, for use with Node.js. In general, the goal of the project is to emulate enough of a subset of a web browser to be useful for testing and scraping real-world web applications. The latest versions of jsdom require Node.js v6 or newer. (Versions of jsdom below v10 still work with Node.js v4, but are unsupported.) As of v10, jsdom has a new API (documented below). The old API is still supported for now; [see its documentation](./lib/old-api.md) for details. ## Basic usage ```js const jsdom = require("jsdom"); const { JSDOM } = jsdom; ``` To use jsdom, you will primarily use the `JSDOM` constructor, which is a named export of the jsdom main module. Pass the constructor a string. You will get back a `JSDOM` object, which has a number of useful properties, notably `window`: ```js const dom = new JSDOM(`

Hello world

`); console.log(dom.window.document.querySelector("p").textContent); // "Hello world" ``` (Note that jsdom will parse the HTML you pass it just like a browser does, including implied ``, ``, and `` tags.) The resulting object is an instance of the `JSDOM` class, which contains a number of useful properties and methods besides `window`. In general it can be used to act on the jsdom from the "outside," doing things that are not possible with the normal DOM APIs. For simple cases, where you don't need any of this functionality, we recommend a coding pattern like ```js const { window } = new JSDOM(`...`); // or even const { document } = (new JSDOM(`...`)).window; ``` Full documentation on everything you can do with the `JSDOM` class is below, in the section "`JSDOM` Object API". ## Customizing jsdom The `JSDOM` constructor accepts a second parameter which can be used to customize your jsdom in the following ways. ### Simple options ```js const dom = new JSDOM(``, { url: "https://example.org/", referrer: "https://example.com/", contentType: "text/html", userAgent: "Mellblomenator/9000", includeNodeLocations: true, storageQuota: 10000000 }); ``` - `url` sets the value returned by `window.location`, `document.URL`, and `document.documentURI`, and affects things like resolution of relative URLs within the document and the same-origin restrictions and referrer used while fetching subresources. It defaults to `"about:blank"`. - `referrer` just affects the value read from `document.referrer`. It defaults to no referrer (which reflects as the empty string). - `contentType` affects the value read from `document.contentType`, and how the document is parsed: as HTML or as XML. Values that are not `"text/html"` or an [XML mime type](https://html.spec.whatwg.org/multipage/infrastructure.html#xml-mime-type) will throw. It defaults to `"text/html"`. - `userAgent` affects the value read from `navigator.userAgent`, as well as the `User-Agent` header sent while fetching subresources. It defaults to \`Mozilla/5.0 (${process.platform}) AppleWebKit/537.36 (KHTML, like Gecko) jsdom/${jsdomVersion}\`. - `includeNodeLocations` preserves the location info produced by the HTML parser, allowing you to retrieve it with the `nodeLocation()` method (described below). It also ensures that line numbers reported in exception stack traces for code running inside ` `); // The script will not be executed, by default: dom.window.document.body.children.length === 1; ``` To enable executing scripts inside the page, you can use the `runScripts: "dangerously"` option: ```js const dom = new JSDOM(` `, { runScripts: "dangerously" }); // The script will be executed and modify the DOM: dom.window.document.body.children.length === 2; ``` Again we emphasize to only use this when feeding jsdom code you know is safe. If you use it on arbitrary user-supplied code, or code from the Internet, you are effectively running untrusted Node.js code, and your machine could be compromised. If you want to execute \_external\_ scripts, included via ` ``` If you do not control the page, you could try workarounds such as polling for the presence of a specific element. For more details, see the discussion in [#640](https://github.com/tmpvar/jsdom/issues/640), especially [@matthewkastor](https://github.com/matthewkastor)'s [insightful comment](https://github.com/tmpvar/jsdom/issues/640#issuecomment-22216965). ### Shared constructors and prototypes At the present time, for most web platform APIs, jsdom shares the same class definition between multiple seemingly-independent jsdoms. That means that, for example, the following situation can occur: ```js const dom1 = new JSDOM(); const dom2 = new JSDOM(); dom1.window.Element.prototype.expando = "blah"; console.log(dom2.window.document.createElement("frameset").expando); // logs "blah" ``` This is done mainly for performance and memory reasons: creating separate copies of all the many classes on the web platform, each time we create a jsdom, would be rather expensive. Nevertheless, we remain interested in one day providing an option to create an "independent" jsdom, at the cost of some performance. ### Missing features in the new API Compared to the old jsdom API from v9.x and before, the new API is noticeably missing fine-grained control of resource loads. Previous versions of jsdom allowed you to set options that were used when making requests (both for the initial request, in the old equivalent of `JSDOM.fromURL()`, and for subresource requests). They also allowed you to control which subresources were requested and applied to the main document, so that you could e.g. download stylesheets but not scripts. Finally, they provided a customizable resource loader that let you intercept any outgoing request and fulfill it with a completely synthetic response. None of these features are yet in the new jsdom API, although we are hoping to add them back soon! This requires a decent amount of behind-the-scenes work to implement in a reasonable way, unfortunately. In the meantime, please feel free to use the old jsdom API to get access to this functionality. It is supported and maintained, although it will not be getting new features. The documentation is found in [lib/old-api.md](./lib/old-api.md). ### Unimplemented parts of the web platform Although we enjoy adding new features to jsdom and keeping it up to date with the latest web specs, it has many missing APIs. Please feel free to file an issue for anything missing, but we're a small and busy team, so a pull request might work even better. Beyond just features that we haven't gotten to yet, there are two major features that are currently outside the scope of jsdom. These are: - \*\*Navigation\*\*: the ability to change the global object, and all other objects, when clicking a link or assigning `location.href` or similar. - \*\*Layout\*\*: the ability to calculate where elements will be visually laid out as a result of CSS, which impacts methods like `getBoundingClientRects()` or properties like `offsetTop`. Currently jsdom has dummy behaviors for some aspects of these features, such as sending a "not implemented" `"jsdomError"` to the virtual console for navigation, or returning zeros for many layout-related properties. Often you can work around these limitations in your code, e.g. by creating new `JSDOM` instances for each page you "navigate" to during a crawl, or using `Object.defineProperty()` to change what various layout-related getters and methods return. Note that other tools in the same space, such as PhantomJS, do support these features. On the wiki, we have a more complete writeup about [jsdom vs. PhantomJS](https://github.com/tmpvar/jsdom/wiki/jsdom-vs.-PhantomJS). ## Getting help If you need help with jsdom, please feel free to use any of the following venues: - The [mailing list](http://groups.google.com/group/jsdom) (best for "how do I" questions) - The [issue tracker](https://github.com/tmpvar/jsdom/issues) (best for bug reports) - The IRC channel: [#jsdom on freenode](irc://irc.freenode.net/jsdom)