



find packages

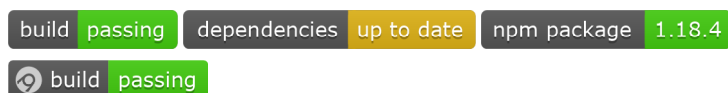
sign up or log in



We need your input. Help make JavaScript better:
[Take the 2017 JavaScript Ecosystem survey »](#)

testem public

Got Scripts? Test'em!



Unit testing in Javascript can be tedious and painful, but Testem makes it so easy that you will actually want to write tests.

Features

- Test-framework agnostic. Support for
 - **Jasmine**
 - **QUnit**
 - **Mocha**
 - **Buster.js**
 - Others, through custom test framework adapters.
- Run tests in all major browsers as well as **Node** and **PhantomJS**
- Two distinct use-cases:
 - Test-Driven-Development(TDD) — designed to streamline the TDD workflow
 - Continuous Integration(CI) — designed to work well with popular CI servers like Jenkins or Teamcity
- Cross-platform support
 - OS X
 - Windows
 - Linux
- Preprocessor support

npm install testem

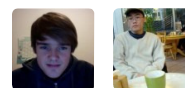
[how? learn more](#)

johanneswuerbach pub...

1.18.4 is the latest of 339 re...

github.com/testem/testem

MIT

Collaborators [list](#)

Stats

3.397 downloads in the last...

41.115 downloads in the la...

298.410 downloads in the l...

83 open issues on GitHub

15 open pull requests on Gi...

Try it out

[Test testem in your bro...](#)

Keywords

javascript, testing, unittest,
browser

- CoffeeScript
- Browserify
- JSHint/JSLint
- everything else

Screenscasts

- Watch this [introductory screencast \(11:39\)](#) to see it in action! This one demonstrates the TDD workflow.
- [Launchers \(12:10\)](#) — more detail about launchers: how to specify what to auto-launch and how to configure one yourself to run tests in **Node**.
- [Continuous Integration \(CI\) Mode \(4:24\)](#) — details about how CI mode works.
- [Making JavaScript Testing Fun With Testem \(22:53\)](#) — a thorough screencast by NetTuts+'s Jeffery Way covering the basics, Jasmine, Mocha/Chai, CoffeeScript and more!

Installation

You need **Node** version 0.10+ or iojs installed on your system. Node is extremely easy to install and has a small footprint, and is really awesome otherwise too, so **just do it**.

Once you have Node installed:

```
npm install testem -g
```

This will install the `testem` executable globally on your system.

Usage

As stated before, Testem supports two use cases: test-driven-development and continuous integration. Let's go over each one.

Development Mode

The simplest way to use Testem, in the TDD spirit, is to start in an empty directory and run the command

Dependencies (26)

`backbone`, `bluebird`, `charm`, `commander`, `consolidate`, `cross-spawn`, `express`, `fireworm`, `glob`, `http-proxy`, `js-yaml`, `lodash.assignin`, `lodash.clonedeep`, `lodash.find`, `lodash.uniqby`, `mkdirp`, `mustache`, `node-notifier`, `npmlog`, `printf`, `rimraf`, `socket.io`, `spawn-args`, `styled_string`, `tap-parser`, `xmldom`

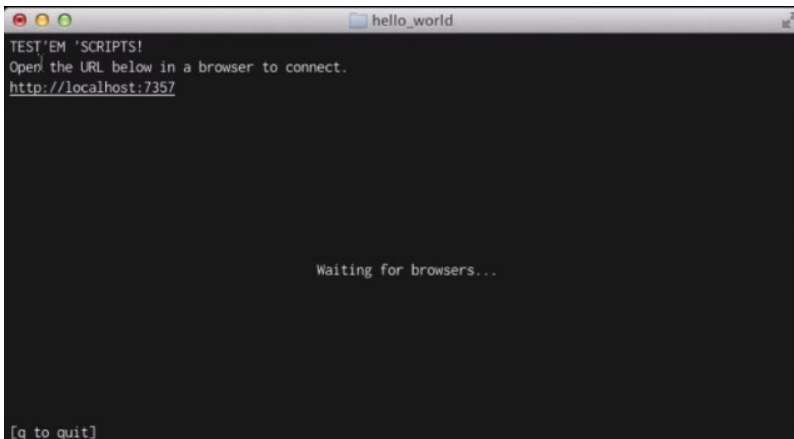
Dependents (37)

`ember-cli`, `ember-electron`, `raureif`, `lineman`, `cordova-plugin-amplify-pay`, `browserify-test`, `testem-wrap`, `grunt-testem-all`, `grunt-testem-mincer`, `driven-cli`, `ember-cli-node-webkit`, `broccoli-testem-plugin`, `ember-cli-testem`, `cabbage`, `broccoli-testem`, `bricks-cli`, `component-testem`, `anvil.testem`, `gulp-testem`, `css-devendorize`, `ember-cli-toranb`, `num_converter`, `testem-multi`, `mimosa-testem-simple`, `grunt-parts`, `rackt-cli-revolunet`, `grunt-contrib-testem`, `broccoli-testem-cli`, `ember-cli-nwjs`, `frampton-cli`, `cordova-plugin-amplify-payment`, `glimmer-build`, `@glimmer/build`, `gpII-testem`, `ember-cli-ajh`, and more

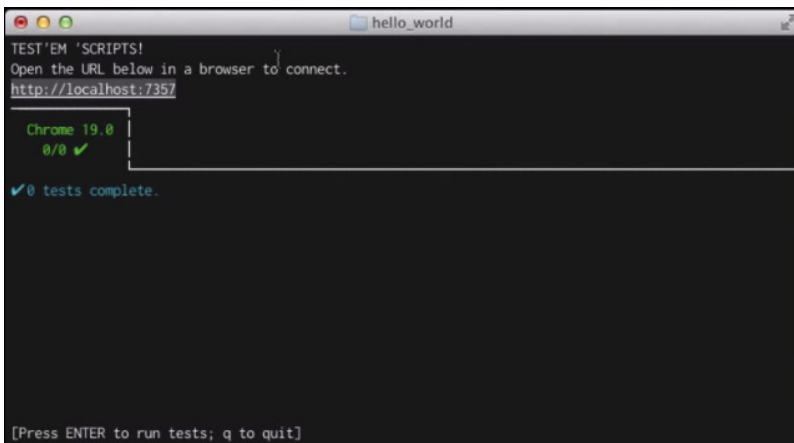
[Tinder](#) is hiring. [View more...](#)

testem

You will see a terminal-based interface which looks like this



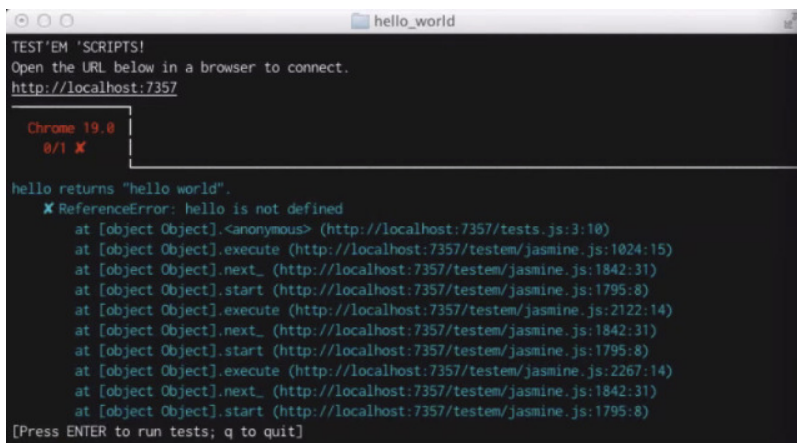
Now open your browser and go to the specified URL. You should now see



We see 0/0 for tests because at this point we haven't written any code. As we write them, Testem will pick up any .js files that were added, include them, and if there are tests, run them automatically. So let's first write `hello_spec.js` in the spirit of "test first" (written in Jasmine)

```
describe('hello', function(){
  it('should say hello', function(){
    expect(hello()).toBe('hello world');
  });
});
```

Save that file and now you should see

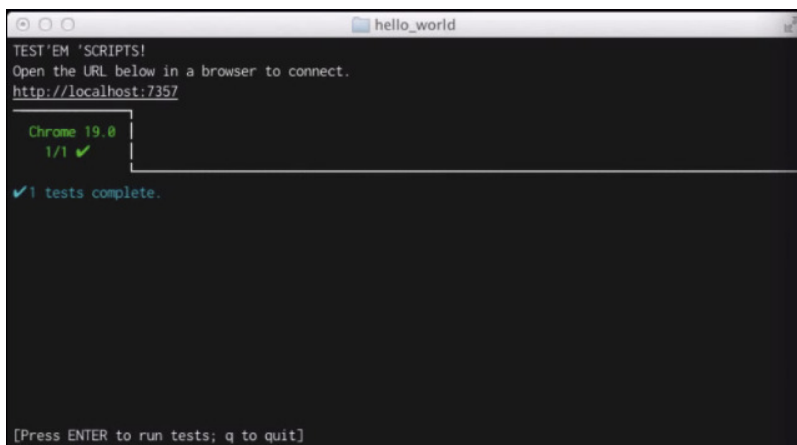


```
TEST'EM 'SCRIPTS!  
Open the URL below in a browser to connect.  
http://localhost:7357  
  
Chrome 19.0  
0/1 ✖  
  
hello returns "hello world".  
✖ ReferenceError: hello is not defined  
  at [object Object].<anonymous> (http://localhost:7357/tests.js:3:10)  
  at [object Object].execute (http://localhost:7357/testem/jasmine.js:1024:15)  
  at [object Object].next_ (http://localhost:7357/testem/jasmine.js:1842:31)  
  at [object Object].start (http://localhost:7357/testem/jasmine.js:1795:8)  
  at [object Object].execute (http://localhost:7357/testem/jasmine.js:2122:14)  
  at [object Object].next_ (http://localhost:7357/testem/jasmine.js:1842:31)  
  at [object Object].start (http://localhost:7357/testem/jasmine.js:1795:8)  
  at [object Object].execute (http://localhost:7357/testem/jasmine.js:2267:14)  
  at [object Object].next_ (http://localhost:7357/testem/jasmine.js:1842:31)  
  at [object Object].start (http://localhost:7357/testem/jasmine.js:1795:8)  
[Press ENTER to run tests; q to quit]
```

Testem should automatically pick up the new files you've added and also any changes that you make to them and rerun the tests. The test fails as we'd expect. Now we implement the spec like so in `hello.js`

```
function hello() {  
  return "hello world";  
}
```

So you should now see



```
TEST'EM 'SCRIPTS!  
Open the URL below in a browser to connect.  
http://localhost:7357  
  
Chrome 19.0  
1/1 ✔  
  
✔ 1 tests complete.  
  
[Press ENTER to run tests; q to quit]
```

Using the Text User Interface

In development mode, Testem has a text-based graphical user interface which uses keyboard-based controls. Here is a list of the control keys

- ENTER : Run the tests
- q : Quit
- ← LEFT ARROW : Move to the next browser tab on the left
- → RIGHT ARROW : Move to the next browser tab on the right
- TAB : switch the target text panel between the top and bottom halves of the split panel (if a split is present)

- ↑ UP ARROW : scroll up in the target text panel
- ↓ DOWN ARROW : scroll down in the target text panel
- SPACE : page down in the target text panel
- b : page up in the target text panel
- d : half a page down target text panel
- u : half a page up target text panel

Command line options

To see all command line options

```
testem --help
```

Continuous Integration Mode

To use Testem for continuous integration

```
testem ci
```

In CI mode, Testem runs your tests on all the browsers that are available on the system one after another.

You can run multiple browsers in parallel in CI mode by specifying the `--parallel` (or `-P`) option to be the number of concurrent running browsers.

```
testem ci -P 5 # run 5 browser in parallel
```

To find out what browsers are currently available - those that Testem knows about and can make use of

```
testem launchers
```

Will print them out. The output might look like

```
$ testem launchers
Browsers available on this system:
IE7
IE8
IE9
Chrome
Firefox
Safari
Safari Technology Preview
Opera
PhantomJS
```

Did you notice that this system has IE versions 7-9? Yes, actually it has only IE9 installed, but Testem uses IE's compatibility mode feature to emulate IE 7 and 8.

When you run `testem ci` to run tests, it outputs the results in the **TAP** format by default, which looks like

```
ok 1 Chrome 16.0 - hello should say hello.

1..1
# tests 1
# pass 1

# ok
```

TAP is a human-readable and language-agnostic test result format. TAP plugins exist for popular CI servers

- **Jenkins TAP plugin** - I've added **detailed instructions** for setup with Jenkins.
- **TeamCity TAP plugin**

TAP Options

By default, the TAP reporter outputs logs from all tests that emit logs. You can disable this behavior and only emit logs for failed tests using:

```
{  
  "tap_quiet_logs": true  
}
```

Other Test Reporters

Testem has other test reporters besides TAP: dot, xunit and teamcity. You can use the -R to specify them

```
testem ci -R dot
```

You can also **add your own reporter**.

Example xunit reporter output

Note that the real output is not pretty printed.

```
<testsuite name="Testem Tests" tests="4" fa  
  <testcase classname="PhantomJS 1.9" name=  
  <testcase classname="PhantomJS 1.9" name=  
  <testcase classname="Chrome" name="myFunc  
  <testcase classname="Chrome" name="myFunc  
    <failure name="myFunc returns false whe  
      <![CDATA[  
        Callstack...  
      ]]>  
    </failure>  
  </testcase>  
</testsuite>
```

Example teamcity reporter output

```
##teamcity[testStarted name='PhantomJS 1.9 - he
##teamcity[testFinished name='PhantomJS 1.9 - h
##teamcity[testStarted name='PhantomJS 1.9 - he
##teamcity[testFinished name='PhantomJS 1.9 - h
##teamcity[testStarted name='PhantomJS 1.9 - gc
##teamcity[testFailed name='PhantomJS 1.9 - goc
##teamcity[testFinished name='PhantomJS 1.9 - g

##teamcity[testSuiteFinished name='mocha.suite'
```

Command line options

To see all command line options for CI

```
testem ci --help
```

Configuration File

For the simplest JavaScript projects, the TDD workflow described above will work fine. There are times when you want to structure your source files into separate directories, or want to have finer control over what files to include. This calls for the `testem.json` configuration file (you can also alternatively use the YAML format with a `testem.yml` file). It looks like

```
{
  "framework": "jasmine",
  "src_files": [
    "hello.js",
    "hello_spec.js"
  ]
}
```

The `src_files` can also be unix glob patterns.


```
{
  "src_files": [
    "js/**/*.js",
    "spec/**/*.js"
  ]
}
```

You can also ignore certain files using `src_files_ignore`.

Update: I've removed the ability to use a space-separated list of globs as a string in the `src_files` property because it disallowed matching files or directories with spaces in them.

```
{
  "src_files": [
    "js/**/*.js",
    "spec/**/*.js"
  ],
  "src_files_ignore": "js/toxic/*.js"
}
```

Read [more details](#) about the config options.

Custom Test Pages

You can also use a custom page for testing. To do this, first you need to specify `test_page` to point to your test page in the config file (framework and `src_files` are irrelevant in this case)

```
{
  "test_page": "tests.html",
  "launch_in_dev": [
    "Chrome"
  ]
}
```

Next, the test page you use needs to have the adapter code installed on them, as specified in the next section.

Include Snippet

Include this snippet directly after your `jasmine.js`, `qunit.js` or `mocha.js` or `buster.js` scripts to enable Testem with your test page.

```
<script src="/testem.js"></script>
```

Or if you are using `require.js` or another loader, just make sure you load `/testem.js` as the next script after the test framework.

'`/testem.js`' here is dynamically generated to be used client-side and it should not be confused with server-side '`testem.js`'.

Dynamic Substitution

To enable dynamic substitutions within the Javascript files in your custom test page, you must

1. name your test page using `.mustache` as the extension
2. use `{{#serve_files}}` to loop over the set of Javascript files to be served, and then reference its `src` property to access their path (or `{{#css_files}}` for stylesheets)

Example:

```
{{#serve_files}}  
<script src="{{src}}"></script>  
{{/serve_files}}  
  
{{#css_files}}  
<link rel="stylesheet" href="{{src}}">  
{{/css_files}}
```

Multiple Test Pages

You can also specify multiple test pages to run by passing an array to the `test_page` option.

```
{
  "test_page": [
    "unit-tests.html",
    "integration-tests.html"
  ]
}
```

This will cause Testem to run each test page in a separate launcher instance for each launcher you are using. This means that if you define 2 test pages and are using 3 launchers you will get 6 unique runs (2 per launcher).

Launchers

Testem has the ability to automatically launch browsers or processes for you. To see the list of launchers Testem knows about, you can use the command

```
testem launchers
```

This will display something like the following

```
Have 5 launchers available; auto-launch info di
```

Launcher	Type	CI	Dev
-----	-----	--	---
Chrome	browser	✓	
Firefox	browser	✓	
Safari	browser	✓	
Opera	browser	✓	
Mocha	process(TAP)	✓	

This displays the current list of launchers that are available. Launchers can launch either a browser or a custom process — as shown in the "Type" column. Custom launchers can be defined to launch custom processes. The "CI" column indicates the launchers which will be automatically launched in CI-mode. Similarly, the "Dev" column lists those that will automatically launch in dev-mode.

Customizing Browser Arguments

Testem passes its own list of arguments to some of the browsers it launches. You can add your own custom arguments to these lists by including the `browser_args` option in your Testem configuration. For example:

```
"browser_args": {  
  "Chrome": [  
    "--auto-open-devtools-for-tabs"  
  ]  
}
```

You can supply arguments to any number of browsers Testem has available by using the launcher name as a key in `browser_args`. Values may be an array of string arguments, a single string, or an object specifying `args` and a `mode` to apply them to.

Read [more details](#) about the browser argument options.

Running Tests in Node and Custom Process Launchers

To run tests in Node you need to create a custom launcher which launches a process which will run your tests. This is nice because it means you can use any test framework - or lack thereof. For example, to make a launcher that runs mocha tests, you would write the following in the config file `testem.json`

```
"launchers": {  
  "Mocha": {  
    "command": "mocha tests/*_tests.js"  
  }  
}
```

When you run `testem`, it will auto-launch the mocha process based on the specified command every time the tests are run. It will display the stdout and well as the stderr of the process inside of the "Mocha" tab in the UI. It will base the pass/fail status on the exit code of the process. In fact, because Testem

can launch any arbitrary process for you, you could very well be using it to run programs in other languages.

Processes with TAP Output

If your process outputs test results in **TAP** format, you can tell that to testem via the `protocol` property. For example

```
"launchers": {  
  "Mocha": {  
    "command": "mocha tests/*_tests.js -R t  
    "protocol": "tap"  
  }  
}
```

When this is done, Testem will read in the process's stdout and parse it as TAP, and then display the test results in Testem's normal format. It will also hide the process's stdout output from the console log panel, although it will still display the stderr.

PhantomJS

PhantomJS is a Webkit-based headless browser. It's fast and it's awesome! Testem will pick it up if you have **PhantomJS** installed in your system and the `phantomjs` executable is in your path. Run

```
testem launchers
```

And verify that it's in the list.

If you want to debug tests in PhantomJS, include the `phantomjs_debug_port` option in your testem configuration, referencing an available port number. Once testem has started PhantomJS, navigate (with a traditional browser) to **<http://localhost>** and attach to one of PhantomJS's browser tabs (probably the second one in the list). `debugger` statements will now break in the debugging console.

If you want to use any of the **PhantomJS command line options**, include the `phantomjs_args` option in your testem

configuration. For example:

```
"phantomjs_args": [  
  "--ignore-ssl-errors=true"  
]
```

You can also customize the phantomjs launcher file by specifying the `phantomjs_launch_script` option. In this launcher you can change options like the `viewportSize`. See `assets/phantom.js` for the default launcher.

Preprocessors (CoffeeScript, LESS, Sass, Browserify, etc)

If you need to run a preprocessor (or indeed any shell command before the start of the tests) use the `before_tests` option, such as

```
"before_tests": "coffee -c *.coffee"
```

And Testem will run it before each test run. For file watching, you may still use the `src_files` option

```
"src_files": [  
  "*.coffee"  
]
```

Since you want to be serving the `.js` files that are generated and not the `.coffee` files, you want to specify the `serve_files` option to tell it that

```
"serve_files": [  
  "*.js"  
]
```

Testem will throw up a big ol' error dialog if the preprocessor command exits with an error code, so code checkers like `jshint` can be used here as well.

If you need to run a command after your tests have completed (such as removing compiled `.js` files), use the `after_tests`

option.

```
"after_tests": "rm *.js"
```

If you would prefer simply to clean up when Testem exits, you can use the `on_exit` option.

Running browser code after tests complete

It is possible to send coverage reports or run other JavaScript in the browser by using the `afterTests` method.

```
Testem.afterTests(  
  function(config, data, callback) {  
    var coverage = window.__coverage__;  
    var postBody = JSON.stringify(coverage)  
    if (postBody) {  
      var xhr = new XMLHttpRequest();  
      xhr.onreadystatechange = function()  
        if (xhr.readyState === 4) {  
          callback();  
        }  
    };  
    xhr.open('POST', 'http://localhost:  
    xhr.send(postBody);  
  }  
);
```

Custom Routes

Sometimes you may want to re-map a URL to a different directory on the file system. Maybe you have the following file structure:

```
+ src
  + hello.js
  + tests.js
+ css
  + styles.css
+ public
  + tests.html
```

Let's say you want to serve `tests.html` at the top level url `/tests.html`, all the Javascripts under `/js` and all the css under `/css`. You can use the "routes" option to do that

```
"routes": {
  "/tests.html": "public/tests.html",
  "/js": "src",
  "/css": "css"
}
```

DIY: Use Any Test Framework

If you want to use Testem with a test framework that's not supported out of the box, you can write your own custom test framework adapter. See [customAdapter.js](#) for an example of how to write a custom adapter.

Then, to use it, in your config file simply set

```
"framework": "custom"
```

And then make sure you include the adapter code in your test suite and you are ready to go. See here for the [full example](#).

Native notifications

If you'd prefer not to be looking at the terminal while developing, you can enable native notifications (e.g. notification center, growl) using the `-g` option.

API Proxy

The proxy option allows you to transparently forward HTTP requests to an external endpoint.

Simply add a `proxies` section to the `testem.json` configuration file.

```
{
  "proxies": {
    "/api": {
      "target": "http://localhost:4200",
      "onlyContentTypes": ["xml", "json"]
    },
    "/xmlapi": {
      "target": "https://localhost:8000",
      "secure": false
    }
  }
}
```

This functionality is implemented as a transparent proxy, hence a request to `http://localhost:7357/api/posts.json` will be proxied to `http://localhost:4200/api/posts.json` without removing the `/api` prefix. Setting the `secure` option to `false` as in the above `/xmlapi` configuration block will ignore TLS certificate validation and allow tests to successfully reach that URL even if testem was launched over HTTP. Other available options can be found here: <https://github.com/nodejitsu/node-http-proxy#options>

To limit the functionality to only certain content types, use `"onlyContentTypes"`.

Example Projects

I've created **examples** for various setups

- **Simple QUnit project**
- **Simple Jasmine project**
- **Jasmine 2**
- **Custom Jasmine project**
- **Custom Jasmine project using Require.js**
- **Simple Mocha Project**
- **Mocha + Chai**

- **Hybrid Project** - Mocha tests running in both the browser and Node.
- **Buster.js Project**
- **Coffeescript Project**
- **Browserify Project**
- **JSHint Example**
- **Custom Test Framework**
- **Tape Example**
- **BrowserStack Integration** **bleeding edge**
- **SauceLabs Integration** **bleeding edge**
- **Code Coverage with Istanbul** **bleeding edge**

Known Issues

1. On Windows, Mocha fails to run under Testem due to an **issue** in Node core. Until that gets resolved, I've made a **workaround** for Mocha. To install this fork of Mocha, do

```
npm install https://github.com/airportyh/moc
```

2. If you are using prototype.js version 1.6.3 or below, you will **encounter issues**.

Contributing

If you want to **contribute to the project**, I am going to do my best to stay out of your way.

Roadmap

1. **BrowserStack** integration - following **Bunyip**'s example
2. Figure out a happy path for testing on mobile browsers (maybe BrowserStack).

Core Maintainer(s)

- **Johannes Würbach**

Community

- **Mailing list:** <https://groups.google.com/forum/?fromgroups#!forum/testem-users>

Credits

Testem depends on the following great software

- [Jasmine](#)
- [QUnit](#)
- [Mocha](#)
- [Node](#)
- [Socket.IO](#)
- [PhantomJS](#)
- [Node-Tap](#)
- [Node-Charm](#)
- [Node Commander](#)
- [JS-Yaml](#)
- [Express](#)
- [jQuery](#)
- [Backbone](#)

You Need Help About npm Legal Stuff

Documentation	About npm, Inc	Terms of Use
Support / Contact Us	Jobs	Code of Conduct
Registry Status	npm Weekly	Package Name Disputes
Website Issues	Blog	Privacy Policy
CLI Issues	Twitter	Reporting Abuse
Security	GitHub	Other policies

npm loves you