This repository | Search Pull requests Issues Marketplace Gist

segmentio / **nightmare**

Watch 299 Star 12,526 Fork 708

A high-level browser automation library. https://open.segment.com

electron headless-chrome browser phantomjs chrome

934 commits **2** branches **85** releases **91** contributors

Branch: master New pull request Create new file Upload files Find file Clone or download

reinpk committed on **GitHub** adding link to niffy Latest commit `2f5c4b9` on Apr 9

lib	added ability to override internal Promise library	5 months ago
test	changed some `var` declarations to `const`	4 months ago
.gitignore	Add 'Referer' header support	a year ago
History.md	Release 2.10.0	4 months ago
Makefile	no more harmony, adding 4/5/6 to circle spec	a year ago
Readme.md	adding link to niffy	3 months ago
circle.yml	circle: auto-publish to npm on semver-ish tags	4 months ago
example.js	Adjusted example script and readme file	a year ago
package.json	Release 2.10.0	4 months ago

Readme.md

Nightmare

Nightmare is a high-level browser automation library from Segment.

The goal is to expose a few simple methods that mimic user actions (like `goto`, `type` and `click`), with an API that feels synchronous for each block of scripting, rather than deeply nested callbacks. It was originally designed for automating tasks across sites that don't have APIs, but is most often used for UI testing and crawling.

Under the covers it uses Electron, which is similar to PhantomJS but roughly 2 times faster and more modern.

Niffy is a perceptual diffing tool built on Nightmare. It helps you detect UI changes and bugs across releases of your web app.

Daydream is a complementary chrome extension built by @stevenmiller888 that generates Nightmare scripts for you while you browse.

Many thanks to @matthewmueller and @rosshinkley for their help on Nightmare.

- Examples
  - UI Testing Quick Start
  - Perceptual Diffing with Niffy & Nightmare
- API
  - Set up an instance
  - Interact with the page
  - Extract from the page
  - Cookies
  - Proxies

Examples

Let's search on DuckDuckGo:

```
var Nightmare = require('nightmare');
var nightmare = Nightmare({ show: true });

nightmare
  .goto('https://duckduckgo.com ')
  .type('#search_form_input_homepage ', 'github nightmare ')
  .click('#search_button_homepage ')
  .wait('#zero_click_wrapper .c-info__title a ')
  .evaluate(function () {
    return document.querySelector('#zero_click_wrapper .c-info__title a ').href;
  })
  .end()
  .then(function (result) {
    console.log(result);
  })
  .catch(function (error) {
    console.error('Search failed: ', error);
  });
```

You can run this with:

```
npm install nightmare
node example.js
```

Or, let's run some mocha tests:

```
var Nightmare = require('nightmare');
var expect = require('chai').expect; // jshint ignore:line

describe('test duckduckgo search results ', function() {
  it('should find the nightmare github link first ', function(done) {
    var nightmare = Nightmare()
    nightmare
      .goto('https://duckduckgo.com ')
      .type('#search_form_input_homepage ', 'github nightmare ')
      .click('#search_button_homepage ')
      .wait('#zero_click_wrapper .c-info__title a ')
      .evaluate(function () {
        return document.querySelector('#zero_click_wrapper .c-info__title a ').href
      })
      .end()
      .then(function(link) {
        expect(link).to.equal('https://github.com/segmentio/nightmare ');
        done();
      })
  });
});
```

You can see examples of every function in the tests here.

Please note that the examples are using the mocha-generators package for Mocha, which enables the support for generators.

To get started with UI Testing, check out this quick start guide.

To install dependencies

```
npm install
```

To run the mocha tests

```
npm test
```

Node versions

Nightmare is intended to be run on NodeJS 4.x or higher.

API

Nightmare(options)

Create a new instance that can navigate around the web. The available options are documented here, along with the following nightmare-specific options.

Nightmare.version

Returns the version of Nightmare.

waitTimeout (default: 30s)

This will throw an exception if the `.wait()` didn't return `true` within the set timeframe.

```
var nightmare = Nightmare({
  waitTimeout: 1000 // in ms
});
```

gotoTimeout (default: 30s)

This will throw an exception if the `.goto()` didn't finish loading within the set timeframe. Note that, even though `goto` normally waits for all the resources on a page to load, a timeout exception is only raised if the DOM itself has not yet loaded.

```
var nightmare = Nightmare({
  gotoTimeout: 1000 // in ms
});
```

loadTimeout (default: infinite)

This will force Nightmare to move on if a page transition caused by an action (eg, `.click()`) didn't finish within the set timeframe. If `loadTimeout` is shorter than `gotoTimeout`, the exceptions thrown by `gotoTimeout` will be suppressed.

```
var nightmare = Nightmare({
  loadTimeout: 1000 // in ms
});
```

executionTimeout (default: 30s)

The maxiumum amount of time to wait for an `.evaluate()` statement to complete.

```
var nightmare = Nightmare({
  executionTimeout: 1000 // in ms
});
```

paths

The default system paths that Electron knows about. Here's a list of available paths:
https://github.com/atom/electron/blob/master/docs/api/app.md#appgetpathname

You can overwrite them in Nightmare by doing the following:

```
var nightmare = Nightmare({
  paths: {
```

```
    userData: '/user/data'
  }
});
```

switches

The command line switches used by the Chrome browser that are also supported by Electron. Here's a list of supported Chrome command line switches: https://github.com/atom/electron/blob/master/docs/api/chrome-command-line-switches.md

```
var nightmare = Nightmare({
  switches: {
    'proxy-server': '1.2.3.4:5678',
    'ignore-certificate-errors': true
  }
});
```

electronPath

The path to prebuilt Electron binary. This is useful for testing on different version Electron. Note that Nightmare only supports the version this package depending on. Please use this option at your own risk.

```
var nightmare = Nightmare({
  electronPath: require('electron')
});
```

dock (OS X)

A boolean to optionally show the Electron icon in the dock (defaults to `false`). This is useful for testing purposes.

```
var nightmare = Nightmare({
  dock: true
});
```

openDevTools

Optionally show the DevTools in the Electron window using `true`, or use an object hash containing `mode: 'detach'` to show in a separate window. The hash gets passed to `contents.openDevTools()` to be handled. This is also useful for testing purposes. Note that this option is honored only if `show` is set to `true`.

```
var nightmare = Nightmare({
  openDevTools: {
    mode: 'detach'
  },
  show: true
});
```

typeInterval (default: 100ms)

How long to wait between keystrokes when using `.type()`.

```
var nightmare = Nightmare({
  typeInterval: 20
});
```

pollInterval (default: 250ms)

How long to wait between checks for the `.wait()` condition to be successful.

```
var nightmare = Nightmare({
  pollInterval: 50 //in ms
});
```

maxAuthRetries (default: 3)

Defines the number of times to retry an authentication when set up with `.authenticate()`.

```
var nightmare = Nightmare({
  maxAuthRetries: 3
});
```

.engineVersions()

Gets the versions for Electron and Chromium.

.useragent(useragent)

Set the `useragent` used by electron.

.authentication(user, password)

Set the `user` and `password` for accessing a web page using basic authentication. Be sure to set it before calling `.goto(url)`.

.end()

Complete any queue operations, disconnect and close the electron process. Note that if you're using promises, `.then()` must be called after `.end()` to run the `.end()` task. Also note that if using a `.end()` callback, the `.end()` call is equivalent to calling `.end()` followed by `.then(fn)`. Consider:

```
nightmare
  .goto(someUrl)
  .end(() => "some value")
  //prints "some value"
  .then((value) => console.log(value));
```

.halt(error, done)

Clears all queued operations, kills the electron process, and passes error message or 'Nightmare Halted' to an unresolved promise. Done will be called after the process has exited.

Interact with the Page

.goto(url[, headers])

Load the page at `url`. Optionally, a `headers` hash can be supplied to set headers on the `goto` request.

When a page load is successful, `goto` returns an object with metadata about the page load, including:

- `url`: The URL that was loaded
- `code`: The HTTP status code (e.g. 200, 404, 500)
- `method`: The HTTP method used (e.g. "GET", "POST")
- `referrer`: The page that the window was displaying prior to this load or an empty string if this is the first page load.
- `headers`: An object representing the response headers for the request as in `{header1-name: header1-value, header2-name: header2-value}`

If the page load fails, the error will be an object with the following properties:

- `message`: A string describing the type of error
- `code`: The underlying error code describing what went wrong. Note this is NOT the HTTP status code. For possible values, see https://code.google.com/p/chromium/codesearch#chromium/src/net/base/net_error_list.h
- `details`: A string with additional details about the error. This may be null or an empty string.
- `url`: The URL that failed to load

Note that any valid response from a server is considered “successful.” That means things like 404 “not found” errors are successful results for `goto`. Only things that would cause no page to appear in the browser window, such as no server responding at the given address, the server hanging up in the middle of a response, or invalid URLs, are errors.

You can also adjust how long `goto` will wait before timing out by setting the `gotoTimeout` option on the Nightmare constructor.

.back()

Go back to the previous page.

.forward()

Go forward to the next page.

.refresh()

Refresh the current page.

.click(selector)

Clicks the `selector` element once.

.mousedown(selector)

Mousedown the `selector` element once.

.mouseup(selector)

Mouseup the `selector` element once.

.mouseover(selector)

Mouseover the `selector` element once.

.type(selector[, text])

Enters the `text` provided into the `selector` element. Empty or falsey values provided for `text` will clear the selector's value.

`.type()` mimics a user typing in a textbox and will emit the proper keyboard events

Key presses can also be fired using Unicode values with `.type()`. For example, if you wanted to fire an enter key press, you would write `.type('body', '\u000d')`.

> If you don't need the keyboard events, consider using `.insert()` instead as it will be faster and more robust.

.insert(selector[, text])

Similar to `.type()`. `.insert()` enters the `text` provided into the `selector` element. Empty or falsey values provided for `text` will clear the selector's value.

`.insert()` is faster than `.type()` but does not trigger the keyboard events.

.check(selector)

checks the `selector` checkbox element.

.uncheck(selector)

unchecks the `selector` checkbox element.

.select(selector, option)

Changes the `selector` dropdown element to the option with attribute [value= `option`]

.scrollTo(top, left)

Scrolls the page to desired position. `top` and `left` are always relative to the top left corner of the document.

.viewport(width, height)

Set the viewport size.

.inject(type, file)

Inject a local `file` onto the current page. The file `type` must be either `js` or `css`.

.evaluate(fn[, arg1, arg2,...])

Invokes `fn` on the page with `arg1, arg2,...` . All the `args` are optional. On completion it returns the return value of `fn` . Useful for extracting information from the page. Here's an example:

```
var selector = 'h1';
nightmare
  .evaluate(function (selector) {
    // now we're executing inside the browser scope.
    return document.querySelector(selector).innerText;
  }, selector) // <-- that's how you pass parameters from Node scope to browser scope
  .then(function(text) {
    // ...
  })
```

Error-first callbacks are supported as a part of evaluate. If the arguments passed are one fewer than the arguments expected for the evaluated function, the evaluation will be passed a callback as the last parameter to the function. For example:

```
var selector = 'h1';
nightmare
  .evaluate(function (selector, done) {
    // now we're executing inside the browser scope.
    setTimeout(() => done(null, document.querySelector(selector).innerText), 2000);
  }, selector)
  .then(function(text) {
    // ...
  })
```

Note that callbacks support only one value argument (eg `function(err, value)`). Ultimately, the callback will get wrapped in a native Promise and only be able to resolve a single value.

Promises are also supported as a part of evaluate. If the return value of the function has a `then` member, `.evaluate()` assumes it is waiting for a promise. For example:

```
var selector = 'h1';
nightmare
  .evaluate(function (selector) {
      return new Promise((resolve, reject) => {
        setTimeout(() => resolve(document.querySelector(selector).innerText), 2000);
  })}, selector)
  .then(function(text) {
    // ...
  })
```

.wait(ms)

Wait for `ms` milliseconds e.g. `.wait(5000)`

.wait(selector)

Wait until the element `selector` is present e.g. `.wait('#pay-button')`

.wait(fn[, arg1, arg2,...])

Wait until the `fn` evaluated on the page with `arg1, arg2,...` returns `true` . All the `args` are optional. See `.evaluate()` for usage.

.header([header, value])

Add a header override for all HTTP requests. If `header` is undefined, the header overrides will be reset.

Extract from the Page

.exists(selector)

Returns whether the selector exists or not on the page.

.visible(selector)

Returns whether the selector is visible or not

.on(event, callback)

Capture page events with the callback. You have to call `.on()` before calling `.goto()`. Supported events are documented here.

Additional "page" events

.on('page', function(type="error", message, stack))

This event is triggered if any javascript exception is thrown on the page. But this event is not triggered if the injected javascript code (e.g. via `.evaluate()`) is throwing an exception.

"page" events

Listen for `window.addEventListener('error')`, `alert(...)`, `prompt(...)` & `confirm(...)`.

.on('page', function(type="error", message, stack))

Listen for top-level page errors. This will get triggered when an error is thrown on the page.

.on('page', function(type="alert", message))

Nightmare disables `window.alert` from popping up by default, but you can still listen for the contents of the alert dialog.

.on('page', function(type="prompt", message, response))

Nightmare disables `window.prompt` from popping up by default, but you can still listen for the message to come up. If you need to handle the confirmation differently, you'll need to use your own preload script.

.on('page', function(type="confirm", message, response))

Nightmare disables `window.confirm` from popping up by default, but you can still listen for the message to come up. If you need to handle the confirmation differently, you'll need to use your own preload script.

.on('console', function(type [, arguments, ...]))

`type` will be either `log`, `warn` or `error` and `arguments` are what gets passed from the console. This event is not triggered if the injected javascript code (e.g. via `.evaluate()`) is using `console.log`.

.once(event, callback)

Similar to `.on()`, but captures page events with the callback one time.

.removeListener(event, callback)

Removes a given listener callback for an event.

.screenshot([path][, clip])

Takes a screenshot of the current page. Useful for debugging. The output is always a `png`. Both arguments are optional. If `path` is provided, it saves the image to the disk. Otherwise it returns a `Buffer` of the image data. If `clip` is provided (as documented here), the image will be clipped to the rectangle.

.html(path, saveType)

Save the current page as html as files to disk at the given path. Save type options are here.

.pdf(path, options)

Saves a PDF to the specified `path`. Options are here.

.title()

Returns the title of the current page.

.url()

Returns the url of the current page.

.path()

Returns the path name of the current page.

Cookies

.cookies.get(name)

Get a cookie by it's `name`. The url will be the current url.

.cookies.get(query)

Query multiple cookies with the `query` object. If a `query.name` is set, it will return the first cookie it finds with that name, otherwise it will query for an array of cookies. If no `query.url` is set, it will use the current url. Here's an example:

```
// get all google cookies that are secure
// and have the path `/query`
nightmare
  .goto('http://google.com')
  .cookies.get({
    path: '/query',
    secure: true
  })
  .then(function(cookies) {
    // do something with the cookies
  })
```

Available properties are documented here:
https://github.com/atom/electron/blob/master/docs/api/session.md#sescookiesgetdetails-callback

.cookies.get()

Get all the cookies for the current url. If you'd like get all cookies for all urls, use: `.get({ url: null })`.

.cookies.set(name, value)

Set a cookie's `name` and `value`. Most basic form, the url will be the current url.

.cookies.set(cookie)

Set a `cookie`. If `cookie.url` is not set, it will set the cookie on the current url. Here's an example:

```
nightmare
  .goto('http://google.com')
  .cookies.set({
    name: 'token',
    value: 'some token',
    path: '/query',
    secure: true
  })
  // ... other actions ...
  .then(function() {
    // ...
  })
```

Available properties are documented here:
https://github.com/atom/electron/blob/master/docs/api/session.md#sescookiessetdetails-callback

.cookies.set(cookies)

Set multiple cookies at once. `cookies` is an array of `cookie` objects. Take a look at the `.cookies.set(cookie)` documentation above for a better idea of what `cookie` should look like.

.cookies.clear([name])

Clear a cookie for the current domain. If `name` is not specified, all cookies for the current domain will be cleared.

```
nightmare
  .goto('http://google.com')
  .cookies.clear('SomeCookieName')
  // ... other actions ...
```

```
  .then(function() {
    // ...
  })
```

.cookies.clearAll()

Clears all cookies for all domains.

```
nightmare
  .goto('http://google.com ')
  .cookies.clearAll()
  // ... other actions ...
  .then(function() {
    //...
  });
```

Proxies

Proxies are supported in Nightmare through switches.

If your proxy requires authentication you also need the authentication call.

The following example not only demonstrates how to use proxies, but you can run it to test if your proxy connection is working:

```
var Nightmare = require('nightmare');

var proxyNightmare = Nightmare({
  switches: {
    'proxy-server': 'my_proxy_server.example.com:8080 ' // set the proxy server here ...
  },
  show: true
});

proxyNightmare
  .authentication('proxyUsername', 'proxyPassword') // ... and authenticate here before `goto`
  .goto('http://www.ipchicken.com ')
  .evaluate(function() {
    return document.querySelector('b').innerText.replace(/[^\d\.]/g, '');
  })
  .end()
  .then(function(ip) { // This will log the Proxy's IP
    console.log('proxy IP:', ip);
  });

// The rest is just normal Nightmare to get your local IP
var regularNightmare = Nightmare({ show: true });

regularNightmare
  .goto('http://www.ipchicken.com ')
  .evaluate(function() {
    return document.querySelector('b').innerText.replace(/[^\d\.]/g, '');
  })
  .end()
  .then(function(ip) { // This will log the your local IP
    console.log('local IP:', ip);
  });
```

Promises

By default, Nightmare uses default native ES6 promises. You can plug in your favorite ES6-style promises library like bluebird or q for convenience!

Here's an example:

```
var Nightmare = require('nightmare');

Nightmare.Promise = require('bluebird');
// OR:
Nightmare.Promise = require('q').Promise;
```

You can also specify a custom Promise library per-instance with the `Promise` constructor option like so:

```
var Nightmare = require('nightmare');

var es6Nightmare = Nightmare();
var bluebirdNightmare = Nightmare({
    Promise: require('bluebird')
});

var es6Promise = es6Nightmare.goto('https://github.com/segmentio/nightmare ').then();
var bluebirdPromise = bluebirdNightmare.goto('https://github.com/segmentio/nightmare ').then();

es6Promise.isFulfilled() // throws: `TypeError: es6EndPromise.isFulfilled is not a function`
bluebirdPromise.isFulfilled() // returns: `true | false`
```

If you're using ES6 with Enhanced Object Literals then specifying your custom Promise library is even easier:

```
var Nightmare = require('nightmare');
var Promise = require('bluebird');

var nightmare = Nightmare({ Promise });
```

Extending Nightmare

Nightmare.action(name, [electronAction|electronNamespace], action|namespace)

You can add your own custom actions to the Nightmare prototype. Here's an example:

```
Nightmare.action('size', function (done) {
  this.evaluate_now(function() {
    var w = Math.max(document.documentElement.clientWidth, window.innerWidth || 0)
    var h = Math.max(document.documentElement.clientHeight, window.innerHeight || 0)
    return {
      height: h,
      width: w
    }
  }, done)
})

Nightmare()
  .goto('http://cnn.com ')
  .size()
  .then(function(size) {
    //... do something with the size information
  });
```

> Remember, this is attached to the static class `Nightmare`, not the instance.

You'll notice we used an internal function `evaluate_now`. This function is different than `nightmare.evaluate` because it runs it immediately, whereas `nightmare.evaluate` is queued.

An easy way to remember: when in doubt, use `evaluate`. If you're creating custom actions, use `evaluate_now`. The technical reason is that since our action has already been queued and we're running it now, we shouldn't re-queue the evaluate function.

We can also create custom namespaces. We do this internally for `nightmare.cookies.get` and `nightmare.cookies.set`. These are useful if you have a bundle of actions you want to expose, but it will clutter up the main nightmare object. Here's an example of that:

```
Nightmare.action('style', {
  background: function (done) {
    this.evaluate_now(function () {
      return window.getComputedStyle(document.body, null).backgroundColor
    }, done)
  }
})

Nightmare()
  .goto('http://google.com ')
```

```
  .style.background()
  .then(function(background) {
    // ... do something interesting with background
  })
```

You can also add custom Electron actions. The additional Electron action or namespace actions take `name`, `options`, `parent`, `win`, `renderer`, and `done`. Note the Electron action comes first, mirroring how `.evaluate()` works. For example:

```
Nightmare.action('clearCache',
  function(name, options, parent, win, renderer, done) {
    parent.respondTo('clearCache', function(done) {
      win.webContents.session.clearCache(done);
    });
    done();
  },
  function(done) {
    this.child.call('clearCache', done);
  });

Nightmare()
  .clearCache()
  .goto('http://example.org')
  //... more actions ...
  .then(function() {
    // ...
  });
```

...would clear the browser's cache before navigating to `example.org`.

.use(plugin)

`nightmare.use` is useful for reusing a set of tasks on an instance. Check out nightmare-swiftly for some examples.

Custom preload script

If you need to do something custom when you first load the window environment, you can specify a custom preload script. Here's how you do that:

```
const path = require('path');

var nightmare = Nightmare({
  webPreferences: {
    preload: path.resolve("custom-script.js")
    //alternative: preload: "absolute/path/to/custom-script.js"
  }
})
```

The only requirement for that script is that you'll need the following prelude:

```
window.__nightmare = {};
__nightmare.ipc = require('electron').ipcRenderer;
```

To benefit of all of nightmare's feedback from the browser, you can instead copy the contents of nightmare's preload script.

Storage Persistence between nightmare instances

By default nightmare will create an in-memory partition for each instance. This means that any localStorage or cookies or any other form of persistent state will be destroyed when nightmare is ended. If you would like to persist state between instances you can use the webPreferences.partition api in electron.

```
var Nightmare = require('nightmare');

nightmare = Nightmare(); // non persistent paritition by default
yield nightmare
  .evaluate(function () {
    window.localStorage.setItem('testing', 'This will not be persisted');
  })
  .end();
```

```
nightmare = Nightmare({
  webPreferences: {
    partition: 'persist: testing'
  }
});
yield nightmare
  .evaluate(function () {
    window.localStorage.setItem('testing', 'This is persisted for other instances with the same parititior
  })
  .end();
```

If you specify a `null` paritition then it will use the electron default behavior (persistent) or any string that starts with `'persist:'` will persist under that partition name, any other string will result in in-memory only storage.

Usage

Installation

Nightmare is a Node.js module, so you'll need to have Node.js installed. Then you just need to `npm install` the module:

```
$ npm install --save nightmare
```

Execution

Nightmare is a node module that can be used in a Node.js script or module. Here's a simple script to open a web page:

```
var Nightmare = require('nightmare'),
  nightmare = Nightmare();

nightmare.goto('http://cnn.com')
  .evaluate(function(){
    return document.title;
  })
  .end()
  .then(function(title){
    console.log(title);
  })
```

If you save this as `cnn.js`, you can run it on the command line like this:

```
npm install nightmare
node cnn.js
```

Common Execution Problems

Nightmare heavily relies on Electron for heavy lifting. And Electron in turn relies on several UI-focused dependencies (eg. libgtk+) which are often missing from server distros.

For help running nightmare on your server distro check out How to run nightmare on Amazon Linux and CentOS guide.

Debugging

There are three good ways to get more information about what's happening inside the headless browser:

1. Use the `DEBUG=*` flag described below.
2. Pass `{ show: true }` to the nightmare constructor to have it create a visible, rendered window that you can watch what's happening.
3. Listen for specific events.

To run the same file with debugging output, run it like this `DEBUG=nightmare node cnn.js` (on Windows use `set DEBUG=nightmare & node cnn.js`).

This will print out some additional information about what's going on:

```
nightmare queueing action  "goto" +0ms
nightmare queueing action  "evaluate" +4ms
Breaking News, U.S., World, Weather, Entertainment  & Video News - CNN.com
```

Debug Flags

All nightmare messages

```
DEBUG=nightmare*
```

Only actions

```
DEBUG=nightmare:actions*
```

Only logs

```
DEBUG=nightmare:log*
```

Tests

Automated tests for nightmare itself are run using Mocha and Chai, both of which will be installed via `npm install`. To run nightmare's tests, just run `make test`.

When the tests are done, you'll see something like this:

```
make test
  .................
  18 passing (1m)
```

Note that if you are using `xvfb`, `make test` will automatically run the tests under an `xvfb-run` wrapper. If you are planning to run the tests headlessly without running `xvfb` first, set the `HEADLESS` environment variable to `0`.

License (MIT)

```
WWWWWW||WWWWWW
 W W W||W W W
      ||
    ( OO )__________
     /  |           \
    /o o|    MIT     \
    \___/||_||__||_|| *
         || ||  || ||
        _||_|| _||_||
       (__|__|(__|__|
```

Copyright (c) 2015 Segment.io, Inc. friends@segment.com

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

© 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About