SuperAgent

SuperAgent is light-weight progressive ajax API crafted for flexibility, readability, and a low learning curve after being frustrated with many of the existing request APIs. It also works with Node.js!

```
request
  .post('/api/pet')
  .send({ name: 'Manny', species: 'cat' })
  .set('X-API-Key', 'foobar')
  .set('Accept', 'application/json')
  .then(res => {
     alert('yay got ' + JSON.stringify(res.body));
  });
```

Test documentation

The following test documentation was generated with Mocha's "doc" reporter, and directly reflects the test suite. This provides an additional source of documentation.

Request basics

A request can be initiated by invoking the appropriate method on the `request` object, then calling `.then()` (or `.end()` or `await`) to send the request. For example a simple **GET** request:

```
request
  .get('/search')
  .then(res => {
     // res.body, res.headers, res.status
  })
  .catch(err => {
     // err.message, err.response
  });
```

HTTP method may also be passed as a string:

```
request('GET', '/search').then(success, failure);
```

Old-style callbacks are also supported, but not recommended. *Instead of* `.then()` you can call `.end()`:

```
request('GET', '/search').end(function(err, res){
  if (res.ok) {}
});
```

Absolute URLs can be used. In web browsers absolute URLs work only if the server implements CORS.

```
request
  .get('https://example.com/search')
  .then(res => {

  });
```

The **Node** client supports making requests to Unix Domain Sockets:

```
// pattern: https?+unix://SOCKET_PATH/REQUEST_PATH
//          Use `%2F` as `/` in SOCKET_PATH
try {
  const res = await request
    .get('http+unix://%2Fabsolute%2Fpath%2Fto%2Funix.sock/search');
  // res.body, res.headers, res.status
} catch(err) {
  // err.message, err.response
}
```

DELETE, **HEAD**, **PATCH**, **POST**, and **PUT** requests can also be used, simply change the method name:

```
request
  .head('/favicon.ico')
  .then(res => {

  });
```

DELETE can be also called as `.del()` for compatibility with old IE where `delete` is a reserved word.

The HTTP method defaults to **GET**, so if you wish, the following is valid:

```
 request('/search', (err, res) => {

 });
```

Setting header fields

Setting header fields is simple, invoke `.set()` with a field name and value:

```
request
  .get('/search')
  .set('API-Key', 'foobar')
  .set('Accept', 'application/json')
  .then(callback);
```

You may also pass an object to set several fields in a single call:

```
request
  .get('/search')
  .set({ 'API-Key': 'foobar', Accept: 'application/json' })
  .then(callback);
```

`GET` requests

The `.query()` method accepts objects, which when used with the **GET** method will form a query-string. The following will produce the path `/search?query=Manny&range=1..5&order=desc`.

```
request
  .get('/search')
  .query({ query: 'Manny' })
  .query({ range: '1..5' })
  .query({ order: 'desc' })
```

```
  .then(res => {

  });
```

Or as a single object:

```
request
  .get('/search')
  .query({ query: 'Manny', range: '1..5', order: 'desc' })
  .then(res => {

  });
```

The `.query()` method accepts strings as well:

```
request
  .get('/querystring')
  .query('search=Manny&range=1..5')
  .then(res => {

  });
```

Or joined:

```
request
  .get('/querystring')
  .query('search=Manny')
  .query('range=1..5')
  .then(res => {
```

```
  });
```

HEAD requests

You can also use the `.query()` method for HEAD requests. The following will produce the path `/users?email=joe@smith.com`.

```
request
  .head('/users')
  .query({ email: 'joe@smith.com' })
  .then(res => {

  });
```

POST / PUT requests

A typical JSON **POST** request might look a little like the following, where we set the Content-Type header field appropriately, and "write" some data, in this case just a JSON string.

```
request.post('/user')
  .set('Content-Type', 'application/json')
  .send('{"name":"tj","pet":"tobi"}')
  .then(callback)
  .catch(errorCallback)
```

Since JSON is undoubtedly the most common, it's the *default*! The following example is equivalent to the previous.

```
request.post('/user')
  .send({ name: 'tj', pet: 'tobi' })
  .then(callback, errorCallback)
```

Or using multiple `.send()` calls:

```
request.post('/user')
  .send({ name: 'tj' })
  .send({ pet: 'tobi' })
  .then(callback, errorCallback)
```

By default sending strings will set the `Content-Type` to `application/x-www-form-urlencoded`, multiple calls will be concatenated with `&`, here resulting in `name=tj&pet=tobi`:

```
request.post('/user')
  .send('name=tj')
  .send('pet=tobi')
  .then(callback, errorCallback);
```

SuperAgent formats are extensible, however by default "json" and "form" are supported. To send the data as `application/x-www-form-urlencoded` simply invoke `.type()` with "form", where the default is "json". This request will **POST** the body "name=tj&pet=tobi".

```
request.post('/user')
  .type('form')
  .send({ name: 'tj' })
```

```
    .send({ pet: 'tobi' })
    .then(callback, errorCallback)
```

Sending a `FormData` object is also supported. The following example will **POST** the content of the HTML form identified by id="myForm":

```
request.post('/user')
  .send(new FormData(document.getElementById('myForm')))
  .then(callback, errorCallback)
```

Setting the `Content-Type`

The obvious solution is to use the `.set()` method:

```
request.post('/user')
  .set('Content-Type', 'application/json')
```

As a short-hand the `.type()` method is also available, accepting the canonicalized MIME type name complete with type/subtype, or simply the extension name such as "xml", "json", "png", etc:

```
request.post('/user')
  .type('application/json')

request.post('/user')
  .type('json')

request.post('/user')
  .type('png')
```

Serializing request body

SuperAgent will automatically serialize JSON and forms. You can setup automatic serialization for other types as well:

```
request.serialize['application/xml'] = function (obj) {
    return 'string generated from obj';
};

// going forward, all requests with a Content-type of
// 'application/xml' will be automatically serialized
```

If you want to send the payload in a custom format, you can replace the built-in serialization with the `.serialize()` method on a per-request basis:

```
request
    .post('/user')
    .send({foo: 'bar'})
    .serialize(obj => {
        return 'string generated from obj';
    });
```

Retrying requests

When given the `.retry()` method, SuperAgent will automatically retry requests, if they fail in a way that is transient or could be due to a flaky Internet connection.

This method has two optional arguments: number of retries (default 1) and a callback. It calls `callback(err, res)` before each retry. The callback may return `true`/`false` to control whether the request should be retried (but the maximum number of retries is always applied).

```
request
  .get('https://example.com/search')
  .retry(2) // or:
  .retry(2, callback)
  .then(finished);
  .catch(failed);
```

Use `.retry()` only with requests that are *idempotent* (i.e. multiple requests reaching the server won't cause undesirable side effects like duplicate purchases).

All request methods are tried by default (which means if you do not want POST requests to be retried, you will need to pass a custom retry callback).

By default the following status codes are retried:

- 408
- 413
- 429
- 500
- 502
- 503
- 504
- 521
- 522
- 524

By default the following error codes are retried:

- `'ETIMEDOUT'`
- `'ECONNRESET'`
- `'EADDRINUSE'`
- `'ECONNREFUSED'`
- `'EPIPE'`
- `'ENOTFOUND'`
- `'ENETUNREACH'`
- `'EAI_AGAIN'`

Setting Accept

In a similar fashion to the `.type()` method it is also possible to set the `Accept` header via the short hand method `.accept()`. Which references `request.types` as well allowing you to specify either the full canonicalized MIME type name as `type/subtype`, or the extension suffix form as "xml", "json", "png", etc. for convenience:

```
request.get('/user')
  .accept('application/json')

request.get('/user')
  .accept('json')

request.post('/user')
  .accept('png')
```

Facebook and Accept JSON

If you are calling Facebook's API, be sure to send an `Accept: application/json` header in your request. If you don't do this,

Facebook will respond with `Content-Type: text/javascript; charset=UTF-8`, which SuperAgent will not parse and thus `res.body` will be undefined. You can do this with either `req.accept('json')` or `req.header('Accept', 'application/json')`. See issue 1078 for details.

Query strings

`req.query(obj)` is a method which may be used to build up a query-string. For example populating `?format=json&dest=/login` on a **POST**:

```
request
  .post('/')
  .query({ format: 'json' })
  .query({ dest: '/login' })
  .send({ post: 'data', here: 'wahoo' })
  .then(callback);
```

By default the query string is not assembled in any particular order. An asciibetically-sorted query string can be enabled with `req.sortQuery()`. You may also provide a custom sorting comparison function with `req.sortQuery(myComparisonFn)`. The comparison function should take 2 arguments and return a negative/zero/positive integer.

```
// default order
request.get('/user')
  .query('name=Nick')
  .query('search=Manny')
  .sortQuery()
  .then(callback)
```

```
// customized sort function
request.get('/user')
  .query('name=Nick')
  .query('search=Manny')
  .sortQuery((a, b) => a.length - b.length)
  .then(callback)
```

TLS options

In Node.js SuperAgent supports methods to configure HTTPS requests:

- `.ca()`: Set the CA certificate(s) to trust
- `.cert()`: Set the client certificate chain(s)
- `.key()`: Set the client private key(s)
- `.pfx()`: Set the client PFX or PKCS12 encoded private key and certificate chain
- `.disableTLSCerts()`: Does not reject expired or invalid TLS certs. Sets internally `rejectUnauthorized=true`. *Be warned, this method allows MITM attacks.*

For more information, see Node.js https.request docs.

```
var key = fs.readFileSync('key.pem'),
    cert = fs.readFileSync('cert.pem');

request
  .post('/client-auth')
  .key(key)
  .cert(cert)
  .then(callback);
```

```
var ca = fs.readFileSync('ca.cert.pem');

request
  .post('https://localhost/private-ca-server')
  .ca(ca)
  .then(res => {});
```

Parsing response bodies

SuperAgent will parse known response-body data for you, currently supporting `application/x-www-form-urlencoded`, `application/json`, and `multipart/form-data`. You can setup automatic parsing for other response-body data as well:

```
//browser
request.parse['application/xml'] = function (str) {
    return {'object': 'parsed from str'};
};

//node
request.parse['application/xml'] = function (res, cb) {
    //parse response text and set res.body here

    cb(null, res);
};

//going forward, responses of type 'application/xml'
//will be parsed automatically
```

You can set a custom parser (that takes precedence over built-in parsers) with the `.buffer(true).parse(fn)` method. If response buffering is not enabled (`.buffer(false)`) then the `response` event will be emitted without waiting for the body parser to finish, so `response.body` won't be available.

JSON / Urlencoded

The property `res.body` is the parsed object, for example if a request responded with the JSON string '{"user":{"name":"tobi"}}', `res.body.user.name` would be "tobi". Likewise the x-www-form-urlencoded value of "user[name]=tobi" would yield the same result. Only one level of nesting is supported. If you need more complex data, send JSON instead.

Arrays are sent by repeating the key. `.send({color: ['red','blue']})` sends `color=red&color=blue`. If you want the array keys to contain `[]` in their name, you must add it yourself, as SuperAgent doesn't add it automatically.

Multipart

The Node client supports *multipart/form-data* via the Formidable module. When parsing multipart responses, the object `res.files` is also available to you. Suppose for example a request responds with the following multipart body:

```
--whoop
Content-Disposition: attachment; name="image"; filename="tobi.png"
Content-Type: image/png

... data here ...
--whoop
Content-Disposition: form-data; name="name"
```

```
Content-Type: text/plain

Tobi
--whoop--
```

You would have the values `res.body.name` provided as "Tobi", and `res.files.image` as a `File` object containing the path on disk, filename, and other properties.

Binary

In browsers, you may use `.responseType('blob')` to request handling of binary response bodies. This API is unnecessary when running in node.js. The supported argument values for this method are

- `'blob'` passed through to the XmlHTTPRequest `responseType` property
- `'arraybuffer'` passed through to the XmlHTTPRequest `responseType` property

```
req.get('/binary.data')
  .responseType('blob')
  .then(res => {
    // res.body will be a browser native Blob type here
  });
```

For more information, see the Mozilla Developer Network xhr.responseType docs.

Response properties

Many helpful flags and properties are set on the `Response` object, ranging from the response text, parsed response body, header fields, status flags and more.

Response text

The `res.text` property contains the unparsed response body string. This property is always present for the client API, and only when the mime type matches "text/*", *"/json", or "x-www-form-urlencoded" by default for node. The reasoning is to conserve memory, as buffering text of large bodies such as multipart files or images is extremely inefficient. To force buffering see the "Buffering responses" section.

Response body

Much like SuperAgent can auto-serialize request data, it can also automatically parse it. When a parser is defined for the Content-Type, it is parsed, which by default includes "application/json" and "application/x-www-form-urlencoded". The parsed object is then available via `res.body`.

Response header fields

The `res.header` contains an object of parsed header fields, lowercasing field names much like node does. For example `res.header['content-length']`.

Response Content-Type

The Content-Type response header is special-cased, providing `res.type`, which is void of the charset (if any). For example the Content-Type of "text/html; charset=utf8" will provide "text/html" as `res.type`, and the `res.charset` property would then contain "utf8".

Response status

The response status flags help determine if the request was a success, among other useful information, making SuperAgent ideal for interacting with RESTful web services. These flags are currently defined as:

```
var type = status / 100 | 0;

// status / class
res.status = status;
res.statusType = type;

// basics
res.info = 1 == type;
res.ok = 2 == type;
res.clientError = 4 == type;
res.serverError = 5 == type;
res.error = 4 == type || 5 == type;

// sugar
res.accepted = 202 == status;
res.noContent = 204 == status || 1223 == status;
res.badRequest = 400 == status;
res.unauthorized = 401 == status;
res.notAcceptable = 406 == status;
res.notFound = 404 == status;
res.forbidden = 403 == status;
```

Aborting requests

To abort requests simply invoke the `req.abort()` method.

Timeouts

Sometimes networks and servers get "stuck" and never respond after accepting a request. Set timeouts to avoid requests waiting forever.

- `req.timeout({deadline:ms})` or `req.timeout(ms)` (where `ms` is a number of milliseconds > 0) sets a deadline for the entire request (including all uploads, redirects, server processing time) to complete. If the response isn't fully downloaded within that time, the request will be aborted.

- `req.timeout({response:ms})` sets maximum time to wait for the first byte to arrive from the server, but it does not limit how long the entire download can take. Response timeout should be at least few seconds longer than just the time it takes the server to respond, because it also includes time to make DNS lookup, TCP/IP and TLS connections, and time to upload request data.

You should use both `deadline` and `response` timeouts. This way you can use a short response timeout to detect unresponsive networks quickly, and a long deadline to give time for downloads on slow, but reliable, networks. Note that both of these timers limit how long *uploads* of attached files are allowed to take. Use long timeouts if you're uploading files.

```
request
  .get('/big-file?network=slow')
  .timeout({
    response: 5000,  // Wait 5 seconds for the server to start sending,
    deadline: 60000, // but allow 1 minute for the file to finish loading.
  })
  .then(res => {
      /* responded in time */
    }, err => {
```

```
    if (err.timeout) { /* timed out! */ } else { /* other error */ }
});
```

Timeout errors have a `.timeout` property.

Authentication

In both Node and browsers auth available via the `.auth()` method:

```
request
  .get('http://local')
  .auth('tobi', 'learnboost')
  .then(callback);
```

In the *Node* client Basic auth can be in the URL as "user:pass":

```
request.get('http://tobi:learnboost@local').then(callback);
```

By default only `Basic` auth is used. In browser you can add `{type:'auto'}` to enable all methods built-in in the browser (Digest, NTLM, etc.):

```
request.auth('digest', 'secret', {type:'auto'})
```

The `auth` method also supports a `type` of `bearer`, to specify token-based authentication:

```
request.auth('my_token', { type: 'bearer' })
```

Following redirects

By default up to 5 redirects will be followed, however you may specify this with the `res.redirects(n)` method:

```
const response = await request.get('/some.png').redirects(2);
```

Redirects exceeding the limit are treated as errors. Use `.ok(res => res.status < 400)` to read them as successful responses.

Agents for global state

Saving cookies

In Node SuperAgent does not save cookies by default, but you can use the `.agent()` method to create a copy of SuperAgent that saves cookies. Each copy has a separate cookie jar.

```
const agent = request.agent();
agent
  .post('/login')
  .then(() => {
    return agent.get('/cookied-page');
  });
```

In browsers cookies are managed automatically by the browser, so the `.agent()` does not isolate cookies.

Default options for multiple requests

Regular request methods called on the agent will be used as defaults for all requests made by that agent.

```
const agent = request.agent()
  .use(plugin)
  .auth(shared);

await agent.get('/with-plugin-and-auth');
await agent.get('/also-with-plugin-and-auth');
```

The complete list of methods that the agent can use to set defaults is: `use`, `on`, `once`, `set`, `query`, `type`, `accept`, `auth`, `withCredentials`, `sortQuery`, `retry`, `ok`, `redirects`, `timeout`, `buffer`, `serialize`, `parse`, `ca`, `key`, `pfx`, `cert`.

Piping data

The Node client allows you to pipe data to and from the request. Please note that `.pipe()` is used **instead of** `.end()`/`.then()` methods.

For example piping a file's contents as the request:

```
const request = require('superagent');
const fs = require('fs');

const stream = fs.createReadStream('path/to/my.json');
const req = request.post('/somewhere');
req.type('json');
stream.pipe(req);
```

Note that when you pipe to a request, superagent sends the piped data with [chunked transfer encoding](), which isn't supported by all servers (for instance, Python WSGI servers).

Or piping the response to a file:

```
const stream = fs.createWriteStream('path/to/my.json');
const req = request.get('/some.json');
req.pipe(stream);
```

It's not possible to mix pipes and callbacks or promises. Note that you should **NOT** attempt to pipe the result of `.end()` or the `Response` object:

```
// Don't do either of these:
const stream = getAWritableStream();
const req = request
  .get('/some.json')
  // BAD: this pipes garbage to the stream and fails in unexpected ways
  .end((err, this_does_not_work) => this_does_not_work.pipe(stream))
const req = request
  .get('/some.json')
  .end()
  // BAD: this is also unsupported, .pipe calls .end for you.
  .pipe(nope_its_too_late);
```

In a [future version]() of superagent, improper calls to `pipe()` will fail.

Multipart requests

SuperAgent is also great for *building* multipart requests for which it provides methods `.attach()` and `.field()`.

When you use `.field()` or `.attach()` you can't use `.send()` and you *must not* set `Content-Type` (the correct type will be set for you).

Attaching files

To send a file use `.attach(name, [file], [options])`. You can attach multiple files by calling `.attach` multiple times. The arguments are:

- `name` — field name in the form.
- `file` — either string with file path or `Blob`/`Buffer` object.
- `options` — (optional) either string with custom file name or `{filename: string}` object. In Node also `{contentType: 'mime/type'}` is supported. In browser create a `Blob` with an appropriate type instead.

```
request
  .post('/upload')
  .attach('image1', 'path/to/felix.jpeg')
  .attach('image2', imageBuffer, 'luna.jpeg')
  .field('caption', 'My cats')
  .then(callback);
```

Field values

Much like form fields in HTML, you can set field values with `.field(name, value)` and `.field({name: value})`. Suppose you want to upload a few images with your name and email, your request might look something like this:

```
request
  .post('/upload')
  .field('user[name]', 'Tobi')
  .field('user[email]', 'tobi@learnboost.com')
  .field('friends[]', ['loki', 'jane'])
  .attach('image', 'path/to/tobi.png')
  .then(callback);
```

Compression

The node client supports compressed responses, best of all, you don't have to do anything! It just works.

Buffering responses

To force buffering of response bodies as `res.text` you may invoke `req.buffer()`. To undo the default of buffering for text responses such as "text/plain", "text/html" etc you may invoke `req.buffer(false)`.

When buffered the `res.buffered` flag is provided, you may use this to handle both buffered and unbuffered responses in the same callback.

CORS

For security reasons, browsers will block cross-origin requests unless the server opts-in using CORS headers. Browsers will also make extra **OPTIONS** requests to check what HTTP headers and methods are allowed by the server. Read more about CORS.

The `.withCredentials()` method enables the ability to send cookies from the origin, however only when `Access-Control-Allow-Origin` is

not a wildcard ("*"), and `Access-Control-Allow-Credentials` is "true".

```
request
  .get('https://api.example.com:4001/')
  .withCredentials()
  .then(res => {
    assert.equal(200, res.status);
    assert.equal('tobi', res.text);
  })
```

Error handling

Your callback function will always be passed two arguments: error and response. If no error occurred, the first argument will be null:

```
request
 .post('/upload')
 .attach('image', 'path/to/tobi.png')
 .then(res => {

 });
```

An "error" event is also emitted, with you can listen for:

```
request
  .post('/upload')
  .attach('image', 'path/to/tobi.png')
  .on('error', handle)
  .then(res => {
```

```
});
```

Note that **superagent considers 4xx and 5xx responses (as well as unhandled 3xx responses) errors by default**. For example, if you get a `304 Not modified`, `403 Forbidden` or `500 Internal server error` response, this status information will be available via `err.status`. Errors from such responses also contain an `err.response` field with all of the properties mentioned in "Response properties". The library behaves in this way to handle the common case of wanting success responses and treating HTTP error status codes as errors while still allowing for custom logic around specific error conditions.

Network failures, timeouts, and other errors that produce no response will contain no `err.status` or `err.response` fields.

If you wish to handle 404 or other HTTP error responses, you can query the `err.status` property. When an HTTP error occurs (4xx or 5xx response) the `res.error` property is an `Error` object, this allows you to perform checks such as:

```
if (err && err.status === 404) {
  alert('oh no ' + res.body.message);
}
else if (err) {
  // all other error types we handle generically
}
```

Alternatively, you can use the `.ok(callback)` method to decide whether a response is an error or not. The callback to the `ok` function

gets a response and returns `true` if the response should be interpreted as success.

```
request.get('/404')
  .ok(res => res.status < 500)
  .then(response => {
    // reads 404 page as a successful response
  })
```

Progress tracking

SuperAgent fires `progress` events on upload and download of large files.

```
request.post(url)
  .attach('field_name', file)
  .on('progress', event => {
    /* the event is:
    {
      direction: "upload" or "download"
      percent: 0 to 100 // may be missing if file size is unknown
      total: // total file size, may be missing
      loaded: // bytes downloaded or uploaded so far
    } */
  })
  .then()
```

Testing on localhost

Forcing specific connection IP address

In Node.js it's possible to ignore DNS resolution and direct all requests to a specific IP address using `.connect()` method. For example, this request will go to localhost instead of `example.com`:

```
const res = await request.get("http://example.com").connect("127.0.0.1");
```

Because the request may be redirected, it's possible to specify multiple hostnames and multiple IPs, as well as a special `*` as the fallback (note: other wildcards are not supported). The requests will keep their `Host` header with the original value. `.connect(undefined)` turns off the feature.

```
const res = await request.get("http://redir.example.com:555")
  .connect({
    "redir.example.com": "127.0.0.1", // redir.example.com:555 will use 127.0.0.1:555
    "www.example.com": false, // don't override this one; use DNS as normal
    "mapped.example.com": { host: "127.0.0.1", port: 8080}, // mapped.example.com:* will use 127.0.0.1:8080
    "*": "proxy.example.com", // all other requests will go to this host
  });
```

Ignoring broken/insecure HTTPS on localhost

In Node.js, when HTTPS is misconfigured and insecure (e.g. using self-signed certificate *without* specifying own `.ca()`), it's still possible to permit requests to `localhost` by calling `.trustLocalhost()`:

```
const res = await request.get("https://localhost").trustLocalhost()
```

Together with `.connect("127.0.0.1")` this may be used to force HTTPS requests to any domain to be re-routed to `localhost` instead.

It's generally safe to ignore broken HTTPS on `localhost`, because the loopback interface is not exposed to untrusted networks. Trusting `localhost` may become the default in the future. Use `.trustLocalhost(false)` to force check of `127.0.0.1`'s authenticity.

We intentionally don't support disabling of HTTPS security when making requests to any other IP, because such options end up abused as a quick "fix" for HTTPS problems. You can get free HTTPS certificates from Let's Encrypt or set your own CA (`.ca(ca_public_pem)`) to make your self-signed certificates trusted.

Promise and Generator support

SuperAgent's request is a "thenable" object that's compatible with JavaScript promises and the `async`/`await` syntax.

```
const res = await request.get(url);
```

If you're using promises, **do not** call `.end()` or `.pipe()`. Any use of `.then()` or `await` disables all other ways of using the request.

Libraries like co or a web framework like koa can `yield` on any SuperAgent method:

```
const req = request
  .get('http://local')
  .auth('tobi', 'learnboost');
const res = yield req;
```

Note that SuperAgent expects the global `Promise` object to be present. You'll need a polyfill to use promises in Internet Explorer or

Node.js 0.10.

Browser and node versions

SuperAgent has two implementations: one for web browsers (using XHR) and one for Node.JS (using core http module). By default Browserify and WebPack will pick the browser version.

If want to use WebPack to compile code for Node.JS, you *must* specify node target in its configuration.

Using browser version in electron

Electron developers report if you would prefer to use the browser version of SuperAgent instead of the Node version, you can `require('superagent/superagent')`. Your requests will now show up in the Chrome developer tools Network tab. Note this environment is not covered by automated test suite and not officially supported.