Need private packages and team management tools? Check out npm Orgs. »

request

2.88.0 • Public • Published a year ago

Readme

20 Dependencies

45,048 Dependents

125 Versions

install

```
> npm i request
```

weekly downloads

15,174,440

version
2.88.0

license
Apache-2.0

open issues
179

pull requests
57

homepage
github.com

repository
github

last publish
a year ago

collaborators




Test with RunKit

Report a vulnerability

Request - Simplified HTTP client

```
npm install request
20 dependencies      version 2.88.0
                     updated a year ago
1,503 ★
```

build passing | coverage 97% | coverage 97% | dependencies up to date | vulnerabilities 0 | gitter join chat

Super simple to use

Request is designed to be the simplest way possible to make http calls. It supports HTTPS and follows redirects by default.

```
var request = require('request');
request('http://www.google.com', function (error, response, body) {
  console.log('error:', error); // Print the error if one occurred
  console.log('statusCode:', response && response.statusCode); // Print the response status co
  console.log('body:', body); // Print the HTML for the Google homepage.
});
```

Table of contents

- Support for HAR 1.2
- **All Available Options**

Request also offers **convenience methods** like `request.defaults` and `request.post`, and there are lots of **usage examples** and several **debugging techniques**.

Streaming

You can stream any response to a file stream.

```
request('http://google.com/doodle.png').pipe(fs.createWriteStream('doodle.png'))
```

You can also stream a file to a PUT or POST request. This method will also check the file extension against a mapping of file extensions to content-types (in this case `application/json`) and use the proper `content-type` in the PUT request (if the headers don't already provide one).

```
fs.createReadStream('file.json').pipe(request.put('http://mysite.com/obj.json'))
```

Request can also `pipe` to itself. When doing so, `content-type` and `content-length` are preserved in the PUT headers.

```
request.get('http://google.com/img.png').pipe(request.put('http://mysite.com/img.png'))
```

Request emits a "response" event when a response is received. The `response` argument will be an instance of **http.IncomingMessage**.

```
request
  .get('http://google.com/img.png')
  .on('response', function(response) {
    console.log(response.statusCode) // 200
    console.log(response.headers['content-type']) // 'image/png'
  })
  .pipe(request.put('http://mysite.com/img.png'))
```

To easily handle errors when streaming requests, listen to the `error` event before piping:

```
request
  .get('http://mysite.com/doodle.png')
  .on('error', function(err) {
    console.log(err)
  })
  .pipe(fs.createWriteStream('doodle.png'))
```

Now let's get fancy.

```
http.createServer(function (req, resp) {
  if (req.url === '/doodle.png') {
    if (req.method === 'PUT') {
      req.pipe(request.put('http://mysite.com/doodle.png'))
    } else if (req.method === 'GET' || req.method === 'HEAD') {
      request.get('http://mysite.com/doodle.png').pipe(resp)
```

```
    }
  }
})
```

You can also `pipe()` from `http.ServerRequest` instances, as well as to `http.ServerResponse` instances. The HTTP method, headers, and entity-body data will be sent. Which means that, if you don't really care about security, you can do:

```
http.createServer(function (req, resp) {
  if (req.url === '/doodle.png') {
    var x = request('http://mysite.com/doodle.png')
    req.pipe(x)
    x.pipe(resp)
  }
})
```

And since `pipe()` returns the destination stream in ≥ Node 0.5.x you can do one line proxying. :)

```
req.pipe(request('http://mysite.com/doodle.png')).pipe(resp)
```

Also, none of this new functionality conflicts with requests previous features, it just expands them.

```
var r = request.defaults({'proxy':'http://localproxy.com'})

http.createServer(function (req, resp) {
  if (req.url === '/doodle.png') {
```

```
    r.get('http://google.com/doodle.png').pipe(resp)
  }
})
```

You can still use intermediate proxies, the requests will still follow HTTP forwards, etc.

back to top

Promises & Async/Await

`request` supports both streaming and callback interfaces natively. If you'd like `request` to return a Promise instead, you can use an alternative interface wrapper for `request`. These wrappers can be useful if you prefer to work with Promises, or if you'd like to use `async` / `await` in ES2017.

Several alternative interfaces are provided by the request team, including:

- `request-promise` (uses Bluebird Promises)
- `request-promise-native` (uses native Promises)
- `request-promise-any` (uses any-promise Promises)

back to top

Forms

`request` supports `application/x-www-form-urlencoded` and `multipart/form-data` form uploads. For `multipart/related` refer to the `multipart` API.

application/x-www-form-urlencoded (URL-Encoded Forms)

URL-encoded forms are simple.

```
request.post('http://service.com/upload', {form:{key:'value'}})
// or
request.post('http://service.com/upload').form({key:'value'})
// or
request.post({url:'http://service.com/upload', form: {key:'value'}}, function(err,httpResponse
```

multipart/form-data (Multipart Form Uploads)

For `multipart/form-data` we use the **form-data** library by **@felixge**. For the most cases, you can pass your upload form data via the `formData` option.

```
var formData = {
  // Pass a simple key-value pair
  my_field: 'my_value',
  // Pass data via Buffers
  my_buffer: Buffer.from([1, 2, 3]),
  // Pass data via Streams
  my_file: fs.createReadStream(__dirname + '/unicycle.jpg'),
  // Pass multiple values /w an Array
  attachments: [
    fs.createReadStream(__dirname + '/attachment1.jpg'),
    fs.createReadStream(__dirname + '/attachment2.jpg')
```

```
  ],
  // Pass optional meta-data with an 'options' object with style: {value: DATA, options: OPTIO
  // Use case: for some types of streams, you'll need to provide "file"-related information ma
  // See the `form-data` README for more information about options: https://github.com/form-da
  custom_file: {
    value:  fs.createReadStream('/dev/urandom'),
    options: {
      filename: 'topsecret.jpg',
      contentType: 'image/jpeg'
    }
  }
};
request.post({url:'http://service.com/upload', formData: formData}, function optionalCallback(
  if (err) {
    return console.error('upload failed:', err);
  }
  console.log('Upload successful!  Server responded with:', body);
});
```

For advanced cases, you can access the form-data object itself via `r.form()`. This can be modified until the request is fired on the next cycle of the event-loop. (Note that this calling `form()` will clear the currently set form data for that request.)

```
// NOTE: Advanced use-case, for normal use see 'formData' usage above
var r = request.post('http://service.com/upload', function optionalCallback(err, httpResponse,
var form = r.form();
```

```
form.append('my_field', 'my_value');
form.append('my_buffer', Buffer.from([1, 2, 3]));
form.append('custom_file', fs.createReadStream(__dirname + '/unicycle.jpg'), {filename: 'unicy
```

See the **form-data README** for more information & examples.

multipart/related

Some variations in different HTTP implementations require a newline/CRLF before, after, or both before and after the boundary of a `multipart/related` request (using the multipart option). This has been observed in the .NET WebAPI version 4.0. You can turn on a boundary preambleCRLF or postamble by passing them as `true` to your request options.

```
request({
  method: 'PUT',
  preambleCRLF: true,
  postambleCRLF: true,
  uri: 'http://service.com/upload',
  multipart: [
    {
      'content-type': 'application/json',
      body: JSON.stringify({foo: 'bar', _attachments: {'message.txt': {follows: true, length
    },
    { body: 'I am an attachment' },
    { body: fs.createReadStream('image.png') }
  ],
  // alternatively pass an object containing additional options
```

```
    multipart: {
      chunked: false,
      data: [
        {
          'content-type': 'application/json',
          body: JSON.stringify({foo: 'bar', _attachments: {'message.txt': {follows: true, leng
        },
        { body: 'I am an attachment' }
      ]
    }
  },
  function (error, response, body) {
    if (error) {
      return console.error('upload failed:', error);
    }
    console.log('Upload successful!  Server responded with:', body);
  })
```

back to top

HTTP Authentication

```
request.get('http://some.server.com/').auth('username', 'password', false);
// or
```

```
request.get('http://some.server.com/', {
  'auth': {
    'user': 'username',
    'pass': 'password',
    'sendImmediately': false
  }
});
// or
request.get('http://some.server.com/').auth(null, null, true, 'bearerToken');
// or
request.get('http://some.server.com/', {
  'auth': {
    'bearer': 'bearerToken'
  }
});
```

If passed as an option, `auth` should be a hash containing values:

- `user` || `username`
- `pass` || `password`
- `sendImmediately` (optional)
- `bearer` (optional)

The method form takes parameters `auth(username, password, sendImmediately, bearer)`.

`sendImmediately` defaults to `true`, which causes a basic or bearer authentication header to be sent. If `sendImmediately` is `false`, then `request` will retry with a proper authentication header after receiving a `401`

response from the server (which must contain a `WWW-Authenticate` header indicating the required authentication method).

Note that you can also specify basic authentication using the URL itself, as detailed in **RFC 1738**. Simply pass the `user:password` before the host with an `@` sign:

```
var username = 'username',
    password = 'password',
    url = 'http://' + username + ':' + password + '@some.server.com';

request({url: url}, function (error, response, body) {
   // Do more stuff with 'body' here
});
```

Digest authentication is supported, but it only works with `sendImmediately` set to `false`; otherwise `request` will send basic authentication on the initial request, which will probably cause the request to fail.

Bearer authentication is supported, and is activated when the `bearer` value is available. The value may be either a `String` or a `Function` returning a `String`. Using a function to supply the bearer token is particularly useful if used in conjunction with `defaults` to allow a single function to supply the last known token at the time of sending a request, or to compute one on the fly.

back to top

Custom HTTP Headers

HTTP Headers, such as `User-Agent`, can be set in the `options` object. In the example below, we call the github API to find out the number of stars and forks for the request repository. This requires a custom `User-Agent` header as well as https.

```
var request = require('request');

var options = {
  url: 'https://api.github.com/repos/request/request',
  headers: {
    'User-Agent': 'request'
  }
};

function callback(error, response, body) {
  if (!error && response.statusCode == 200) {
    var info = JSON.parse(body);
    console.log(info.stargazers_count + " Stars");
    console.log(info.forks_count + " Forks");
  }
}

request(options, callback);
```

back to top

OAuth Signing

OAuth version 1.0 is supported. The default signing algorithm is **HMAC-SHA1**:

```
// OAuth1.0 - 3-legged server side flow (Twitter example)
// step 1
var qs = require('querystring')
  , oauth =
    { callback: 'http://mysite.com/callback/'
    , consumer_key: CONSUMER_KEY
    , consumer_secret: CONSUMER_SECRET
    }
  , url = 'https://api.twitter.com/oauth/request_token'
  ;
request.post({url:url, oauth:oauth}, function (e, r, body) {
  // Ideally, you would take the body in the response
  // and construct a URL that a user clicks on (like a sign in button).
  // The verifier is only available in the response after a user has
  // verified with twitter that they are authorizing your app.

  // step 2
  var req_data = qs.parse(body)
  var uri = 'https://api.twitter.com/oauth/authenticate'
    + '?' + qs.stringify({oauth_token: req_data.oauth_token})
  // redirect the user to the authorize uri
```

```
// step 3
// after the user is redirected back to your server
var auth_data = qs.parse(body)
  , oauth =
    { consumer_key: CONSUMER_KEY
    , consumer_secret: CONSUMER_SECRET
    , token: auth_data.oauth_token
    , token_secret: req_data.oauth_token_secret
    , verifier: auth_data.oauth_verifier
    }
  , url = 'https://api.twitter.com/oauth/access_token'
  ;
request.post({url:url, oauth:oauth}, function (e, r, body) {
  // ready to make signed requests on behalf of the user
  var perm_data = qs.parse(body)
    , oauth =
      { consumer_key: CONSUMER_KEY
      , consumer_secret: CONSUMER_SECRET
      , token: perm_data.oauth_token
      , token_secret: perm_data.oauth_token_secret
      }
    , url = 'https://api.twitter.com/1.1/users/show.json'
    , qs =
      { screen_name: perm_data.screen_name
      , user_id: perm_data.user_id
      }
```

```
      ;
    request.get({url:url, oauth:oauth, qs:qs, json:true}, function (e, r, user) {
      console.log(user)
    })
  })
})
```

For **RSA-SHA1 signing**, make the following changes to the OAuth options object:

- Pass `signature_method : 'RSA-SHA1'`
- Instead of `consumer_secret` , specify a `private_key` string in PEM format

For **PLAINTEXT signing**, make the following changes to the OAuth options object:

- Pass `signature_method : 'PLAINTEXT'`

To send OAuth parameters via query params or in a post body as described in The **Consumer Request Parameters** section of the oauth1 spec:

- Pass `transport_method : 'query'` or `transport_method : 'body'` in the OAuth options object.
- `transport_method` defaults to `'header'`

To use **Request Body Hash** you can either

- Manually generate the body hash and pass it as a string `body_hash: '...'`
- Automatically generate the body hash by passing `body_hash: true`

back to top

Proxies

If you specify a `proxy` option, then the request (and any subsequent redirects) will be sent via a connection to the proxy server.

If your endpoint is an `https` url, and you are using a proxy, then request will send a `CONNECT` request to the proxy server *first*, and then use the supplied connection to connect to the endpoint.

That is, first it will make a request like:

```
HTTP/1.1 CONNECT endpoint-server.com:80
Host: proxy-server.com
User-Agent: whatever user agent you specify
```

and then the proxy server make a TCP connection to `endpoint-server` on port `80`, and return a response that looks like:

```
HTTP/1.1 200 OK
```

At this point, the connection is left open, and the client is communicating directly with the `endpoint-server.com` machine.

See **the wikipedia page on HTTP Tunneling** for more information.

By default, when proxying `http` traffic, request will simply make a standard proxied `http` request. This is done by making the `url` section of the initial line of the request a fully qualified url to the endpoint.

For example, it will make a single request that looks like:

```
HTTP/1.1 GET http://endpoint-server.com/some-url
Host: proxy-server.com
Other-Headers: all go here

request body or whatever
```

Because a pure "http over http" tunnel offers no additional security or other features, it is generally simpler to go with a straightforward HTTP proxy in this case. However, if you would like to force a tunneling proxy, you may set the `tunnel` option to `true` .

You can also make a standard proxied `http` request by explicitly setting `tunnel : false` , but **note that this will allow the proxy to see the traffic to/from the destination server**.

If you are using a tunneling proxy, you may set the `proxyHeaderWhiteList` to share certain headers with the proxy.

You can also set the `proxyHeaderExclusiveList` to share certain headers only with the proxy and not with destination host.

By default, this set is:

```
accept
accept-charset
accept-encoding
accept-language
accept-ranges
cache-control
content-encoding
```

```
content-language
content-length
content-location
content-md5
content-range
content-type
connection
date
expect
max-forwards
pragma
proxy-authorization
referer
te
transfer-encoding
user-agent
via
```

Note that, when using a tunneling proxy, the `proxy-authorization` header and any headers from custom `proxyHeaderExclusiveList` are *never* sent to the endpoint server, but only to the proxy server.

Controlling proxy behaviour using environment variables

The following environment variables are respected by `request`:

- `HTTP_PROXY` / `http_proxy`
- `HTTPS_PROXY` / `https_proxy`

- `NO_PROXY` / `no_proxy`

When `HTTP_PROXY` / `http_proxy` are set, they will be used to proxy non-SSL requests that do not have an explicit `proxy` configuration option present. Similarly, `HTTPS_PROXY` / `https_proxy` will be respected for SSL requests that do not have an explicit `proxy` configuration option. It is valid to define a proxy in one of the environment variables, but then override it for a specific request, using the `proxy` configuration option. Furthermore, the `proxy` configuration option can be explicitly set to false / null to opt out of proxying altogether for that request.

`request` is also aware of the `NO_PROXY` / `no_proxy` environment variables. These variables provide a granular way to opt out of proxying, on a per-host basis. It should contain a comma separated list of hosts to opt out of proxying. It is also possible to opt of proxying when a particular destination port is used. Finally, the variable may be set to `*` to opt out of the implicit proxy configuration of the other environment variables.

Here's some examples of valid `no_proxy` values:

- `google.com` - don't proxy HTTP/HTTPS requests to Google.
- `google.com:443` - don't proxy HTTPS requests to Google, but *do* proxy HTTP requests to Google.
- `google.com:443, yahoo.com:80` - don't proxy HTTPS requests to Google, and don't proxy HTTP requests to Yahoo!
- `*` - ignore `https_proxy` / `http_proxy` environment variables altogether.

back to top

UNIX Domain Sockets

`request` supports making requests to **UNIX Domain Sockets**. To make one, use the following URL scheme:

```
/* Pattern */ 'http://unix:SOCKET:PATH'
```

```
/* Example */ request.get('http://unix:/absolute/path/to/unix.socket:/request/path')
```

Note: The `SOCKET` path is assumed to be absolute to the root of the host file system.

back to top

TLS/SSL Protocol

TLS/SSL Protocol options, such as `cert`, `key` and `passphrase`, can be set directly in `options` object, in the `agentOptions` property of the `options` object, or even in `https.globalAgent.options`. Keep in mind that, although `agentOptions` allows for a slightly wider range of configurations, the recommended way is via `options` object directly, as using `agentOptions` or `https.globalAgent.options` would not be applied in the same way in proxied environments (as data travels through a TLS connection instead of an http/https agent).

```
var fs = require('fs')
    , path = require('path')
    , certFile = path.resolve(__dirname, 'ssl/client.crt')
    , keyFile = path.resolve(__dirname, 'ssl/client.key')
    , caFile = path.resolve(__dirname, 'ssl/ca.cert.pem')
    , request = require('request');

var options = {
    url: 'https://api.some-server.com/',
    cert: fs.readFileSync(certFile),
    key: fs.readFileSync(keyFile),
    passphrase: 'password',
```

```
    ca: fs.readFileSync(caFile)
};

request.get(options);
```

Using `options.agentOptions`

In the example below, we call an API that requires client side SSL certificate (in PEM format) with passphrase protected private key (in PEM format) and disable the SSLv3 protocol:

```
var fs = require('fs')
    , path = require('path')
    , certFile = path.resolve(__dirname, 'ssl/client.crt')
    , keyFile = path.resolve(__dirname, 'ssl/client.key')
    , request = require('request');

var options = {
    url: 'https://api.some-server.com/',
    agentOptions: {
        cert: fs.readFileSync(certFile),
        key: fs.readFileSync(keyFile),
        // Or use `pfx` property replacing `cert` and `key` when using private key, certificat
        // pfx: fs.readFileSync(pfxFilePath),
        passphrase: 'password',
        securityOptions: 'SSL_OP_NO_SSLv3'
    }
};
```

```
request.get(options);
```

It is able to force using SSLv3 only by specifying `secureProtocol` :

```
request.get({
    url: 'https://api.some-server.com/',
    agentOptions: {
        secureProtocol: 'SSLv3_method'
    }
});
```

It is possible to accept other certificates than those signed by generally allowed Certificate Authorities (CAs). This can be useful, for example, when using self-signed certificates. To require a different root certificate, you can specify the signing CA by adding the contents of the CA's certificate file to the `agentOptions` . The certificate the domain presents must be signed by the root certificate specified:

```
request.get({
    url: 'https://api.some-server.com/',
    agentOptions: {
        ca: fs.readFileSync('ca.cert.pem')
    }
});
```

back to top

Support for HAR 1.2

The `options.har` property will override the values: `url`, `method`, `qs`, `headers`, `form`, `formData`, `body`, `json`, as well as construct multipart data and read files from disk when `request.postData.params[].fileName` is present without a matching `value`.

A validation step will check if the HAR Request format matches the latest spec (v1.2) and will skip parsing if not matching.

```
var request = require('request')
request({
  // will be ignored
  method: 'GET',
  uri: 'http://www.google.com',

  // HTTP Archive Request Object
  har: {
    url: 'http://www.mockbin.com/har',
    method: 'POST',
    headers: [
      {
        name: 'content-type',
        value: 'application/x-www-form-urlencoded'
      }
    ],
```

```
    postData: {
      mimeType: 'application/x-www-form-urlencoded',
      params: [
        {
          name: 'foo',
          value: 'bar'
        },
        {
          name: 'hello',
          value: 'world'
        }
      ]
    }
  }
})

// a POST request will be sent to http://www.mockbin.com
// with body an application/x-www-form-urlencoded body:
// foo=bar&hello=world
```

back to top

request(options, callback)

The first argument can be either a `url` or an `options` object. The only required option is `uri` ; all others are optional.

- `uri` || `url` - fully qualified uri or a parsed url object from `url.parse()`
- `baseUrl` - fully qualified uri string used as the base url. Most useful with `request.defaults`, for example when you want to do many requests to the same domain. If `baseUrl` is `https://example.com/api/`, then requesting `/end/point?test=true` will fetch `https://example.com/api/end/point?test=true`. When `baseUrl` is given, `uri` must also be a string.
- `method` - http method (default: `"GET"`)
- `headers` - http headers (default: `{}`)

- `qs` - object containing querystring values to be appended to the `uri`
- `qsParseOptions` - object containing options to pass to the qs.parse method. Alternatively pass options to the querystring.parse method using this format `{sep:';', eq:':', options:{}}`
- `qsStringifyOptions` - object containing options to pass to the qs.stringify method. Alternatively pass options to the querystring.stringify method using this format `{sep:';', eq:':', options:{}}`. For example, to change the way arrays are converted to query strings using the `qs` module pass the `arrayFormat` option with one of `indices|brackets|repeat`
- `useQuerystring` - if true, use `querystring` to stringify and parse querystrings, otherwise use `qs` (default: `false`). Set this option to `true` if you need arrays to be serialized as `foo=bar&foo=baz` instead of the default `foo[0]=bar&foo[1]=baz`.

- `body` - entity body for PATCH, POST and PUT requests. Must be a `Buffer`, `String` or `ReadStream`. If `json` is `true`, then `body` must be a JSON-serializable object.
- `form` - when passed an object or a querystring, this sets `body` to a querystring representation of value, and adds `Content-type: application/x-www-form-urlencoded` header. When passed no options, a `FormData` instance is returned (and is piped to request). See "Forms" section above.
- `formData` - data to pass for a `multipart/form-data` request. See Forms section above.
- `multipart` - array of objects which contain their own headers and `body` attributes. Sends a `multipart/related` request. See Forms section above.

  - Alternatively you can pass in an object `{chunked: false, data: []}` where `chunked` is used to specify whether the request is sent in chunked transfer encoding In non-chunked requests, data items with body streams are not allowed.
- `preambleCRLF` - append a newline/CRLF before the boundary of your `multipart/form-data` request.
- `postambleCRLF` - append a newline/CRLF at the end of the boundary of your `multipart/form-data` request.
- `json` - sets `body` to JSON representation of value and adds `Content-type: application/json` header. Additionally, parses the response body as JSON.
- `jsonReviver` - a reviver function that will be passed to `JSON.parse()` when parsing a JSON response body.
- `jsonReplacer` - a replacer function that will be passed to `JSON.stringify()` when stringifying a JSON request body.

- `auth` - a hash containing values `user` || `username`, `pass` || `password`, and `sendImmediately` (optional). See documentation above.
- `oauth` - options for OAuth HMAC-SHA1 signing. See documentation above.
- `hawk` - options for Hawk signing. The `credentials` key must contain the necessary signing info, see hawk docs for details.
- `aws` - `object` containing AWS signing information. Should have the properties `key`, `secret`, and optionally `session` (note that this only works for services that require session as part of the canonical string). Also requires the property `bucket`, unless you're specifying your `bucket` as part of the path, or the request doesn't use a bucket (i.e. GET Services). If you want to use AWS sign version 4 use the parameter `sign_version` with value `4` otherwise the default is version 2. If you are using SigV4, you can also include a `service` property that specifies the service name. **Note:** you need to `npm install aws4` first.
- `httpSignature` - options for the HTTP Signature Scheme using Joyent's library. The `keyId` and `key` properties must be specified. See the docs for other options.

- `followRedirect` - follow HTTP 3xx responses as redirects (default: `true`). This property can also be implemented as function which gets `response` object as a single argument and should return `true` if redirects should continue or

`false` otherwise.

- `followAllRedirects` - follow non-GET HTTP 3xx responses as redirects (default: `false`)
- `followOriginalHttpMethod` - by default we redirect to HTTP method GET. you can enable this property to redirect to the original HTTP method (default: `false`)
- `maxRedirects` - the maximum number of redirects to follow (default: `10`)
- `removeRefererHeader` - removes the referer header when a redirect happens (default: `false`). **Note:** if true, referer header set in the initial request is preserved during redirect chain.

- `encoding` - encoding to be used on `setEncoding` of response data. If `null` , the `body` is returned as a `Buffer` . Anything else **(including the default value of `undefined`)** will be passed as the encoding parameter to `toString()` (meaning this is effectively `utf8` by default). (**Note:** if you expect binary data, you should set `encoding: null` .)
- `gzip` - if `true` , add an `Accept-Encoding` header to request compressed content encodings from the server (if not already present) and decode supported content encodings in the response. **Note:** Automatic decoding of the response content is performed on the body data returned through `request` (both through the `request` stream and passed to the callback function) but is not performed on the `response` stream (available from the `response` event) which is the unmodified `http.IncomingMessage` object which may contain compressed data. See example below.
- `jar` - if `true` , remember cookies for future use (or define your custom cookie jar; see examples section)

- `agent` - `http(s).Agent` instance to use
- `agentClass` - alternatively specify your agent's class name
- `agentOptions` - and pass its options. **Note:** for HTTPS see tls API doc for TLS/SSL options and the documentation above.
- `forever` - set to `true` to use the forever-agent **Note:** Defaults to `http(s).Agent({keepAlive:true})` in node 0.12+
- `pool` - an object describing which agents to use for the request. If this option is omitted the request will use the global agent (as long as your options allow for it). Otherwise, request will search the pool for your custom agent. If no custom

agent is found, a new agent will be created and added to the pool. **Note:** `pool` is used only when the `agent` option is not specified.
  - A `maxSockets` property can also be provided on the `pool` object to set the max number of sockets for all agents created (ex: `pool: {maxSockets: Infinity}`).
  - Note that if you are sending multiple requests in a loop and creating multiple new `pool` objects, `maxSockets` will not work as intended. To work around this, either use `request.defaults` with your pool options or create the pool object with the `maxSockets` property outside of the loop.
- `timeout` - integer containing the number of milliseconds to wait for a server to send response headers (and start the response body) before aborting the request. Note that if the underlying TCP connection cannot be established, the OS-wide TCP connection timeout will overrule the `timeout` option (the default in Linux can be anywhere from 20-120 seconds).

- `localAddress` - local interface to bind for network connections.
- `proxy` - an HTTP proxy to be used. Supports proxy Auth with Basic Auth, identical to support for the `url` parameter (by embedding the auth info in the `uri`)
- `strictSSL` - if `true` , requires SSL certificates be valid. **Note:** to use your own certificate authority, you need to specify an agent that was created with that CA as an option.
- `tunnel` - controls the behavior of `HTTP CONNECT` tunneling as follows:
  - `undefined` (default) - `true` if the destination is `https` , `false` otherwise
  - `true` - always tunnel to the destination by making a `CONNECT` request to the proxy
  - `false` - request the destination as a `GET` request.
- `proxyHeaderWhiteList` - a whitelist of headers to send to a tunneling proxy.
- `proxyHeaderExclusiveList` - a whitelist of headers to send exclusively to a tunneling proxy and not to destination.

- `time` - if `true` , the request-response cycle (including all redirects) is timed at millisecond resolution. When set, the following properties are added to the response object:
  - `elapsedTime` Duration of the entire request/response in milliseconds (*deprecated*).

- `responseStartTime` Timestamp when the response began (in Unix Epoch milliseconds) (*deprecated*).
- `timingStart` Timestamp of the start of the request (in Unix Epoch milliseconds).
- `timings` Contains event timestamps in millisecond resolution relative to `timingStart`. If there were redirects, the properties reflect the timings of the final request in the redirect chain:
  - `socket` Relative timestamp when the `http` module's `socket` event fires. This happens when the socket is assigned to the request.
  - `lookup` Relative timestamp when the `net` module's `lookup` event fires. This happens when the DNS has been resolved.
  - `connect`: Relative timestamp when the `net` module's `connect` event fires. This happens when the server acknowledges the TCP connection.
  - `response`: Relative timestamp when the `http` module's `response` event fires. This happens when the first bytes are received from the server.
  - `end`: Relative timestamp when the last bytes of the response are received.
- `timingPhases` Contains the durations of each request phase. If there were redirects, the properties reflect the timings of the final request in the redirect chain:
  - `wait`: Duration of socket initialization (`timings.socket`)
  - `dns`: Duration of DNS lookup (`timings.lookup - timings.socket`)
  - `tcp`: Duration of TCP connection (`timings.connect - timings.socket`)
  - `firstByte`: Duration of HTTP server response (`timings.response - timings.connect`)
  - `download`: Duration of HTTP download (`timings.end - timings.response`)
  - `total`: Duration entire HTTP round-trip (`timings.end`)

- `har` - a HAR 1.2 Request Object, will be processed from HAR format into options overwriting matching values *(see the HAR 1.2 section for details)*

- `callback` - alternatively pass the request's callback in the options object

The callback argument gets 3 arguments:

1. An `error` when applicable (usually from `http.ClientRequest` object)
2. An `http.IncomingMessage` object (Response object)
3. The third is the `response` body (`String` or `Buffer` , or JSON object if the `json` option is supplied)

back to top

Convenience methods

There are also shorthand methods for different HTTP METHODs and some other conveniences.

request.defaults(options)

This method **returns a wrapper** around the normal request API that defaults to whatever options you pass to it.

Note: `request.defaults()` **does not** modify the global request API; instead, it **returns a wrapper** that has your default settings applied to it.

Note: You can call `.defaults()` on the wrapper that is returned from `request.defaults` to add/override defaults that were previously defaulted.

For example:

```
//requests using baseRequest() will set the 'x-token' header
var baseRequest = request.defaults({
  headers: {'x-token': 'my-token'}
})

//requests using specialRequest() will include the 'x-token' header set in
```

```
//baseRequest and will also include the 'special' header
var specialRequest = baseRequest.defaults({
  headers: {special: 'special value'}
})
```

request.METHOD()

These HTTP method convenience functions act just like `request()` but with a default method already set for you:

- *request.get()*: Defaults to `method: "GET"`.
- *request.post()*: Defaults to `method: "POST"`.
- *request.put()*: Defaults to `method: "PUT"`.
- *request.patch()*: Defaults to `method: "PATCH"`.
- *request.del() / request.delete()*: Defaults to `method: "DELETE"`.
- *request.head()*: Defaults to `method: "HEAD"`.
- *request.options()*: Defaults to `method: "OPTIONS"`.

request.cookie()

Function that creates a new cookie.

```
request.cookie('key1=value1')
```

request.jar()

Function that creates a new cookie jar.

```
request.jar()
```

back to top

Debugging

There are at least three ways to debug the operation of `request`:

1. Launch the node process like `NODE_DEBUG=request node script.js` (`lib,request,otherlib` works too).
2. Set `require('request').debug = true` at any time (this does the same thing as #1).
3. Use the **request-debug module** to view request and response headers and bodies.

back to top

Timeouts

Most requests to external servers should have a timeout attached, in case the server is not responding in a timely manner. Without a timeout, your code may have a socket open/consume resources for minutes or more.

There are two main types of timeouts: **connection timeouts** and **read timeouts**. A connect timeout occurs if the timeout is hit while your client is attempting to establish a connection to a remote machine (corresponding to the **connect() call** on the socket). A read timeout occurs any time the server is too slow to send back a part of the response.

These two situations have widely different implications for what went wrong with the request, so it's useful to be able to distinguish them. You can detect timeout errors by checking `err.code` for an 'ETIMEDOUT' value. Further, you can detect whether the timeout was a connection timeout by checking if the `err.connect` property is set to `true`.

```
request.get('http://10.255.255.1', {timeout: 1500}, function(err) {
```

```
    console.log(err.code === 'ETIMEDOUT');
    // Set to `true` if the timeout was a connection timeout, `false` or
    // `undefined` otherwise.
    console.log(err.connect === true);
    process.exit(0);
});
```

Examples:

```
var request = require('request')
  , rand = Math.floor(Math.random()*100000000).toString()
  ;
request(
  { method: 'PUT'
  , uri: 'http://mikeal.iriscouch.com/testjs/' + rand
  , multipart:
    [ { 'content-type': 'application/json'
      ,  body: JSON.stringify({foo: 'bar', _attachments: {'message.txt': {follows: true, len
      }
    , { body: 'I am an attachment' }
    ]
  }
, function (error, response, body) {
    if(response.statusCode == 201){
      console.log('document saved as: http://mikeal.iriscouch.com/testjs/'+ rand)
```

```
    } else {
      console.log('error: '+ response.statusCode)
      console.log(body)
    }
  }
)
```

For backwards-compatibility, response compression is not supported by default. To accept gzip-compressed responses, set the `gzip` option to `true` . Note that the body data passed through `request` is automatically decompressed while the response object is unmodified and will contain compressed data if the server sent a compressed response.

```
var request = require('request')
request(
  { method: 'GET'
  , uri: 'http://www.google.com'
  , gzip: true
  }
, function (error, response, body) {
    // body is the decompressed response body
    console.log('server encoded the data as: ' + (response.headers['content-encoding'] || 'i
    console.log('the decoded data is: ' + body)
  }
)
.on('data', function(data) {
  // decompressed data as it is received
```

```
    console.log('decoded chunk: ' + data)
  })
  .on('response', function(response) {
    // unmodified http.IncomingMessage object
    response.on('data', function(data) {
      // compressed data as it is received
      console.log('received ' + data.length + ' bytes of compressed data')
    })
  })
```

Cookies are disabled by default (else, they would be used in subsequent requests). To enable cookies, set `jar` to `true` (either in `defaults` or `options`).

```
var request = request.defaults({jar: true})
request('http://www.google.com', function () {
  request('http://images.google.com')
})
```

To use a custom cookie jar (instead of `request`'s global cookie jar), set `jar` to an instance of `request.jar()` (either in `defaults` or `options`)

```
var j = request.jar()
var request = request.defaults({jar:j})
request('http://www.google.com', function () {
  request('http://images.google.com')
```

```
})
```

OR

```
var j = request.jar();
var cookie = request.cookie('key1=value1');
var url = 'http://www.google.com';
j.setCookie(cookie, url);
request({url: url, jar: j}, function () {
  request('http://images.google.com')
})
```

To use a custom cookie store (such as a `FileCookieStore` which supports saving to and restoring from JSON files), pass it as a parameter to `request.jar()`:

```
var FileCookieStore = require('tough-cookie-filestore');
// NOTE - currently the 'cookies.json' file must already exist!
var j = request.jar(new FileCookieStore('cookies.json'));
request = request.defaults({ jar : j })
request('http://www.google.com', function() {
  request('http://images.google.com')
})
```

The cookie store must be a `tough-cookie` store and it must support synchronous operations; see the `CookieStore API docs` for details.

To inspect your cookie jar after a request:

```
var j = request.jar()
request({url: 'http://www.google.com', jar: j}, function () {
  var cookie_string = j.getCookieString(url); // "key1=value1; key2=value2; ..."
  var cookies = j.getCookies(url);
  // [{key: 'key1', value: 'value1', domain: "www.google.com", ...}, ...]
})
```

back to top

Keywords

http simple util utility

HELP

Documentation

Resources

Support / Contact Us

Registry Status

Report Issues

npm Community Site

Security

ABOUT NPM

About npm, Inc

JavaScriptSurvey.com

Events

Jobs

Press

npm Weekly

Blog

Twitter

GitHub

TERMS & POLICIES

Terms of Use

Code of Conduct

Package Name Disputes

Privacy Policy

Reporting Abuse

Other policies