Node.js v5.11.1 Documentation

Table of Contents

HTTP#

```
Stability: 2 - Stable
```

To use the HTTP server and client one must `require('http')`.

The HTTP interfaces in Node.js are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages. The interface is careful to never buffer entire requests or responses--the user is able to stream data.

HTTP message headers are represented by an object like this:

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'host': 'mysite.com',
  'accept': '*/*' }
```

Keys are lowercased. Values are not modified.

In order to support the full spectrum of possible HTTP applications, Node.js's HTTP API is very low-level. It deals with stream handling and message parsing only. It parses a message into headers and body but it does not parse the actual headers or the body.

See `message.headers` for details on how duplicate headers are handled.

The raw headers as they were received are retained in the `rawHeaders` property, which is an array of `[key, value, key2, value2, ...]`. For example, the previous message header object might have a `rawHeaders` list like the following:

```
[ 'ConTent-Length', '123456',
  'content-LENGTH', '123',
  'content-type', 'text/plain',
  'CONNECTION', 'keep-alive',
  'Host', 'mysite.com',
  'accepT', '*/*' ]
```

Class: http.Agent#

The HTTP Agent is used for pooling sockets used in HTTP client requests.

The HTTP Agent also defaults client requests to using Connection:keep-alive. If no pending HTTP requests are waiting on a socket to become free the socket is closed. This means that Node.js's pool has the benefit of keep-alive when under load but still does not require developers to manually close the HTTP clients using KeepAlive.

If you opt into using HTTP KeepAlive, you can create an Agent object with that flag set to `true`. (See the constructor options.) Then, the Agent will keep unused sockets in a pool for later use. They will be explicitly marked so as to not keep the Node.js process running. However, it is still a good idea to explicitly `destroy()` KeepAlive agents when they are no longer in use, so that the Sockets will be shut down.

Sockets are removed from the agent's pool when the socket emits either a `'close'` event or a special `'agentRemove'` event. This means that if you intend to keep one HTTP request open for a long time and don't want it to stay in the pool you can do something along the lines of:

```
http.get(options, (res) => {
  // Do stuff
}).on('socket', (socket) => {
  socket.emit('agentRemove');
});
```

Alternatively, you could just opt out of pooling entirely using `agent:false`:

```
http.get({
  hostname: 'localhost',
  port: 80,
  path: '/',
  agent: false  // create a new agent just for this one request
}, (res) => {
  // Do stuff with response
})
```

new Agent([options])#

- `options` <Object> Set of configurable options to set on the agent. Can have the following fields:
  - `keepAlive` <Boolean> Keep sockets around in a pool to be used by other requests in the future. Default = `false`
  - `keepAliveMsecs` <Integer> When using HTTP KeepAlive, how often to send TCP KeepAlive packets over sockets being kept alive. Default = `1000`. Only relevant if `keepAlive` is set to `true`.
  - `maxSockets` <Number> Maximum number of sockets to allow per host. Default = `Infinity`.
  - `maxFreeSockets` <Number> Maximum number of sockets to leave open in a free state. Only relevant if `keepAlive` is set to `true`. Default = `256`.

The default `http.globalAgent` that is used by `http.request()` has all of these values set to their respective defaults.

To configure any of them, you must create your own `http.Agent` object.

```
const http = require('http');
var keepAliveAgent = new http.Agent({ keepAlive: true });
options.agent = keepAliveAgent;
http.request(options, onResponseCallback);
```

agent.createConnection(options[, callback])#

Produces a socket/stream to be used for HTTP requests.

By default, this function is the same as `net.createConnection()`. However, custom Agents may override this method in case greater flexibility is desired.

A socket/stream can be supplied in one of two ways: by returning the socket/stream from this function, or by passing the socket/stream to `callback`.

`callback` has a signature of `(err, stream)`.

agent.destroy()#

Destroy any sockets that are currently in use by the agent.

It is usually not necessary to do this. However, if you are using an agent with KeepAlive enabled, then it is best to explicitly shut down the agent when you know that it will no longer be used. Otherwise, sockets may hang open for quite a long time before the server terminates them.

agent.freeSockets#

An object which contains arrays of sockets currently awaiting use by the Agent when HTTP KeepAlive is used. Do not modify.

agent.getName(options)#

Get a unique name for a set of request options, to determine whether a connection can be reused. In the http agent, this returns `host:port:localAddress`. In the https agent, the name includes the CA, cert, ciphers, and other HTTPS/TLS-specific options that determine socket reusability.

Options:

- `host`: A domain name or IP address of the server to issue the request to.
- `port`: Port of remote server.
- `localAddress`: Local interface to bind for network connections when issuing the request.

agent.maxFreeSockets#

By default set to 256. For Agents supporting HTTP KeepAlive, this sets the maximum number of sockets that will be left open in the free state.

agent.maxSockets#

By default set to Infinity. Determines how many concurrent sockets the agent can have open per origin. Origin is either a 'host:port' or 'host:port:localAddress' combination.

agent.requests#

An object which contains queues of requests that have not yet been assigned to sockets. Do not modify.

agent.sockets#

An object which contains arrays of sockets currently in use by the Agent. Do not modify.

Class: http.ClientRequest#

This object is created internally and returned from `http.request()`. It represents an *in-progress* request whose header has already been queued. The header is still mutable using the `setHeader(name, value)`, `getHeader(name)`, `removeHeader(name)` API. The actual header will be sent along with the first data chunk or when closing the connection.

To get the response, add a listener for `'response'` to the request object. `'response'` will be emitted from the request object when the response headers have been received. The `'response'` event is executed with one argument which is an instance of `http.IncomingMessage`.

During the `'response'` event, one can add listeners to the response object; particularly to listen for the `'data'` event.

If no `'response'` handler is added, then the response will be entirely discarded. However, if you add a `'response'` event handler, then you **must** consume the data from the response object, either by calling `response.read()` whenever there is a `'readable'` event, or by adding a `'data'` handler, or by calling the `.resume()` method. Until the data is consumed, the `'end'` event will not fire. Also, until the data is read it will consume memory that can eventually lead to a 'process out of memory' error.

Note: Node.js does not check whether Content-Length and the length of the body which has been transmitted are equal or not.

The request implements the Writable Stream interface. This is an `EventEmitter` with the following events:

Event: 'abort'#

```
function () { }
```

Emitted when the request has been aborted by the client. This event is only emitted on the first call to `abort()`.

Event: 'checkExpectation'#

```
function (request, response) { }
```

Emitted each time a request with an http Expect header is received, where the value is not 100-continue. If this event isn't listened for, the server will automatically respond with a 417 Expectation Failed as appropriate.

Note that when this event is emitted and handled, the `request` event will not be emitted.

Event: 'connect'#

```
function (response, socket, head) { }
```

Emitted each time a server responds to a request with a `CONNECT` method. If this event isn't being listened for, clients receiving a `CONNECT` method will have their connections closed.

A client server pair that show you how to listen for the `'connect'` event.

```
const http = require('http');
const net = require('net');
const url = require('url');

// Create an HTTP tunneling proxy
var proxy = http.createServer( (req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('okay');
});
proxy.on('connect', (req, cltSocket, head) => {
  // connect to an origin server
  var srvUrl = url.parse(`http://${req.url}`);
  var srvSocket = net.connect(srvUrl.port, srvUrl.hostname, () => {
    cltSocket.write('HTTP/1.1 200 Connection Established\r\n' +
                    'Proxy-agent: Node.js-Proxy\r\n' +
                    '\r\n');
    srvSocket.write(head);
    srvSocket.pipe(cltSocket);
    cltSocket.pipe(srvSocket);
  });
});

// now that proxy is running
proxy.listen(1337, '127.0.0.1', () => {

  // make a request to a tunneling proxy
  var options = {
    port: 1337,
    hostname: '127.0.0.1',
    method: 'CONNECT',
    path: 'www.google.com:80'
  };

  var req = http.request(options);
  req.end();

  req.on('connect', (res, socket, head) => {
    console.log('got connected!');

    // make a request over an HTTP tunnel
    socket.write('GET / HTTP/1.1\r\n' +
                 'Host: www.google.com:80\r\n' +
                 'Connection: close\r\n' +
                 '\r\n');
    socket.on('data', (chunk) => {
      console.log(chunk.toString());
    });
    socket.on('end', () => {
      proxy.close();
    });
  });
});
```

Event: 'continue'#

`function () { }`

Emitted when the server sends a '100 Continue' HTTP response, usually because the request contained 'Expect: 100-continue'. This is an instruction that the client should send the request body.

Event: 'response'#

`function (response) { }`

Emitted when a response is received to this request. This event is emitted only once. The `response` argument will be an instance of `http.IncomingMessage`.

Event: 'socket'#

`function (socket) { }`

Emitted after a socket is assigned to this request.

Event: 'upgrade'#

```
function (response, socket, head) { }
```

Emitted each time a server responds to a request with an upgrade. If this event isn't being listened for, clients receiving an upgrade header will have their connections closed.

A client server pair that show you how to listen for the `'upgrade'` event.

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

// Create an HTTP server
var srv = http.createServer( (req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('okay');
});
srv.on('upgrade', (req, socket, head) => {
  socket.write('HTTP/1.1 101 Web Socket Protocol Handshake\r\n' +
               'Upgrade: WebSocket\r\n' +
               'Connection: Upgrade\r\n' +
               '\r\n');

  socket.pipe(socket); // echo back
});

// now that server is running
srv.listen(1337, '127.0.0.1', () => {

  // make a request
  var options = {
    port: 1337,
    hostname: '127.0.0.1',
    headers: {
      'Connection': 'Upgrade',
      'Upgrade': 'websocket'
    }
  };

  var req = http.request(options);
  req.end();

  req.on('upgrade', (res, socket, upgradeHead) => {
    console.log('got upgraded!');
    socket.end();
    process.exit(0);
  });
});
```

request.abort()#

Marks the request as aborting. Calling this will cause remaining data in the response to be dropped and the socket to be destroyed.

request.end([data][, encoding][, callback])#

Finishes sending the request. If any parts of the body are unsent, it will flush them to the stream. If the request is chunked, this will send the terminating `'0\r\n\r\n'`.

If `data` is specified, it is equivalent to calling `response.write(data, encoding)` followed by `request.end(callback)`.

If `callback` is specified, it will be called when the request stream is finished.

request.flushHeaders()#

Flush the request headers.

For efficiency reasons, Node.js normally buffers the request headers until you call `request.end()` or write the first chunk of request data. It then tries hard to pack the request headers and data into a single TCP packet.

That's usually what you want (it saves a TCP round-trip) but not when the first data isn't sent until possibly much later. `request.flushHeaders()` lets you bypass the optimization and kickstart the request.

request.setNoDelay([noDelay])#

Once a socket is assigned to this request and is connected `socket.setNoDelay()` will be called.

request.setSocketKeepAlive([enable][, initialDelay])#

Once a socket is assigned to this request and is connected `socket.setKeepAlive()` will be called.

request.setTimeout(timeout[, callback])#

Once a socket is assigned to this request and is connected `socket.setTimeout()` will be called.

- `timeout` <Number> Milliseconds before a request is considered to be timed out.
- `callback` <Function> Optional function to be called when a timeout occurs. Same as binding to the `timeout` event.

request.write(chunk[, encoding][, callback])#

Sends a chunk of the body. By calling this method many times, the user can stream a request body to a server--in that case it is suggested to use the `['Transfer-Encoding', 'chunked']` header line when creating the request.

The `chunk` argument should be a `Buffer` or a string.

The `encoding` argument is optional and only applies when `chunk` is a string. Defaults to `'utf8'`.

The `callback` argument is optional and will be called when this chunk of data is flushed.

Returns `request`.

Class: http.Server#

This class inherits from `net.Server` and has the following additional events:

Event: 'checkContinue'#

```
function (request, response) { }
```

Emitted each time a request with an http Expect: 100-continue is received. If this event isn't listened for, the server will automatically respond with a 100 Continue as appropriate.

Handling this event involves calling `response.writeContinue()` if the client should continue to send the request body, or generating an appropriate HTTP response (e.g., 400 Bad Request) if the client should not continue to send the request body.

Note that when this event is emitted and handled, the `'request'` event will not be emitted.

Event: 'clientError'#

```
function (exception, socket) { }
```

If a client connection emits an `'error'` event, it will be forwarded here.

`socket` is the `net.Socket` object that the error originated from.

Event: 'close'#

```
function () { }
```

Emitted when the server closes.

Event: 'connect'#

```
function (request, socket, head) { }
```

Emitted each time a client requests a http `CONNECT` method. If this event isn't listened for, then clients requesting a `CONNECT` method will have their connections closed.

- `request` is the arguments for the http request, as it is in the request event.
- `socket` is the network socket between the server and client.
- `head` is an instance of Buffer, the first packet of the tunneling stream, this may be empty.

After this event is emitted, the request's socket will not have a `'data'` event listener, meaning you will need to bind to it in order to handle data sent to the server on that socket.

Event: 'connection'#

```
function (socket) { }
```

When a new TCP stream is established. `socket` is an object of type `net.Socket`. Usually users will not want to access this event. In particular, the socket will not emit `'readable'` events because of how the protocol parser attaches to the socket. The `socket` can also be accessed at `request.connection`.

Event: 'request'#

```
function (request, response) { }
```

Emitted each time there is a request. Note that there may be multiple requests per connection (in the case of keep-alive connections). `request` is an instance of `http.IncomingMessage` and `response` is an instance of `http.ServerResponse`.

Event: 'upgrade'#

```
function (request, socket, head) { }
```

Emitted each time a client requests a http upgrade. If this event isn't listened for, then clients requesting an upgrade will have their connections closed.

- `request` is the arguments for the http request, as it is in the request event.
- `socket` is the network socket between the server and client.
- `head` is an instance of Buffer, the first packet of the upgraded stream, this may be empty.

After this event is emitted, the request's socket will not have a `'data'` event listener, meaning you will need to bind to it in order to handle data sent to the server on that socket.

server.close([callback])#

Stops the server from accepting new connections. See `net.Server.close()`.

server.listen(handle[, callback])#

- `handle` <Object>
- `callback` <Function>

The `handle` object can be set to either a server or socket (anything with an underlying `_handle` member), or a `{fd: <n>}` object.

This will cause the server to accept connections on the specified handle, but it is presumed that the file descriptor or handle has already been bound to a port or domain socket.

Listening on a file descriptor is not supported on Windows.

This function is asynchronous. The last parameter `callback` will be added as a listener for the `'listening'` event. See also `net.Server.listen()`.

Returns `server`.

server.listen(path[, callback])#

Start a UNIX socket server listening for connections on the given `path`.

This function is asynchronous. The last parameter `callback` will be added as a listener for the `'listening'` event. See also `net.Server.listen(path)`.

server.listen(port[, hostname][, backlog][, callback])#

Begin accepting connections on the specified `port` and `hostname`. If the `hostname` is omitted, the server will accept connections on any IPv6 address (`::`) when IPv6 is available, or any IPv4 address (`0.0.0.0`) otherwise. A port value of zero will assign a random port.

To listen to a unix socket, supply a filename instead of port and hostname.

Backlog is the maximum length of the queue of pending connections. The actual length will be determined by your OS through sysctl settings such as `tcp_max_syn_backlog` and `somaxconn` on linux. The default value of this parameter is 511 (not 512).

This function is asynchronous. The last parameter `callback` will be added as a listener for the `'listening'` event. See also `net.Server.listen(port)`.

server.listening#

A Boolean indicating whether or not the server is listening for connections.

server.maxHeadersCount#

Limits maximum incoming headers count, equal to 1000 by default. If set to 0 - no limit will be applied.

server.setTimeout(msecs, callback)#

- `msecs` <Number>
- `callback` <Function>

Sets the timeout value for sockets, and emits a `'timeout'` event on the Server object, passing the socket as an argument, if a timeout occurs.

If there is a `'timeout'` event listener on the Server object, then it will be called with the timed-out socket as an argument.

By default, the Server's timeout value is 2 minutes, and sockets are destroyed automatically if they time out. However, if you assign a callback to the Server's `'timeout'` event, then you are responsible for handling socket timeouts.

Returns `server`.

server.timeout#

- <Number> Default = 120000 (2 minutes)

The number of milliseconds of inactivity before a socket is presumed to have timed out.

Note that the socket timeout logic is set up on connection, so changing this value only affects *new* connections to the server, not any existing connections.

Set to 0 to disable any kind of automatic timeout behavior on incoming connections.

Class: http.ServerResponse#

This object is created internally by a HTTP server--not by the user. It is passed as the second parameter to the `'request'` event.

The response implements the Writable Stream interface. This is an `EventEmitter` with the following events:

Event: 'close'#

```
function () { }
```

Indicates that the underlying connection was terminated before `response.end()` was called or able to flush.

Event: 'finish'#

```
function () { }
```

Emitted when the response has been sent. More specifically, this event is emitted when the last segment of the response headers and body have been handed off to the operating system for transmission over the network. It does not imply that the client has received anything yet.

After this event, no more events will be emitted on the response object.

response.addTrailers(headers)#

This method adds HTTP trailing headers (a header but at the end of the message) to the response.

Trailers will **only** be emitted if chunked encoding is used for the response; if it is not (e.g., if the request was HTTP/1.0), they will be silently discarded.

Note that HTTP requires the `Trailer` header to be sent if you intend to emit trailers, with a list of the header fields in its value. E.g.,

```
response.writeHead(200, { 'Content-Type': 'text/plain',
                          'Trailer': 'Content-MD5' });
response.write(fileData);
response.addTrailers({'Content-MD5': '7895bf4b8828b55ceaf47747b4bca667'});
response.end();
```

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

response.end([data][, encoding][, callback])#

This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete. The method, `response.end()`, MUST be called on each response.

If `data` is specified, it is equivalent to calling `response.write(data, encoding)` followed by `response.end(callback)`.

If `callback` is specified, it will be called when the response stream is finished.

response.finished#

Boolean value that indicates whether the response has completed. Starts as `false`. After `response.end()` executes, the value will be `true`.

response.getHeader(name)#

Reads out a header that's already been queued but not sent to the client. Note that the name is case insensitive. This can only be called before headers get implicitly flushed.

Example:

```
var contentType = response.getHeader('content-type');
```

response.headersSent#

Boolean (read-only). True if headers were sent, false otherwise.

response.removeHeader(name)#

Removes a header that's queued for implicit sending.

Example:

```
response.removeHeader('Content-Encoding');
```

response.sendDate#

When true, the Date header will be automatically generated and sent in the response if it is not already present in the headers. Defaults to true.

This should only be disabled for testing; HTTP requires the Date header in responses.

response.setHeader(name, value)#

Sets a single header value for implicit headers. If this header already exists in the to-be-sent headers, its value will be replaced. Use an array of strings here if you need to send multiple headers with the same name.

Example:

```
response.setHeader('Content-Type', 'text/html');
```

or

```
response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
```

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

When headers have been set with `response.setHeader()`, they will be merged with any headers passed to `response.writeHead()`, with the headers passed to `response.writeHead()` given precedence.

```
// returns content-type = text/plain
const server = http.createServer((req,res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('ok');
});
```

response.setTimeout(msecs, callback)#

- `msecs` <Number>
- `callback` <Function>

Sets the Socket's timeout value to `msecs`. If a callback is provided, then it is added as a listener on the `'timeout'` event on the response object.

If no `'timeout'` listener is added to the request, the response, or the server, then sockets are destroyed when they time out. If you assign a handler on the request, the response, or the server's `'timeout'` events, then it is your responsibility to handle timed out sockets.

Returns `response`.

response.statusCode#

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status code that will be sent to the client when the headers get flushed.

Example:

```
response.statusCode = 404;
```

After response header was sent to the client, this property indicates the status code which was sent out.

response.statusMessage#

When using implicit headers (not calling `response.writeHead()` explicitly), this property controls the status message that will be sent to the client when the headers get flushed. If this is left as `undefined` then the standard message for the status code will be used.

Example:

```
response.statusMessage = 'Not found';
```

After response header was sent to the client, this property indicates the status message which was sent out.

response.write(chunk[, encoding][, callback])#

If this method is called and `response.writeHead()` has not been called, it will switch to implicit header mode and flush the implicit headers.

This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.

`chunk` can be a string or a buffer. If `chunk` is a string, the second parameter specifies how to encode it into a byte stream. By default the `encoding` is `'utf8'`. The last parameter `callback` will be called when this chunk of data is flushed.

Note: This is the raw HTTP body and has nothing to do with higher-level multi-part body encodings that may be used.

The first time `response.write()` is called, it will send the buffered header information and the first body to the client. The second time `response.write()` is called, Node.js assumes you're going to be streaming data, and sends that separately. That is, the response is buffered up to the first chunk of body.

Returns `true` if the entire data was flushed successfully to the kernel buffer. Returns `false` if all or part of the data was queued in user memory. `'drain'` will be emitted when the buffer is free again.

response.writeContinue()#

Sends a HTTP/1.1 100 Continue message to the client, indicating that the request body should be sent. See the `'checkContinue'` event on `Server`.

response.writeHead(statusCode[, statusMessage][, headers])#

Sends a response header to the request. The status code is a 3-digit HTTP status code, like `404`. The last argument, `headers`, are the response headers. Optionally one can give a human-readable `statusMessage` as the second argument.

Example:

```
var body = 'hello world';
response.writeHead(200, {
  'Content-Length': body.length,
  'Content-Type': 'text/plain' });
```

This method must only be called once on a message and it must be called before `response.end()` is called.

If you call `response.write()` or `response.end()` before calling this, the implicit/mutable headers will be calculated and call this function for you.

When headers have been set with `response.setHeader()`, they will be merged with any headers passed to `response.writeHead()`, with the headers passed to `response.writeHead()` given precedence.

```
// returns content-type = text/plain
const server = http.createServer((req,res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('ok');
});
```

Note that Content-Length is given in bytes not characters. The above example works because the string `'hello world'` contains only single byte characters. If the body contains higher coded characters then `Buffer.byteLength()` should be used to determine the number of bytes in a given encoding. And Node.js does not check whether Content-Length and the length of the body which has been transmitted are equal or not.

Attempting to set a header field name or value that contains invalid characters will result in a `TypeError` being thrown.

Class: http.IncomingMessage#

An `IncomingMessage` object is created by `http.Server` or `http.ClientRequest` and passed as the first argument to the `'request'` and `'response'` event respectively. It may be used to access response status, headers and data.

It implements the Readable Stream interface, as well as the following additional events, methods, and properties.

Event: 'close'#

```
function () { }
```

Indicates that the underlying connection was closed. Just like `'end'`, this event occurs only once per response.

message.headers#

The request/response headers object.

Key-value pairs of header names and values. Header names are lower-cased. Example:

```
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
console.log(request.headers);
```

Duplicates in raw headers are handled in the following ways, depending on the header name:

- Duplicates of `age`, `authorization`, `content-length`, `content-type`, `etag`, `expires`, `from`, `host`, `if-modified-since`, `if-unmodified-since`, `last-modified`, `location`, `max-forwards`, `proxy-authorization`, `referer`, `retry-after`, or `user-agent` are discarded.
- `set-cookie` is always an array. Duplicates are added to the array.
- For all other headers, the values are joined together with ', '.

message.httpVersion#

In case of server request, the HTTP version sent by the client. In the case of client response, the HTTP version of the connected-to server. Probably either `'1.1'` or `'1.0'`.

Also `message.httpVersionMajor` is the first integer and `message.httpVersionMinor` is the second.

message.method#

Only valid for request obtained from `http.Server`.

The request method as a string. Read only. Example: `'GET'`, `'DELETE'`.

message.rawHeaders#

The raw request/response headers list exactly as they were received.

Note that the keys and values are in the same list. It is *not* a list of tuples. So, the even-numbered offsets are key values, and the odd-numbered offsets are the associated values.

Header names are not lowercased, and duplicates are not merged.

```
// Prints something like:
```

```
//
// [ 'user-agent',
//   'this is invalid because there can be only one',
//   'User-Agent',
//   'curl/7.22.0',
//   'Host',
//   '127.0.0.1:8000',
//   'ACCEPT',
//   '*/*' ]
console.log(request.rawHeaders);
```

message.rawTrailers#

The raw request/response trailer keys and values exactly as they were received. Only populated at the `'end'` event.

message.setTimeout(msecs, callback)#

- `msecs` <Number>
- `callback` <Function>

Calls `message.connection.setTimeout(msecs, callback)`.

Returns `message`.

message.statusCode#

Only valid for response obtained from `http.ClientRequest`.

The 3-digit HTTP response status code. E.G. `404`.

message.statusMessage#

Only valid for response obtained from `http.ClientRequest`.

The HTTP response status message (reason phrase). E.G. `OK` or `Internal Server Error`.

message.socket#

The `net.Socket` object associated with the connection.

With HTTPS support, use `request.socket.getPeerCertificate()` to obtain the client's authentication details.

message.trailers#

The request/response trailers object. Only populated at the `'end'` event.

message.url[#]

Only valid for request obtained from [`http.Server`].

Request URL string. This contains only the URL that is present in the actual HTTP request. If the request is:

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

Then `request.url` will be:

```
'/status?name=ryan'
```

If you would like to parse the URL into its parts, you can use `require('url').parse(request.url)`. Example:

```
$ node
> require('url').parse('/status?name=ryan')
{
  href: '/status?name=ryan',
  search: '?name=ryan',
  query: 'name=ryan',
  pathname: '/status'
}
```

If you would like to extract the params from the query string, you can use the `require('querystring').parse` function, or pass `true` as the second argument to `require('url').parse`. Example:

```
$ node
> require('url').parse('/status?name=ryan', true)
{
  href: '/status?name=ryan',
  search: '?name=ryan',
  query: {name: 'ryan'},
  pathname: '/status'
}
```

http.METHODS[#]

- <Array>

A list of the HTTP methods that are supported by the parser.

http.STATUS_CODES#

- <Object>

A collection of all the standard HTTP response status codes, and the short description of each. For example, `http.STATUS_CODES[404] === 'Not Found'`.

http.createClient([port][, host])#

```
Stability: 0 - Deprecated: Use http.request() instead.
```

Constructs a new HTTP client. `port` and `host` refer to the server to be connected to.

http.createServer([requestListener])#

Returns a new instance of `http.Server`.

The `requestListener` is a function which is automatically added to the `'request'` event.

http.get(options[, callback])#

Since most requests are GET requests without bodies, Node.js provides this convenience method. The only difference between this method and `http.request()` is that it sets the method to GET and calls `req.end()` automatically.

Example:

```
http.get('http://www.google.com/index.html', (res) => {
  console.log(`Got response: ${res.statusCode}`);
  // consume response body
  res.resume();
}).on('error', (e) => {
  console.log(`Got error: ${e.message}`);
});
```

http.globalAgent#

Global instance of Agent which is used as the default for all http client requests.

http.request(options[, callback])#

Node.js maintains several connections per server to make HTTP requests. This function allows one to transparently issue requests.

`options` can be an object or a string. If `options` is a string, it is automatically parsed with [`url.parse()`](url.parse()).

Options:

- `protocol`: Protocol to use. Defaults to `'http:'`.
- `host`: A domain name or IP address of the server to issue the request to. Defaults to `'localhost'`.
- `hostname`: Alias for `host`. To support `url.parse()` `hostname` is preferred over `host`.
- `family`: IP address family to use when resolving `host` and `hostname`. Valid values are `4` or `6`. When unspecified, both IP v4 and v6 will be used.
- `port`: Port of remote server. Defaults to 80.
- `localAddress`: Local interface to bind for network connections.
- `socketPath`: Unix Domain Socket (use one of host:port or socketPath).
- `method`: A string specifying the HTTP request method. Defaults to `'GET'`.
- `path`: Request path. Defaults to `'/'`. Should include query string if any. E.G. `'/index.html?page=12'`. An exception is thrown when the request path contains illegal characters. Currently, only spaces are rejected but that may change in the future.
- `headers`: An object containing request headers.
- `auth`: Basic authentication i.e. `'user:password'` to compute an Authorization header.
- `agent`: Controls `Agent` behavior. When an Agent is used request will default to `Connection: keep-alive`. Possible values:
  - `undefined` (default): use `http.globalAgent` for this host and port.
  - `Agent` object: explicitly use the passed in `Agent`.
  - `false`: opts out of connection pooling with an Agent, defaults request to `Connection: close`.
- `createConnection`: A function that produces a socket/stream to use for the request when the `agent` option is not used. This can be used to avoid creating a custom Agent class just to override the default `createConnection` function. See `agent.createConnection()` for more details.

The optional `callback` parameter will be added as a one time listener for the `'response'` event.

`http.request()` returns an instance of the `http.ClientRequest` class. The `ClientRequest` instance is a writable stream. If one needs to upload a file with a POST request, then write to the `ClientRequest` object.

Example:

```
var postData = querystring.stringify({
  'msg' : 'Hello World!'
});
```

```
var options = {
  hostname: 'www.google.com',
  port: 80,
  path: '/upload',
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': postData.length
  }
};

var req = http.request(options, (res) => {
  console.log(`STATUS: ${res.statusCode}`);
  console.log(`HEADERS: ${JSON.stringify(res.headers)}`);
  res.setEncoding('utf8');
  res.on('data', (chunk) => {
    console.log(`BODY: ${chunk}`);
  });
  res.on('end', () => {
    console.log('No more data in response.')
  })
});

req.on('error', (e) => {
  console.log(`problem with request: ${e.message}`);
});

// write data to request body
req.write(postData);
req.end();
```

Note that in the example `req.end()` was called. With `http.request()` one must always call `req.end()` to signify that you're done with the request - even if there is no data being written to the request body.

If any error is encountered during the request (be that with DNS resolution, TCP level errors, or actual HTTP parse errors) an `'error'` event is emitted on the returned request object. As with all `'error'` events, if no listeners are registered the error will be thrown.

There are a few special headers that should be noted.

- Sending a 'Connection: keep-alive' will notify Node.js that the connection to the server should be persisted until the next request.

- Sending a 'Content-length' header will disable the default chunked encoding.

- Sending an 'Expect' header will immediately send the request headers. Usually, when sending 'Expect: 100-continue', you should both set a timeout and listen for the `'continue'` event. See RFC2616 Section 8.2.3 for more information.
- Sending an Authorization header will override using the `auth` option to compute basic authentication.