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, so 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': 'example.com',
  'accept': '*/*' }
```

Keys are lowercased. Values are not modified.

In order to support the full spectrum of possible HTTP applications, the Node.js 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', 'example.com',
  'accepT', '*/*' ]
```

Class: `http.Agent`

An `Agent` is responsible for managing connection persistence and reuse for HTTP clients. It maintains a queue of pending requests for a given host and port, reusing a single socket connection for each until the queue is empty, at which time the socket is either destroyed or put into a pool where it is kept to be used again for requests to the same host and port. Whether it is destroyed or pooled depends on the `keepAlive` option.

Pooled connections have TCP Keep-Alive enabled for them, but servers may still close idle connections, in which case they will be removed from the pool and a new connection will be made when a new HTTP request is made for that host and port. Servers may also refuse to allow multiple requests over the same connection, in which case the connection will have to be remade for every request and cannot be pooled. The `Agent` will still make the requests to that server, but each one will occur over a new connection.

When a connection is closed by the client or the server, it is removed from the pool. Any unused sockets in the pool will be unrefed so as not to keep the Node.js process running when there are no outstanding requests. (see `socket.unref()`).

It is good practice, to `destroy()` an `Agent` instance when it is no longer in use, because unused sockets consume OS resources.

Sockets are removed from an agent when the socket emits either a `'close'` event or an `'agentRemove'` event. When intending to keep one HTTP request open for a long time without keeping it in the agent, something like the following may be done:

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

An agent may also be used for an individual request. By providing `{agent: false}` as an option to the `http.get()` or `http.request()` functions, a one-time use `Agent` with default options will be used for the client connection.

`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 even when there are no outstanding requests, so they can be used for future requests without having to reestablish a TCP connection. Not to be confused with the `keep-alive` value of the `Connection` header. The `Connection: keep-alive` header is always sent when using an agent except when the `Connection` header is explicitly specified or when the `keepAlive` and `maxSockets` options are respectively set to `false` and `Infinity` , in which case `Connection: close` will be used. **Default:** `false` .
  - `keepAliveMsecs` {number} When using the `keepAlive` option, specifies the initial delay for TCP Keep-Alive packets. Ignored when the `keepAlive` option is `false` or `undefined` . **Default:** `1000` .
  - `maxSockets` {number} Maximum number of sockets to allow per host. If the same host opens multiple concurrent connections, each request will use new socket until the `maxSockets` value is reached. If the host attempts to open more connections than `maxSockets` , the additional requests will enter into a pending request queue, and will enter active connection state when an

existing connection terminates. This makes sure there are at most `maxSockets` active connections at any point in time, from a given host. **Default:** `Infinity`.
  - `maxTotalSockets` {number} Maximum number of sockets allowed for all hosts in total. Each request will use a new socket until the maximum is reached. **Default:** `Infinity`.
  - `maxFreeSockets` {number} Maximum number of sockets per host to leave open in a free state. Only relevant if `keepAlive` is set to `true`. **Default:** `256`.
  - `scheduling` {string} Scheduling strategy to apply when picking the next free socket to use. It can be `'fifo'` or `'lifo'`. The main difference between the two scheduling strategies is that `'lifo'` selects the most recently used socket, while `'fifo'` selects the least recently used socket. In case of a low rate of request per second, the `'lifo'` scheduling will lower the risk of picking a socket that might have been closed by the server due to inactivity. In case of a high rate of request per second, the `'fifo'` scheduling will maximize the number of open sockets, while the `'lifo'` scheduling will keep it as low as possible. **Default:** `'lifo'`.
  - `timeout` {number} Socket timeout in milliseconds. This will set the timeout when the socket is created.

`options` in [`socket.connect()`](#) are also supported.

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, a custom [`http.Agent`](#) instance must be created.

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

`agent.createConnection(options[, callback])`

- `options` {Object} Options containing connection details. Check [`net.createConnection()`](#) for the format of the options
- `callback` {Function} Callback function that receives the created socket
- Returns: {stream.Duplex}

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`.

This method is guaranteed to return an instance of the {net.Socket} class, a subclass of {stream.Duplex}, unless the user specifies a socket type other than {net.Socket}.

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

`agent.keepSocketAlive(socket)`

- `socket` {stream.Duplex}

Called when `socket` is detached from a request and could be persisted by the `Agent`. Default behavior is to:

```
socket.setKeepAlive(true, this.keepAliveMsecs);
socket.unref();
return true;
```

This method can be overridden by a particular `Agent` subclass. If this method returns a falsy value, the socket will be destroyed instead of persisting it for use with the next request.

The `socket` argument can be an instance of {net.Socket}, a subclass of {stream.Duplex}.

`agent.reuseSocket(socket, request)`

- `socket` {stream.Duplex}
- `request` {http.ClientRequest}

Called when `socket` is attached to `request` after being persisted because of the keep-alive options. Default behavior is to:

```
socket.ref();
```

This method can be overridden by a particular `Agent` subclass.

The `socket` argument can be an instance of {net.Socket}, a subclass of {stream.Duplex}.

`agent.destroy()`

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

It is usually not necessary to do this. However, if using an agent with `keepAlive` enabled, then it is best to explicitly shut down the agent when it is no longer needed. Otherwise, sockets might stay open for quite a long time before the server terminates them.

`agent.freeSockets`

- {Object}

An object which contains arrays of sockets currently awaiting use by the agent when `keepAlive` is enabled. Do not modify.

Sockets in the `freeSockets` list will be automatically destroyed and removed from the array on `'timeout'`.

`agent.getName([options])`

- `options` {Object} A set of options providing information for name generation
  - `host` {string} A domain name or IP address of the server to issue the request to
  - `port` {number} Port of remote server
  - `localAddress` {string} Local interface to bind for network connections when issuing the request
  - `family` {integer} Must be 4 or 6 if this doesn't equal `undefined`.
- Returns: {string}

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

`agent.maxFreeSockets`

- {number}

By default set to 256. For agents with `keepAlive` enabled, this sets the maximum number of sockets that will be left open in the free state.

`agent.maxSockets`

- {number}

By default set to `Infinity`. Determines how many concurrent sockets the agent can have open per origin. Origin is the returned value of `agent.getName()`.

`agent.maxTotalSockets`

- {number}

By default set to `Infinity`. Determines how many concurrent sockets the agent can have open. Unlike `maxSockets`, this parameter applies across all origins.

`agent.requests`

- {Object}

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

`agent.sockets`

- {Object}

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

Class: `http.ClientRequest`

- Extends: {http.OutgoingMessage}

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 calling `request.end()`.

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 a `'response'` event handler is added, then the data from the response object **must** be consumed, 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.

For backward compatibility, `res` will only emit `'error'` if there is an `'error'` listener registered.

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

Event: `'abort'`

> *Stability: 0 - Deprecated. Listen for the `'close'` event instead.*

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

Event: `'connect'`

- `response` {http.IncomingMessage}
- `socket` {stream.Duplex}
- `head` {Buffer}

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

This event is guaranteed to be passed an instance of the {net.Socket} class, a subclass of {stream.Duplex}, unless the user specifies a socket type other than {net.Socket}.

A client and server pair demonstrating how to listen for the `'connect'` event:

```
const http = require('http');
const net = require('net');
const { URL } = require('url');

// Create an HTTP tunneling proxy
const proxy = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('okay');
});
proxy.on('connect', (req, clientSocket, head) => {
  // Connect to an origin server
  const { port, hostname } = new URL(`http://${req.url}`);
  const serverSocket = net.connect(port || 80, hostname, () => {
    clientSocket.write('HTTP/1.1 200 Connection Established\r\n' +
                    'Proxy-agent: Node.js-Proxy\r\n' +
                    '\r\n');
    serverSocket.write(head);
    serverSocket.pipe(clientSocket);
    clientSocket.pipe(serverSocket);
  });
});

// Now that proxy is running
proxy.listen(1337, '127.0.0.1', () => {
  // Make a request to a tunneling proxy
  const options = {
    port: 1337,
    host: '127.0.0.1',
    method: 'CONNECT',
    path: 'www.google.com:80'
  };

  const 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'`

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: `'information'`

- `info` {Object}
  - `httpVersion` {string}
  - `httpVersionMajor` {integer}
  - `httpVersionMinor` {integer}
  - `statusCode` {integer}
  - `statusMessage` {string}
  - `headers` {Object}
  - `rawHeaders` {string[]}

Emitted when the server sends a 1xx intermediate response (excluding 101 Upgrade). The listeners of this event will receive an object containing the HTTP version, status code, status message, key-value headers object, and array with the raw header names followed by their respective values.

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

const options = {
  host: '127.0.0.1',
  port: 8080,
  path: '/length_request'
};

// Make a request
const req = http.request(options);
req.end();

req.on('information', (info) => {
  console.log(`Got information prior to main response: ${info.statusCode}`);
});
```

101 Upgrade statuses do not fire this event due to their break from the traditional HTTP request/response chain, such as web sockets, in-place TLS upgrades, or HTTP 2.0. To be notified of 101 Upgrade notices, listen for the `'upgrade'` event instead.

Event: `'response'`

- `response` {http.IncomingMessage}

Emitted when a response is received to this request. This event is emitted only once.

Event: `'socket'`

- `socket` {stream.Duplex}

This event is guaranteed to be passed an instance of the {net.Socket} class, a subclass of {stream.Duplex}, unless the user specifies a socket type other than {net.Socket}.

Event: `'timeout'`

Emitted when the underlying socket times out from inactivity. This only notifies that the socket has been idle. The request must be destroyed manually.

See also: `request.setTimeout()`.

Event: `'upgrade'`

- `response` {http.IncomingMessage}
- `socket` {stream.Duplex}
- `head` {Buffer}

Emitted each time a server responds to a request with an upgrade. If this event is not being listened for and the response status code is 101 Switching Protocols, clients receiving an upgrade header will have their connections closed.

This event is guaranteed to be passed an instance of the {net.Socket} class, a subclass of {stream.Duplex}, unless the user specifies a socket type other than {net.Socket}.

A client server pair demonstrating how to listen for the `'upgrade'` event.

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

// Create an HTTP server
```

```
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('okay');
});
server.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
server.listen(1337, '127.0.0.1', () => {

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

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

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

`request.abort()`

> *Stability: 0 - Deprecated: Use [`request.destroy()`] instead.*

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

`request.aborted`

> *Stability: 0 - Deprecated. Check [`request.destroyed`] instead.*

- {boolean}

The `request.aborted` property will be `true` if the request has been aborted.

`request.connection`

> *Stability: 0 - Deprecated. Use `request.socket`.*

- {stream.Duplex}

See `request.socket`.

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

- `data` {string|Buffer}
- `encoding` {string}
- `callback` {Function}
- Returns: {this}

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 `request.write(data, encoding)` followed by `request.end(callback)`.

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

`request.destroy([error])`

- `error` {Error} Optional, an error to emit with `'error'` event.
- Returns: {this}

Destroy the request. Optionally emit an `'error'` event, and emit a `'close'` event. Calling this will cause remaining data in the response to be dropped and the socket to be destroyed.

See `writable.destroy()` for further details.

`request.destroyed`

- {boolean}

Is `true` after `request.destroy()` has been called.

See `writable.destroyed` for further details.

`request.finished`

> *Stability: 0 - Deprecated. Use `request.writableEnded`.*

- {boolean}

The `request.finished` property will be `true` if `request.end()` has been called. `request.end()` will automatically be called if the request was initiated via `http.get()`.

`request.flushHeaders()`

Flushes the request headers.

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

That's usually desired (it saves a TCP round-trip), but not when the first data is not sent until possibly much later. `request.flushHeaders()` bypasses the optimization and kickstarts the request.

`request.getHeader(name)`

- `name` {string}
- Returns: {any}

Reads out a header on the request. The name is case-insensitive. The type of the return value depends on the arguments provided to `request.setHeader()`.

```
request.setHeader('content-type', 'text/html');
request.setHeader('Content-Length', Buffer.byteLength(body));
request.setHeader('Cookie', ['type=ninja', 'language=javascript']);
const contentType = request.getHeader('Content-Type');
// 'contentType' is 'text/html'
const contentLength = request.getHeader('Content-Length');
// 'contentLength' is of type number
const cookie = request.getHeader('Cookie');
// 'cookie' is of type string[]
```

`request.getRawHeaderNames()`

- Returns: {string[]}

Returns an array containing the unique names of the current outgoing raw headers. Header names are returned with their exact casing being set.

```
request.setHeader('Foo', 'bar');
request.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headerNames = request.getRawHeaderNames();
// headerNames === ['Foo', 'Set-Cookie']
```

`request.maxHeadersCount`

- {number} **Default:** `2000`

Limits maximum response headers count. If set to 0, no limit will be applied.

`request.path`

- {string} The request path.

`request.method`

- {string} The request method.

`request.host`

- {string} The request host.

`request.protocol`

- {string} The request protocol.

`request.removeHeader(name)`

- `name` {string}

Removes a header that's already defined into headers object.

```
request.removeHeader('Content-Type');
```

`request.reusedSocket`

- {boolean} Whether the request is send through a reused socket.

When sending request through a keep-alive enabled agent, the underlying socket might be reused. But if server closes connection at unfortunate time, client may run into a 'ECONNRESET' error.

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

// Server has a 5 seconds keep-alive timeout by default
http
  .createServer((req, res) => {
    res.write('hello\n');
    res.end();
  })
  .listen(3000);

setInterval(() => {
  // Adapting a keep-alive agent
  http.get('http://localhost:3000', { agent }, (res) => {
    res.on('data', (data) => {
      // Do nothing
    });
  });
}, 5000); // Sending request on 5s interval so it's easy to hit idle timeout
```

By marking a request whether it reused socket or not, we can do automatic error retry base on it.

```
const http = require('http');
const agent = new http.Agent({ keepAlive: true });

function retriableRequest() {
  const req = http
    .get('http://localhost:3000', { agent }, (res) => {
      // ...
    })
    .on('error', (err) => {
      // Check if retry is needed
      if (req.reusedSocket && err.code === 'ECONNRESET') {
        retriableRequest();
      }
    });
}

retriableRequest();
```

`request.setHeader(name, value)`

- `name` {string}
- `value` {any}

Sets a single header value for headers object. If this header already exists in the to-be-sent headers, its value will be replaced. Use an array of strings here to send multiple headers with the same name. Non-string values will be stored without modification. Therefore, `request.getHeader()` may return non-string values. However, the non-string values will be converted to strings for network transmission.

```
request.setHeader('Content-Type', 'application/json');
```

or

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

`request.setNoDelay([noDelay])`

- `noDelay` {boolean}

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

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

- `enable` {boolean}
- `initialDelay` {number}

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

`request.setTimeout(timeout[, callback])`

- `timeout` {number} Milliseconds before a request times out.
- `callback` {Function} Optional function to be called when a timeout occurs. Same as binding to the `'timeout'` event.
- Returns: {http.ClientRequest}

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

`request.socket`

- {stream.Duplex}

Reference to the underlying socket. Usually users will not want to access this property. In particular, the socket will not emit `'readable'` events because of how the protocol parser attaches to the socket.

```
const http = require('http');
const options = {
  host: 'www.google.com',
};
const req = http.get(options);
req.end();
req.once('response', (res) => {
  const ip = req.socket.localAddress;
  const port = req.socket.localPort;
  console.log(`Your IP address is ${ip} and your source port is ${port}.`);
  // Consume response object
});
```

This property is guaranteed to be an instance of the {net.Socket} class, a subclass of {stream.Duplex}, unless the user specified a socket type other than {net.Socket}.

`request.writableEnded`

- {boolean}

Is `true` after [`request.end()`]() has been called. This property does not indicate whether the data has been flushed, for this use [`request.writableFinished`]() instead.

`request.writableFinished`

- {boolean}

Is `true` if all data has been flushed to the underlying system, immediately before the [`'finish'`]() event is emitted.

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

- `chunk` {string|Buffer}
- `encoding` {string}
- `callback` {Function}
- Returns: {boolean}

Sends a chunk of the body. This method can be called multiple times. If no `Content-Length` is set, data will automatically be encoded in HTTP Chunked transfer encoding, so that server knows when the data ends. The `Transfer-Encoding: chunked` header is added. Calling [`request.end()`]() is necessary to finish sending the request.

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, but only if the chunk is non-empty.

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.

When `write` function is called with empty string or buffer, it does nothing and waits for more input.

Class: `http.Server`

- Extends: {net.Server}

Event: `'checkContinue'`

- `request` {http.IncomingMessage}
- `response` {http.ServerResponse}

Emitted each time a request with an HTTP `Expect: 100-continue` is received. If this event is not 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.

When this event is emitted and handled, the `'request'` event will not be emitted.

Event: `'checkExpectation'`

- `request` {http.IncomingMessage}
- `response` {http.ServerResponse}

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

When this event is emitted and handled, the `'request'` event will not be emitted.

Event: `'clientError'`

- `exception` {Error}
- `socket` {stream.Duplex}

If a client connection emits an `'error'` event, it will be forwarded here. Listener of this event is responsible for closing/destroying the underlying socket. For example, one may wish to more gracefully close the socket with a custom HTTP response instead of abruptly severing the connection.

This event is guaranteed to be passed an instance of the {net.Socket} class, a subclass of {stream.Duplex}, unless the user specifies a socket type other than {net.Socket}.

Default behavior is to try close the socket with a HTTP '400 Bad Request', or a HTTP '431 Request Header Fields Too Large' in the case of a `HPE_HEADER_OVERFLOW` error. If the socket is not writable or has already written data it is immediately destroyed.

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

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

const server = http.createServer((req, res) => {
  res.end();
});
server.on('clientError', (err, socket) => {
  socket.end('HTTP/1.1 400 Bad Request\r\n\r\n');
});
server.listen(8000);
```

When the `'clientError'` event occurs, there is no `request` or `response` object, so any HTTP response sent, including response headers and payload, *must* be written directly to the `socket` object. Care must be taken to ensure the response is a properly formatted HTTP response message.

`err` is an instance of `Error` with two extra columns:

- `bytesParsed` : the bytes count of request packet that Node.js may have parsed correctly;
- `rawPacket` : the raw packet of current request.

In some cases, the client has already received the response and/or the socket has already been destroyed, like in case of `ECONNRESET` errors. Before trying to send data to the socket, it is better to check that it is still writable.

```
server.on('clientError', (err, socket) => {
  if (err.code === 'ECONNRESET' || !socket.writable) {
    return;
  }

  socket.end('HTTP/1.1 400 Bad Request\r\n\r\n');
});
```

Event: `'close'`

Emitted when the server closes.

Event: `'connect'`

- `request` {http.IncomingMessage} Arguments for the HTTP request, as it is in the `'request'` event
- `socket` {stream.Duplex} Network socket between the server and client
- `head` {Buffer} The first packet of the tunneling stream (may be empty)

Emitted each time a client requests an HTTP `CONNECT` method. If this event is not listened for, then clients requesting a `CONNECT` method will have their connections closed.

This event is guaranteed to be passed an instance of the {net.Socket} class, a subclass of {stream.Duplex}, unless the user specifies a socket type other than {net.Socket}.

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

Event: `'connection'`

- `socket` {stream.Duplex}

This event is emitted when a new TCP stream is established. `socket` is typically 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.socket`.

This event can also be explicitly emitted by users to inject connections into the HTTP server. In that case, any `Duplex` stream can be passed.

If `socket.setTimeout()` is called here, the timeout will be replaced with `server.keepAliveTimeout` when the socket has served a request (if `server.keepAliveTimeout` is non-zero).

This event is guaranteed to be passed an instance of the {net.Socket} class, a subclass of {stream.Duplex}, unless the user specifies a socket type other than {net.Socket}.

Event: `'request'`

- `request` {http.IncomingMessage}
- `response` {http.ServerResponse}

Emitted each time there is a request. There may be multiple requests per connection (in the case of HTTP Keep-Alive connections).

Event: `'upgrade'`

- `request` {http.IncomingMessage} Arguments for the HTTP request, as it is in the [`'request'`]() event
- `socket` {stream.Duplex} Network socket between the server and client
- `head` {Buffer} The first packet of the upgraded stream (may be empty)

Emitted each time a client requests an HTTP upgrade. Listening to this event is optional and clients cannot insist on a protocol change.

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

This event is guaranteed to be passed an instance of the {net.Socket} class, a subclass of {stream.Duplex}, unless the user specifies a socket type other than {net.Socket}.

`server.close([callback])`

- `callback` {Function}

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

`server.headersTimeout`

- {number} **Default:** `60000`

Limit the amount of time the parser will wait to receive the complete HTTP headers.

In case of inactivity, the rules defined in [`server.timeout`]() apply. However, that inactivity based timeout would still allow the connection to be kept open if the headers are being sent very slowly (by default, up to a byte per 2 minutes). In order to prevent this, whenever header data arrives an additional check is made that more than `server.headersTimeout` milliseconds has not passed since the connection was established. If the check fails, a `'timeout'` event is emitted on the server object, and (by default) the socket is destroyed. See [`server.timeout`]() for more information on how timeout behavior can be customized.

`server.listen()`

Starts the HTTP server listening for connections. This method is identical to [`server.listen()`]() from [`net.Server`]().

`server.listening`

- {boolean} Indicates whether or not the server is listening for connections.

`server.maxHeadersCount`

- {number} **Default:** `2000`

Limits maximum incoming headers count. If set to 0, no limit will be applied.

`server.requestTimeout`

- {number} **Default:** `0`

Sets the timeout value in milliseconds for receiving the entire request from the client.

If the timeout expires, the server responds with status 408 without forwarding the request to the request listener and then closes the connection.

It must be set to a non-zero value (e.g. 120 seconds) to protect against potential Denial-of-Service attacks in case the server is deployed without a reverse proxy in front.

`server.setTimeout([msecs][, callback])`

- `msecs` {number} **Default:** 0 (no timeout)
- `callback` {Function}
- Returns: {http.Server}

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 does not timeout sockets. However, if a callback is assigned to the Server's `'timeout'` event, timeouts must be handled explicitly.

`server.maxRequestsPerSocket`

- {number} Requests per socket. **Default:** 0 (no limit)

The maximum number of requests socket can handle before closing keep alive connection.

A value of `0` will disable the limit.

When the limit is reached it will set the `Connection` header value to `close`, but will not actually close the connection, subsequent requests sent after the limit is reached will get `503 Service Unavailable` as a response.

`server.timeout`

- {number} Timeout in milliseconds. **Default:** 0 (no timeout)

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

A value of `0` will disable the timeout behavior on incoming connections.

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

`server.keepAliveTimeout`

- {number} Timeout in milliseconds. **Default:** `5000` (5 seconds).

The number of milliseconds of inactivity a server needs to wait for additional incoming data, after it has finished writing the last response, before a socket will be destroyed. If the server receives new data before the keep-alive timeout has fired, it will reset the regular inactivity timeout, i.e., `server.timeout`.

A value of `0` will disable the keep-alive timeout behavior on incoming connections. A value of `0` makes the http server behave similarly to Node.js versions prior to 8.0.0, which did not have a keep-alive timeout.

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

Class: `http.ServerResponse`

- Extends: {Stream}

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

Event: `'close'`

Indicates that the response is completed, or its underlying connection was terminated prematurely (before the response completion).

Event: `'finish'`

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.

`response.addTrailers(headers)`

- `headers` {Object}

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.

HTTP requires the `Trailer` header to be sent in order 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.connection`

> *Stability: 0 - Deprecated. Use* `response.socket`.

- {stream.Duplex}

See `response.socket`.

`response.cork()`

See `writable.cork()`.

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

- `data` {string|Buffer}
- `encoding` {string}
- `callback` {Function}
- Returns: {this}

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 similar in effect 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`

> *Stability: 0 - Deprecated. Use [`response.writableEnded`]().*

* {boolean}

The `response.finished` property will be `true` if [`response.end()`]() has been called.

`response.flushHeaders()`

Flushes the response headers. See also: [`request.flushHeaders()`]().

`response.getHeader(name)`

* `name` {string}
* Returns: {any}

Reads out a header that's already been queued but not sent to the client. The name is case-insensitive. The type of the return value depends on the arguments provided to [`response.setHeader()`]().

```
response.setHeader('Content-Type', 'text/html');
response.setHeader('Content-Length', Buffer.byteLength(body));
response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);
const contentType = response.getHeader('content-type');
// contentType is 'text/html'
const contentLength = response.getHeader('Content-Length');
// contentLength is of type number
const setCookie = response.getHeader('set-cookie');
// setCookie is of type string[]
```

`response.getHeaderNames()`

* Returns: {string[]}

Returns an array containing the unique names of the current outgoing headers. All header names are lowercase.

```
response.setHeader('Foo', 'bar');
response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headerNames = response.getHeaderNames();
// headerNames === ['foo', 'set-cookie']
```

`response.getHeaders()`

* Returns: {Object}

Returns a shallow copy of the current outgoing headers. Since a shallow copy is used, array values may be mutated without additional calls to various header-related http module methods. The keys of the returned object are the header names and the values are the respective header values. All header names are lowercase.

The object returned by the `response.getHeaders()` method *does not* prototypically inherit from the JavaScript `Object`. This means that typical `Object` methods such as `obj.toString()`, `obj.hasOwnProperty()`, and others are not defined and *will not work*.

```
response.setHeader('Foo', 'bar');
response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headers = response.getHeaders();
// headers === { foo: 'bar', 'set-cookie': ['foo=bar', 'bar=baz'] }
```

`response.hasHeader(name)`

- `name` {string}
- Returns: {boolean}

Returns `true` if the header identified by `name` is currently set in the outgoing headers. The header name matching is case-insensitive.

```
const hasContentType = response.hasHeader('content-type');
```

`response.headersSent`

- {boolean}

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

`response.removeHeader(name)`

- `name` {string}

Removes a header that's queued for implicit sending.

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

`response.req`

- {http.IncomingMessage}

A reference to the original HTTP `request` object.

`response.sendDate`

- {boolean}

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)`

- `name` {string}
- `value` {any}
- Returns: {http.ServerResponse}

Returns the response object.

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 to send multiple headers with the same name. Non-string values will be stored without modification. Therefore, `response.getHeader()` may return non-string values. However, the non-string values will be converted to strings for network transmission. The same response object is returned to the caller, to enable call chaining.

```
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');
});
```

If `response.writeHead()` method is called and this method has not been called, it will directly write the supplied header values onto the network channel without caching internally, and the `response.getHeader()` on the header will not yield the expected result. If progressive population of headers is desired with potential future retrieval and modification, use `response.setHeader()` instead of `response.writeHead()`.

`response.setTimeout(msecs[, callback])`

- `msecs` {number}
- `callback` {Function}
- Returns: {http.ServerResponse}

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 a handler is assigned to the request, the response, or the server's `'timeout'` events, timed out sockets must be handled explicitly.

`response.socket`

- {stream.Duplex}

Reference to the underlying socket. Usually users will not want to access this property. In particular, the socket will not emit `'readable'` events because of how the protocol parser attaches to the socket. After `response.end()`, the property is nulled.

```
const http = require('http');
const server = http.createServer((req, res) => {
  const ip = res.socket.remoteAddress;
  const port = res.socket.remotePort;
  res.end(`Your IP address is ${ip} and your source port is ${port}.`);
}).listen(3000);
```

This property is guaranteed to be an instance of the {net.Socket} class, a subclass of {stream.Duplex}, unless the user specified a socket type other than {net.Socket}.

`response.statusCode`

- {number} **Default:** `200`

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.

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

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

`response.statusMessage`

- {string}

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.

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

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

`response.uncork()`

See `writable.uncork()`.

`response.writableEnded`

- {boolean}

Is `true` after `response.end()` has been called. This property does not indicate whether the data has been flushed, for this use `response.writableFinished` instead.

`response.writableFinished`

- {boolean}

Is `true` if all data has been flushed to the underlying system, immediately before the `'finish'` event is emitted.

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

- `chunk` {string|Buffer}
- `encoding` {string} **Default:** `'utf8'`
- `callback` {Function}
- Returns: {boolean}

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.

In the `http` module, the response body is omitted when the request is a HEAD request. Similarly, the `204` and `304` responses *must not* include a message 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. `callback` will be called when this chunk of data is flushed.

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 chunk of the body to the client. The second time `response.write()` is called, Node.js assumes data will be streamed, and sends the new data separately. That is, the response is buffered up to the first chunk of the 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])`

- `statusCode` {number}
- `statusMessage` {string}
- `headers` {Object|Array}
- Returns: {http.ServerResponse}

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.

`headers` may be an `Array` where 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. The array is in the same format as `request.rawHeaders`.

Returns a reference to the `ServerResponse`, so that calls can be chained.

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

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

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

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.

If this method is called and `response.setHeader()` has not been called, it will directly write the supplied header values onto the network channel without caching internally, and the `response.getHeader()` on the header will not yield the expected result. If progressive population of headers is desired with potential future retrieval and modification, use `response.setHeader()` instead.

```
// 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');
});
```

`Content-Length` is given in bytes, not characters. Use `Buffer.byteLength()` to determine the length of the body in bytes. 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.

`response.writeProcessing()`

Sends a HTTP/1.1 102 Processing message to the client, indicating that the request body should be sent.

Class: `http.IncomingMessage`

- Extends: {stream.Readable}

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.

Different from its `socket` value which is a subclass of {stream.Duplex}, the `IncomingMessage` itself extends {stream.Readable} and is created separately to parse and emit the incoming HTTP headers and payload, as the underlying socket may be reused multiple times in case of keep-alive.

Event: `'aborted'`

> *Stability: 0 - Deprecated. Listen for `'close'` event instead.*

Emitted when the request has been aborted.

Event: `'close'`

Emitted when the request has been completed.

`message.aborted`

> *Stability: 0 - Deprecated. Check `message.destroyed` from {stream.Readable}.*

- {boolean}

The `message.aborted` property will be `true` if the request has been aborted.

`message.complete`

- {boolean}

The `message.complete` property will be `true` if a complete HTTP message has been received and successfully parsed.

This property is particularly useful as a means of determining if a client or server fully transmitted a message before a connection was terminated:

```
const req = http.request({
  host: '127.0.0.1',
  port: 8080,
  method: 'POST'
}, (res) => {
  res.resume();
  res.on('end', () => {
    if (!res.complete)
      console.error(
        'The connection was terminated while the message was still being sent');
  });
});
```

`message.connection`

> *Stability: 0 - Deprecated. Use [`message.socket`].*

Alias for [`message.socket`].

`message.destroy([error])`

- `error` {Error}
- Returns: {this}

Calls `destroy()` on the socket that received the `IncomingMessage`. If `error` is provided, an `'error'` event is emitted on the socket and `error` is passed as an argument to any listeners on the event.

`message.headers`

- {Object}

The request/response headers object.

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

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

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`, `server`, or `user-agent` are discarded.
- `set-cookie` is always an array. Duplicates are added to the array.
- For duplicate `cookie` headers, the values are joined together with '; '.
- For all other headers, the values are joined together with ', '.

`message.httpVersion`

- {string}

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`

- {string}

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

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

`message.rawHeaders`

- {string[]}

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

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`

- {string[]}

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}
- Returns: {http.IncomingMessage}

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

`message.socket`

- {stream.Duplex}

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

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

This property is guaranteed to be an instance of the {net.Socket} class, a subclass of {stream.Duplex}, unless the user specified a socket type other than {net.Socket} or internally nulled.

`message.statusCode`

- {number}

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

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

`message.statusMessage`

- {string}

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

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

`message.trailers`

- {Object}

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

`message.url`

- {string}

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

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

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

To parse the URL into its parts:

```
new URL(request.url, `http://${request.getHeaders().host}`);
```

When `request.url` is `'/status?name=ryan'` and `request.getHeaders().host` is `'localhost:3000'`:

```
$ node
> new URL(request.url, `http://${request.getHeaders().host}`)
URL {
  href: 'http://localhost:3000/status?name=ryan',
  origin: 'http://localhost:3000',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'localhost:3000',
  hostname: 'localhost',
  port: '3000',
  pathname: '/status',
  search: '?name=ryan',
  searchParams: URLSearchParams { 'name' => 'ryan' },
  hash: ''
}
```

Class: `http.OutgoingMessage`

- Extends: {Stream}

This class serves as the parent class of `http.ClientRequest` and `http.ServerResponse`. It is an abstract of outgoing message from the perspective of the participants of HTTP transaction.

Event: `'drain'`

Emitted when the buffer of the message is free again.

Event: `'finish'`

Emitted when the transmission is finished successfully.

Event: `'prefinish'`

Emitted when `outgoingMessage.end` was called. When the event is emitted, all data has been processed but not necessarily completely flushed.

`outgoingMessage.addTrailers(headers)`

- `headers` {Object}

Adds HTTP trailers (headers but at the end of the message) to the message.

Trailers are **only** be emitted if the message is chunked encoded. If not, the trailer will be silently discarded.

HTTP requires the `Trailer` header to be sent to emit trailers, with a list of header fields in its value, e.g.

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

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

`outgoingMessage.connection`

> *Stability: 0 - Deprecated: Use [`outgoingMessage.socket`] instead.*

Aliases of `outgoingMessage.socket`

`outgoingMessage.cork()`

See [`writable.cork()`].

`outgoingMessage.destroy([error])`

- `error` {Error} Optional, an error to emit with `error` event
- Returns: {this}

Destroys the message. Once a socket is associated with the message and is connected, that socket will be destroyed as well.

`outgoingMessage.end(chunk[, encoding][, callback])`

- `chunk` {string | Buffer}
- `encoding` {string} Optional, **Default**: `utf8`
- `callback` {Function} Optional
- Returns: {this}

Finishes the outgoing message. If any parts of the body are unsent, it will flush them to the underlying system. If the message is chunked, it will send the terminating chunk `0\r\n\r\n`, and send the trailer (if any).

If `chunk` is specified, it is equivalent to call `outgoingMessage.write(chunk, encoding)`, followed by `outgoingMessage.end(callback)`.

If `callback` is provided, it will be called when the message is finished. (equivalent to the callback to event `finish`)

`outgoingMessage.flushHeaders()`

Compulsorily flushes the message headers

For efficiency reason, Node.js normally buffers the message headers until `outgoingMessage.end()` is called or the first chunk of message data is written. It then tries to pack the headers and data into a single TCP packet.

It is usually desired (it saves a TCP round-trip), but not when the first data is not sent until possibly much later. `outgoingMessage.flushHeaders()` bypasses the optimization and kickstarts the request.

`outgoingMessage.getHeader(name)`

- `name` {string} Name of header
- Returns {string | undefined}

Gets the value of HTTP header with the given name. If such a name doesn't exist in message, it will be `undefined`.

`outgoingMessage.getHeaderNames()`

- Returns {string[]}

Returns an array of names of headers of the outgoing outgoingMessage. All names are lowercase.

`outgoingMessage.getHeaders()`

- Returns: {Object}

Returns a shallow copy of the current outgoing headers. Since a shallow copy is used, array values may be mutated without additional calls to various header-related HTTP module methods. The keys of the returned object are the header names and the values are the respective header values. All header names are lowercase.

The object returned by the `outgoingMessage.getHeaders()` method does not prototypically inherit from the JavaScript Object. This means that typical Object methods such as `obj.toString()`, `obj.hasOwnProperty()`, and others are not defined and will not work.

```
outgoingMessage.setHeader('Foo', 'bar');
outgoingMessage.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

const headers = outgoingMessage.getHeaders();
// headers === { foo: 'bar', 'set-cookie': ['foo=bar', 'bar=baz'] }
```

`outgoingMessage.hasHeader(name)`

- `name` {string}
- Returns {boolean}

Returns `true` if the header identified by `name` is currently set in the outgoing headers. The header name is case-insensitive.

```
const hasContentType = outgoingMessage.hasHeader('content-type');
```

`outgoingMessage.headersSent`

- {boolean}

Read-only. `true` if the headers were sent, otherwise `false`.

`outgoingMessage.pipe()`

Overrides the pipe method of legacy `Stream` which is the parent class of `http.outgoingMessage`.

Since `OutgoingMessage` should be a write-only stream, call this function will throw an `Error`. Thus, it disabled the pipe method it inherits from `Stream`.

The User should not call this function directly.

`outgoingMessage.removeHeader()`

Removes a header that is queued for implicit sending.

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

`outgoingMessage.setHeader(name, value)`

- `name` {string} Header name
- `value` {string} Header value
- Returns: {this}

Sets a single header value for the header object.

`outgoingMessage.setTimeout(msesc[, callback])`

- `msesc` {number}
- `callback` {Function} Optional function to be called when a timeout occurs. Same as binding to the `timeout` event.
- Returns: {this}

Once a socket is associated with the message and is connected, `socket.setTimeout()` will be called with `msecs` as the first parameter.

`outgoingMessage.socket`

- {stream.Duplex}

Reference to the underlying socket. Usually, users will not want to access this property.

After calling `outgoingMessage.end()`, this property will be nulled.

`outgoingMessage.uncork()`

See `writable.uncork()`

`outgoingMessage.writableCorked`

- {number}

This `outgoingMessage.writableCorked` will return the time how many `outgoingMessage.cork()` have been called.

`outgoingMessage.writableEnded`

- {boolean}

Readonly, `true` if `outgoingMessage.end()` has been called. Noted that this property does not reflect whether the data has been flush. For that purpose, use `message.writableFinished` instead.

`outgoingMessage.writableFinished`

- {boolean}

Readonly. `true` if all data has been flushed to the underlying system.

`outgoingMessage.writableHighWaterMark`

- {number}

This `outgoingMessage.writableHighWaterMark` will be the `highWaterMark` of underlying socket if socket exists. Else, it would be the default `highWaterMark`.

`highWaterMark` is the maximum amount of data that can be potentially buffered by the socket.

`outgoingMessage.writableLength`

- {number}

Readonly, This `outgoingMessage.writableLength` contains the number of bytes (or objects) in the buffer ready to send.

`outgoingMessage.writableObjectMode`

- {boolean}

Readonly, always returns `false`.

`outgoingMessage.write(chunk[, encoding][, callback])`

- `chunk` {string | Buffer}
- `encoding` {string} **Default**: `utf8`
- `callback` {Function}
- Returns {boolean}

If this method is called and the header is not sent, it will call `this._implicitHeader` to flush implicit header. If the message should not have a body (indicated by `this._hasBody`), the call is ignored and `chunk` will not be sent. It could be useful when handling a particular message which must not include a body. e.g. response to `HEAD` request, `204` and `304` response.

`chunk` can be a string or a buffer. When `chunk` is a string, the `encoding` parameter specifies how to encode `chunk` into a byte stream. `callback` will be called when the `chunk` is flushed.

If the message is transferred in chucked encoding (indicated by `this.chunkedEncoding`), `chunk` will be flushed as one chunk among a stream of chunks. Otherwise, it will be flushed as the body of message.

This method handles the raw body of the HTTP message and has nothing to do with higher-level multi-part body encodings that may be used.

If it is the first call to this method of a message, it will send the buffered header first, then flush the `chunk` as described above.

The second and successive calls to this method will assume the data will be streamed and send the new data separately. It means that the response is buffered up to the first chunk of the 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 the user memory. Event `drain` will be emitted when the buffer is free again.

`http.METHODS`

- {string[]}

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.createServer([options][, requestListener])`

- `options` {Object}
    - `IncomingMessage` {http.IncomingMessage} Specifies the `IncomingMessage` class to be used. Useful for extending the original `IncomingMessage`. **Default:** `IncomingMessage`.
    - `ServerResponse` {http.ServerResponse} Specifies the `ServerResponse` class to be used. Useful for extending the original `ServerResponse`. **Default:** `ServerResponse`.
    - `insecureHTTPParser` {boolean} Use an insecure HTTP parser that accepts invalid HTTP headers when `true`. Using the insecure parser should be avoided. See `--insecure-http-parser` for more information. **Default:** `false`
    - `maxHeaderSize` {number} Optionally overrides the value of `--max-http-header-size` for requests received by this server, i.e. the maximum length of request headers in bytes. **Default:** 16384 (16 KB).
    - `noDelay` {boolean} If set to `true`, it disables the use of Nagle's algorithm immediately after a new incoming connection is received. **Default:** `true`.
    - `keepAlive` {boolean} If set to `true`, it enables keep-alive functionality on the socket immediately after a new incoming connection is received, similarly on what is done in [`socket.setKeepAlive([enable][, initialDelay])`] [`socket.setKeepAlive(enable, initialDelay)`]. **Default:** `false`.
    - `keepAliveInitialDelay` {number} If set to a positive number, it sets the initial delay before the first keepalive probe is sent on an idle socket. **Default:** `0`.
- `requestListener` {Function}
- Returns: {http.Server}

Returns a new instance of `http.Server`.

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

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

// Create a local server to receive data from
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({
    data: 'Hello World!'
  }));
});

server.listen(8000);
```

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

// Create a local server to receive data from
const server = http.createServer();

// Listen to the request event
server.on('request', (request, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({
    data: 'Hello World!'
  }));
});

server.listen(8000);
```

`http.get(options[, callback])`

`http.get(url[, options][, callback])`

- `url` {string | URL}
- `options` {Object} Accepts the same `options` as [`http.request()`], with the `method` always set to `GET`. Properties that are inherited from the prototype are ignored.
- `callback` {Function}
- Returns: {http.ClientRequest}

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. The callback must take care to consume the response data for reasons stated in [`http.ClientRequest`] section.

The `callback` is invoked with a single argument that is an instance of [`http.IncomingMessage`].

JSON fetching example:

```
http.get('http://localhost:8000/', (res) => {
  const { statusCode } = res;
  const contentType = res.headers['content-type'];
  let error;
  // Any 2xx status code signals a successful response but
  // here we're only checking for 200.
  if (statusCode !== 200) {
    error = new Error('Request Failed.\n' +
                      `Status Code: ${statusCode}`);
  } else if (!/^application\/json/.test(contentType)) {
    error = new Error('Invalid content-type.\n' +
                      `Expected application/json but received ${contentType}`);
  }
  if (error) {
    console.error(error.message);
    // Consume response data to free up memory
    res.resume();
    return;
  }

  res.setEncoding('utf8');
  let rawData = '';
  res.on('data', (chunk) => { rawData += chunk; });
  res.on('end', () => {
    try {
      const parsedData = JSON.parse(rawData);
      console.log(parsedData);
    } catch (e) {
      console.error(e.message);
    }
  });
}).on('error', (e) => {
  console.error(`Got error: ${e.message}`);
});

// Create a local server to receive data from
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({
    data: 'Hello World!'
  }));
});

server.listen(8000);
```

`http.globalAgent`

- {http.Agent}

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

`http.maxHeaderSize`

- {number}

Read-only property specifying the maximum allowed size of HTTP headers in bytes. Defaults to 16 KB. Configurable using the `--max-http-header-size` CLI option.

This can be overridden for servers and client requests by passing the `maxHeaderSize` option.

`http.request(options[, callback])`

`http.request(url[, options][, callback])`

- `url` {string | URL}
- `options` {Object}
  - `agent` {http.Agent | boolean} Controls `Agent` behavior. Possible values:
    - `undefined` (default): use `http.globalAgent` for this host and port.
    - `Agent` object: explicitly use the passed in `Agent` .
    - `false` : causes a new `Agent` with default values to be used.
  - `auth` {string} Basic authentication (`'user:password'`) to compute an Authorization header.
  - `createConnection` {Function} 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. Any `Duplex` stream is a valid return value.
  - `defaultPort` {number} Default port for the protocol. **Default:** `agent.defaultPort` if an `Agent` is used, else `undefined` .
  - `family` {number} IP address family to use when resolving `host` or `hostname` . Valid values are `4` or `6` . When unspecified, both IP v4 and v6 will be used.
  - `headers` {Object} An object containing request headers.
  - `hints` {number} Optional `dns.lookup()` hints.
  - `host` {string} A domain name or IP address of the server to issue the request to. **Default:** `'localhost'` .
  - `hostname` {string} Alias for `host` . To support `url.parse()` , `hostname` will be used if both `host` and `hostname` are specified.
  - `insecureHTTPParser` {boolean} Use an insecure HTTP parser that accepts invalid HTTP headers when `true` . Using the insecure parser should be avoided. See `--insecure-http-parser` for more information. **Default:** `false`
  - `localAddress` {string} Local interface to bind for network connections.
  - `localPort` {number} Local port to connect from.
  - `lookup` {Function} Custom lookup function. **Default:** `dns.lookup()` .
  - `maxHeaderSize` {number} Optionally overrides the value of `--max-http-header-size` (the maximum length of response headers in bytes) for responses received from the server. **Default:** 16384 (16 KB).
  - `method` {string} A string specifying the HTTP request method. **Default:** `'GET'` .
  - `path` {string} Request path. 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. **Default:** `'/'` .
  - `port` {number} Port of remote server. **Default:** `defaultPort` if set, else `80` .
  - `protocol` {string} Protocol to use. **Default:** `'http:'` .

  - `setHost` {boolean}: Specifies whether or not to automatically add the `Host` header. Defaults to `true` .
  - `socketPath` {string} Unix domain socket. Cannot be used if one of `host` or `port` is specified, as those specify a TCP Socket.
  - `timeout` {number}: A number specifying the socket timeout in milliseconds. This will set the timeout before the socket is connected.
  - `signal` {AbortSignal}: An AbortSignal that may be used to abort an ongoing request.
- `callback` {Function}
- Returns: {http.ClientRequest}

`options` in `socket.connect()` are also supported.

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

`url` can be a string or a `URL` object. If `url` is a string, it is automatically parsed with `new URL()` . If it is a `URL` object, it will be automatically converted to an ordinary `options` object.

If both `url` and `options` are specified, the objects are merged, with the `options` properties taking precedence.

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.

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

const postData = JSON.stringify({
  'msg': 'Hello World!'
});

const options = {
  hostname: 'www.google.com',
  port: 80,
  path: '/upload',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': Buffer.byteLength(postData)
  }
};

const 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.error(`problem with request: ${e.message}`);
});

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

In the example `req.end()` was called. With `http.request()` one must always call `req.end()` to signify the end of 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', both a timeout and a listener for the `'continue'` event should be set. See RFC 2616 Section 8.2.3 for more information.
- Sending an Authorization header will override using the `auth` option to compute basic authentication.

Example using a `URL` as `options`:

```
const options = new URL('http://abc:xyz@example.com');

const req = http.request(options, (res) => {
  // ...
});
```

In a successful request, the following events will be emitted in the following order:

- `'socket'`
- `'response'`
  - `'data'` any number of times, on the `res` object (`'data'` will not be emitted at all if the response body is empty, for instance, in most redirects)
  - `'end'` on the `res` object
- `'close'`

In the case of a connection error, the following events will be emitted:

- `'socket'`
- `'error'`

- `'close'`

In the case of a premature connection close before the response is received, the following events will be emitted in the following order:

- `'socket'`
- `'error'` with an error with message `'Error: socket hang up'` and code `'ECONNRESET'`
- `'close'`

In the case of a premature connection close after the response is received, the following events will be emitted in the following order:

- `'socket'`
- `'response'`
  - `'data'` any number of times, on the `res` object
- (connection closed here)
- `'aborted'` on the `res` object
- `'error'` on the `res` object with an error with message `'Error: aborted'` and code `'ECONNRESET'`.
- `'close'`
- `'close'` on the `res` object

If `req.destroy()` is called before a socket is assigned, the following events will be emitted in the following order:

- (`req.destroy()` called here)
- `'error'` with an error with message `'Error: socket hang up'` and code `'ECONNRESET'`
- `'close'`

If `req.destroy()` is called before the connection succeeds, the following events will be emitted in the following order:

- `'socket'`
- (`req.destroy()` called here)
- `'error'` with an error with message `'Error: socket hang up'` and code `'ECONNRESET'`
- `'close'`

If `req.destroy()` is called after the response is received, the following events will be emitted in the following order:

- `'socket'`
- `'response'`
  - `'data'` any number of times, on the `res` object
- (`req.destroy()` called here)
- `'aborted'` on the `res` object
- `'error'` on the `res` object with an error with message `'Error: aborted'` and code `'ECONNRESET'`.
- `'close'`
- `'close'` on the `res` object

If `req.abort()` is called before a socket is assigned, the following events will be emitted in the following order:

- (`req.abort()` called here)

- `'abort'`
- `'close'`

If `req.abort()` is called before the connection succeeds, the following events will be emitted in the following order:

- `'socket'`
- (`req.abort()` called here)
- `'abort'`
- `'error'` with an error with message `'Error: socket hang up'` and code `'ECONNRESET'`
- `'close'`

If `req.abort()` is called after the response is received, the following events will be emitted in the following order:

- `'socket'`
- `'response'`
  - `'data'` any number of times, on the `res` object
- (`req.abort()` called here)
- `'abort'`
- `'aborted'` on the `res` object
- `'error'` on the `res` object with an error with message `'Error: aborted'` and code `'ECONNRESET'`.
- `'close'`
- `'close'` on the `res` object

Setting the `timeout` option or using the `setTimeout()` function will not abort the request or do anything besides add a `'timeout'` event.

Passing an `AbortSignal` and then calling `abort` on the corresponding `AbortController` will behave the same way as calling `.destroy()` on the request itself.

`http.validateHeaderName(name)`

- `name` {string}

Performs the low-level validations on the provided `name` that are done when `res.setHeader(name, value)` is called.

Passing illegal value as `name` will result in a [`TypeError`] being thrown, identified by `code: 'ERR_INVALID_HTTP_TOKEN'`.

It is not necessary to use this method before passing headers to an HTTP request or response. The HTTP module will automatically validate such headers. Examples:

Example:

```
const { validateHeaderName } = require('http');

try {
  validateHeaderName('');
} catch (err) {
  err instanceof TypeError; // --> true
  err.code; // --> 'ERR_INVALID_HTTP_TOKEN'
  err.message; // --> 'Header name must be a valid HTTP token [""]'
}
```

`http.validateHeaderValue(name, value)`

- `name` {string}
- `value` {any}

Performs the low-level validations on the provided `value` that are done when `res.setHeader(name, value)` is called.

Passing illegal value as `value` will result in a [`TypeError`](#) being thrown.

- Undefined value error is identified by `code: 'ERR_HTTP_INVALID_HEADER_VALUE'`.
- Invalid value character error is identified by `code: 'ERR_INVALID_CHAR'`.

It is not necessary to use this method before passing headers to an HTTP request or response. The HTTP module will automatically validate such headers.

Examples:

```
const { validateHeaderValue } = require('http');

try {
  validateHeaderValue('x-my-header', undefined);
} catch (err) {
  err instanceof TypeError; // --> true
  err.code === 'ERR_HTTP_INVALID_HEADER_VALUE'; // --> true
  err.message; // --> 'Invalid value "undefined" for header "x-my-header"'
}

try {
  validateHeaderValue('x-my-header', 'oʊmɪgə');
} catch (err) {
  err instanceof TypeError; // --> true
  err.code === 'ERR_INVALID_CHAR'; // --> true
  err.message; // --> 'Invalid character in header content ["x-my-header"]'
}
```