HTTP/2

Stability: 2 - Stable

Source Code: lib/http2.js

The `node:http2` module provides an implementation of the HTTP/2 protocol. It can be accessed using:

```
const http2 = require('node:http2');
```

COPY

Determining if crypto support is unavailable

It is possible for Node.js to be built without including support for the `node:crypto` module. In such cases, attempting to `import` from `node:http2` or calling `require('node:http2')` will result in an error being thrown.

When using CommonJS, the error thrown can be caught using try/catch:

```
let http2;
try {
  http2 = require('node:http2');
} catch (err) {
  console.error('http2 support is disabled!');
}
```

COPY

When using the lexical ESM `import` keyword, the error can only be caught if a handler for `process.on('uncaughtException')` is registered *before* any attempt to load the module is made (using, for instance, a preload module).

When using ESM, if there is a chance that the code may be run on a build of Node.js where crypto support is not enabled, consider using the `import()` function instead of the lexical `import` keyword:

```
let http2;
try {
  http2 = await import('node:http2');
} catch (err) {
  console.error('http2 support is disabled!');
}
```

COPY

Core API

The Core API provides a low-level interface designed specifically around support for HTTP/2 protocol features. It is specifically *not* designed for compatibility with the existing HTTP/1 module API. However, the Compatibility API is.

The `http2` Core API is much more symmetric between client and server than the `http` API. For instance, most events, like `'error'`, `'connect'` and `'stream'`, can be emitted either by client-side code or server-side code.

Server-side example

The following illustrates a simple HTTP/2 server using the Core API. Since there are no browsers known that support unencrypted HTTP/2, the use of `http2.createSecureServer()` is necessary when communicating with browser clients.

```mjs
import { createSecureServer } from 'node:http2';
import { readFileSync } from 'node:fs';

const server = createSecureServer({
  key: readFileSync('localhost-privkey.pem'),
  cert: readFileSync('localhost-cert.pem'),
});

server.on('error', (err) => console.error(err));

server.on('stream', (stream, headers) => {
  // stream is a Duplex
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200,
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(8443);
```

```cjs
const http2 = require('node:http2');
const fs = require('node:fs');

const server = http2.createSecureServer({
  key: fs.readFileSync('localhost-privkey.pem'),
  cert: fs.readFileSync('localhost-cert.pem'),
});
server.on('error', (err) => console.error(err));

server.on('stream', (stream, headers) => {
  // stream is a Duplex
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200,
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(8443);
```

COPY

To generate the certificate and key for this example, run:

```bash
openssl req -x509 -newkey rsa:2048 -nodes -sha256 -subj '/CN=localhost' \
  -keyout localhost-privkey.pem -out localhost-cert.pem
```

COPY

Client-side example

The following illustrates an HTTP/2 client:

```mjs
import { connect } from 'node:http2';
import { readFileSync } from 'node:fs';

const client = connect('https://localhost:8443', {
  ca: readFileSync('localhost-cert.pem'),
});
client.on('error', (err) => console.error(err));

const req = client.request({ ':path': '/' });

req.on('response', (headers, flags) => {
  for (const name in headers) {
    console.log(`${name}: ${headers[name]}`);
  }
});

req.setEncoding('utf8');
let data = '';
req.on('data', (chunk) => { data += chunk; });
req.on('end', () => {
  console.log(`\n${data}`);
  client.close();
});
req.end();
```

```cjs
const http2 = require('node:http2');
const fs = require('node:fs');

const client = http2.connect('https://localhost:8443', {
  ca: fs.readFileSync('localhost-cert.pem'),
});
client.on('error', (err) => console.error(err));

const req = client.request({ ':path': '/' });

req.on('response', (headers, flags) => {
  for (const name in headers) {
    console.log(`${name}: ${headers[name]}`);
  }
});

req.setEncoding('utf8');
let data = '';
req.on('data', (chunk) => { data += chunk; });
req.on('end', () => {
  console.log(`\n${data}`);
  client.close();
```

```
});
req.end();
```

COPY

Class: Http2Session

- Extends: <EventEmitter>

Instances of the `http2.Http2Session` class represent an active communications session between an HTTP/2 client and server. Instances of this class are *not* intended to be constructed directly by user code.

Each `Http2Session` instance will exhibit slightly different behaviors depending on whether it is operating as a server or a client. The `http2session.type` property can be used to determine the mode in which an `Http2Session` is operating. On the server side, user code should rarely have occasion to work with the `Http2Session` object directly, with most actions typically taken through interactions with either the `Http2Server` or `Http2Stream` objects.

User code will not create `Http2Session` instances directly. Server-side `Http2Session` instances are created by the `Http2Server` instance when a new HTTP/2 connection is received. Client-side `Http2Session` instances are created using the `http2.connect()` method.

Http2Session and sockets

Every `Http2Session` instance is associated with exactly one `net.Socket` or `tls.TLSSocket` when it is created. When either the `Socket` or the `Http2Session` are destroyed, both will be destroyed.

Because of the specific serialization and processing requirements imposed by the HTTP/2 protocol, it is not recommended for user code to read data from or write data to a `Socket` instance bound to a `Http2Session` . Doing so can put the HTTP/2 session into an indeterminate state causing the session and the socket to become unusable.

Once a `Socket` has been bound to an `Http2Session` , user code should rely solely on the API of the `Http2Session` .

Event: 'close'

The `'close'` event is emitted once the `Http2Session` has been destroyed. Its listener does not expect any arguments.

Event: 'connect'

- `session` <Http2Session>
- `socket` <net.Socket>

The `'connect'` event is emitted once the `Http2Session` has been successfully connected to the remote peer and communication may begin.

User code will typically not listen for this event directly.

Event: 'error'

- `error` <Error>

The `'error'` event is emitted when an error occurs during the processing of an `Http2Session` .

Event: 'frameError'

- `type` <integer> The frame type.
- `code` <integer> The error code.
- `id` <integer> The stream id (or `0` if the frame isn't associated with a stream).

The `'frameError'` event is emitted when an error occurs while attempting to send a frame on the session. If the frame that could not be sent is associated with a specific `Http2Stream` , an attempt to emit a `'frameError'` event on the `Http2Stream` is made.

If the `'frameError'` event is associated with a stream, the stream will be closed and destroyed immediately following the `'frameError'` event. If the event is not associated with a stream, the `Http2Session` will be shut down immediately following the `'frameError'` event.

Event: 'goaway'

- `errorCode` <number> The HTTP/2 error code specified in the `GOAWAY` frame.
- `lastStreamID` <number> The ID of the last stream the remote peer successfully processed (or `0` if no ID is specified).
- `opaqueData` <Buffer> If additional opaque data was included in the `GOAWAY` frame, a `Buffer` instance will be passed containing that data.

The `'goaway'` event is emitted when a `GOAWAY` frame is received.

The `Http2Session` instance will be shut down automatically when the `'goaway'` event is emitted.

Event: 'localSettings'

- `settings` <HTTP/2 Settings Object> A copy of the `SETTINGS` frame received.

The `'localSettings'` event is emitted when an acknowledgment `SETTINGS` frame has been received.

When using `http2session.settings()` to submit new settings, the modified settings do not take effect until the `'localSettings'` event is emitted.

```
session.settings({ enablePush: false });

session.on('localSettings', (settings) => {
  /* Use the new settings */
});
```

COPY

Event: 'ping'

- `payload` <Buffer> The `PING` frame 8-byte payload

The `'ping'` event is emitted whenever a `PING` frame is received from the connected peer.

Event: 'remoteSettings'

- `settings` <HTTP/2 Settings Object> A copy of the `SETTINGS` frame received.

The `'remoteSettings'` event is emitted when a new `SETTINGS` frame is received from the connected peer.

```
session.on('remoteSettings', (settings) => {
  /* Use the new settings */
});
```

COPY

Event: 'stream'

- `stream` <Http2Stream> A reference to the stream
- `headers` <HTTP/2 Headers Object> An object describing the headers
- `flags` <number> The associated numeric flags
- `rawHeaders` <Array> An array containing the raw header names followed by their respective values.

The `'stream'` event is emitted when a new `Http2Stream` is created.

```
session.on('stream', (stream, headers, flags) => {
  const method = headers[':method'];
  const path = headers[':path'];
  // ...
  stream.respond({
    ':status': 200,
    'content-type': 'text/plain; charset=utf-8',
  });
  stream.write('hello ');
  stream.end('world');
});
```

COPY

On the server side, user code will typically not listen for this event directly, and would instead register a handler for the `'stream'` event emitted by the `net.Server` or `tls.Server` instances returned by `http2.createServer()` and `http2.createSecureServer()`, respectively, as in the example below:

```
import { createServer } from 'node:http2';

// Create an unencrypted HTTP/2 server
const server = createServer();

server.on('stream', (stream, headers) => {
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200,
  });
  stream.on('error', (error) => console.error(error));
  stream.end('<h1>Hello World</h1>');
});

server.listen(8000);
```

```
const http2 = require('node:http2');

// Create an unencrypted HTTP/2 server
const server = http2.createServer();

server.on('stream', (stream, headers) => {
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200,
  });
  stream.on('error', (error) => console.error(error));
  stream.end('<h1>Hello World</h1>');
});

server.listen(8000);
```

COPY

Even though HTTP/2 streams and network sockets are not in a 1:1 correspondence, a network error will destroy each individual stream and must be handled on the stream level, as shown above.

Event: 'timeout'

After the `http2session.setTimeout()` method is used to set the timeout period for this `Http2Session`, the `'timeout'` event is emitted if there is no activity on the `Http2Session` after the configured number of milliseconds. Its listener does not expect any arguments.

```
session.setTimeout(2000);
session.on('timeout', () => { /* .. */ });
```
COPY

http2session.alpnProtocol

- <string> | <undefined>

Value will be `undefined` if the `Http2Session` is not yet connected to a socket, `h2c` if the `Http2Session` is not connected to a `TLSSocket`, or will return the value of the connected `TLSSocket`'s own `alpnProtocol` property.

http2session.close([callback])

- `callback` <Function>

Gracefully closes the `Http2Session`, allowing any existing streams to complete on their own and preventing new `Http2Stream` instances from being created. Once closed, `http2session.destroy()` *might* be called if there are no open `Http2Stream` instances.

If specified, the `callback` function is registered as a handler for the `'close'` event.

http2session.closed

- <boolean>

Will be `true` if this `Http2Session` instance has been closed, otherwise `false`.

http2session.connecting

- <boolean>

Will be `true` if this `Http2Session` instance is still connecting, will be set to `false` before emitting `connect` event and/or calling the `http2.connect` callback.

http2session.destroy([error][, code])

- `error` <Error> An `Error` object if the `Http2Session` is being destroyed due to an error.
- `code` <number> The HTTP/2 error code to send in the final `GOAWAY` frame. If unspecified, and `error` is not undefined, the default is `INTERNAL_ERROR`, otherwise defaults to `NO_ERROR`.

Immediately terminates the `Http2Session` and the associated `net.Socket` or `tls.TLSSocket`.

Once destroyed, the `Http2Session` will emit the `'close'` event. If `error` is not undefined, an `'error'` event will be emitted immediately before the `'close'` event.

If there are any remaining open `Http2Streams` associated with the `Http2Session`, those will also be destroyed.

http2session.destroyed

- <boolean>

Will be `true` if this `Http2Session` instance has been destroyed and must no longer be used, otherwise `false`.

http2session.encrypted

- <boolean> | <undefined>

Value is `undefined` if the `Http2Session` session socket has not yet been connected, `true` if the `Http2Session` is connected with a `TLSSocket`, and `false` if the `Http2Session` is connected to any other kind of socket or stream.

http2session.goaway([code[, lastStreamID[, opaqueData]]])

- `code` <number> An HTTP/2 error code
- `lastStreamID` <number> The numeric ID of the last processed `Http2Stream`
- `opaqueData` <Buffer> | <TypedArray> | <DataView> A `TypedArray` or `DataView` instance containing additional data to be carried within the `GOAWAY` frame.

Transmits a `GOAWAY` frame to the connected peer *without* shutting down the `Http2Session`.

http2session.localSettings

- <HTTP/2 Settings Object>

A prototype-less object describing the current local settings of this `Http2Session`. The local settings are local to *this* `Http2Session` instance.

http2session.originSet

- <string[]> | <undefined>

If the `Http2Session` is connected to a `TLSSocket`, the `originSet` property will return an `Array` of origins for which the `Http2Session` may be considered authoritative.

The `originSet` property is only available when using a secure TLS connection.

http2session.pendingSettingsAck

- <boolean>

Indicates whether the `Http2Session` is currently waiting for acknowledgment of a sent `SETTINGS` frame. Will be `true` after calling the `http2session.settings()` method. Will be `false` once all sent `SETTINGS` frames have been acknowledged.

http2session.ping([payload,]callback)

- `payload` <Buffer> | <TypedArray> | <DataView> Optional ping payload.
- `callback` <Function>
- Returns: <boolean>

Sends a `PING` frame to the connected HTTP/2 peer. A `callback` function must be provided. The method will return `true` if the `PING` was sent, `false` otherwise.

The maximum number of outstanding (unacknowledged) pings is determined by the `maxOutstandingPings` configuration option. The default maximum is 10.

If provided, the `payload` must be a `Buffer`, `TypedArray`, or `DataView` containing 8 bytes of data that will be transmitted with the `PING` and returned with the ping acknowledgment.

The callback will be invoked with three arguments: an error argument that will be `null` if the `PING` was successfully acknowledged, a `duration` argument that reports the number of milliseconds elapsed since the ping was sent and the acknowledgment was received, and a `Buffer` containing the 8-byte `PING` payload.

```
session.ping(Buffer.from('abcdefgh'), (err, duration, payload) => {
  if (!err) {
    console.log(`Ping acknowledged in ${duration} milliseconds`);
    console.log(`With payload '${payload.toString()}'`);
  }
});
```

COPY

If the `payload` argument is not specified, the default payload will be the 64-bit timestamp (little endian) marking the start of the `PING` duration.

http2session.ref()

Calls `ref()` on this `Http2Session` instance's underlying `net.Socket`.

http2session.remoteSettings

- `<HTTP/2 Settings Object>`

A prototype-less object describing the current remote settings of this `Http2Session`. The remote settings are set by the *connected* HTTP/2 peer.

http2session.setLocalWindowSize(windowSize)

- `windowSize` `<number>`

Sets the local endpoint's window size. The `windowSize` is the total window size to set, not the delta.

```
import { createServer } from 'node:http2';

const server = createServer();
const expectedWindowSize = 2 ** 20;
server.on('session', (session) => {

  // Set local window size to be 2 ** 20
  session.setLocalWindowSize(expectedWindowSize);
});
```

```
const http2 = require('node:http2');

const server = http2.createServer();
const expectedWindowSize = 2 ** 20;
server.on('session', (session) => {

  // Set local window size to be 2 ** 20
  session.setLocalWindowSize(expectedWindowSize);
});
```

COPY

For http2 clients the proper event is either `'connect'` or `'remoteSettings'`.

http2session.setTimeout(msecs, callback)

- `msecs` `<number>`

- `callback` <Function>

Used to set a callback function that is called when there is no activity on the `Http2Session` after `msecs` milliseconds. The given `callback` is registered as a listener on the `'timeout'` event.

http2session.socket

- <net.Socket> | <tls.TLSSocket>

Returns a `Proxy` object that acts as a `net.Socket` (or `tls.TLSSocket`) but limits available methods to ones safe to use with HTTP/2.

`destroy`, `emit`, `end`, `pause`, `read`, `resume`, and `write` will throw an error with code `ERR_HTTP2_NO_SOCKET_MANIPULATION`. See Http2Session and Sockets for more information.

`setTimeout` method will be called on this `Http2Session`.

All other interactions will be routed directly to the socket.

http2session.state

Provides miscellaneous information about the current state of the `Http2Session`.

- <Object>
  - `effectiveLocalWindowSize` <number> The current local (receive) flow control window size for the `Http2Session`.
  - `effectiveRecvDataLength` <number> The current number of bytes that have been received since the last flow control `WINDOW_UPDATE`.
  - `nextStreamID` <number> The numeric identifier to be used the next time a new `Http2Stream` is created by this `Http2Session`.
  - `localWindowSize` <number> The number of bytes that the remote peer can send without receiving a `WINDOW_UPDATE`.
  - `lastProcStreamID` <number> The numeric id of the `Http2Stream` for which a `HEADERS` or `DATA` frame was most recently received.
  - `remoteWindowSize` <number> The number of bytes that this `Http2Session` may send without receiving a `WINDOW_UPDATE`.
  - `outboundQueueSize` <number> The number of frames currently within the outbound queue for this `Http2Session`.
  - `deflateDynamicTableSize` <number> The current size in bytes of the outbound header compression state table.
  - `inflateDynamicTableSize` <number> The current size in bytes of the inbound header compression state table.

An object describing the current status of this `Http2Session`.

http2session.settings([settings][, callback])

- `settings` <HTTP/2 Settings Object>
- `callback` <Function> Callback that is called once the session is connected or right away if the session is already connected.
  - `err` <Error> | <null>
  - `settings` <HTTP/2 Settings Object> The updated `settings` object.
  - `duration` <integer>

Updates the current local settings for this `Http2Session` and sends a new `SETTINGS` frame to the connected HTTP/2 peer.

Once called, the `http2session.pendingSettingsAck` property will be `true` while the session is waiting for the remote peer to acknowledge the new settings.

The new settings will not become effective until the `SETTINGS` acknowledgment is received and the `'localSettings'` event is emitted. It is possible to send multiple `SETTINGS` frames while acknowledgment is still pending.

http2session.type

- <number>

The `http2session.type` will be equal to `http2.constants.NGHTTP2_SESSION_SERVER` if this `Http2Session` instance is a server, and `http2.constants.NGHTTP2_SESSION_CLIENT` if the instance is a client.

http2session.unref()

Calls `unref()` on this `Http2Session` instance's underlying `net.Socket`.

Class: ServerHttp2Session

- Extends: `<Http2Session>`

serverhttp2session.altsvc(alt, originOrStream)

- `alt` `<string>` A description of the alternative service configuration as defined by RFC 7838.
- `originOrStream` `<number>` | `<string>` | `<URL>` | `<Object>` Either a URL string specifying the origin (or an `Object` with an `origin` property) or the numeric identifier of an active `Http2Stream` as given by the `http2stream.id` property.

Submits an `ALTSVC` frame (as defined by RFC 7838) to the connected client.

```
import { createServer } from 'node:http2';

const server = createServer();
server.on('session', (session) => {
  // Set altsvc for origin https://example.org:80
  session.altsvc('h2=":8000"', 'https://example.org:80');
});

server.on('stream', (stream) => {
  // Set altsvc for a specific stream
  stream.session.altsvc('h2=":8000"', stream.id);
});
```

```
const http2 = require('node:http2');

const server = http2.createServer();
server.on('session', (session) => {
  // Set altsvc for origin https://example.org:80
  session.altsvc('h2=":8000"', 'https://example.org:80');
});

server.on('stream', (stream) => {
  // Set altsvc for a specific stream
  stream.session.altsvc('h2=":8000"', stream.id);
});
```

COPY

Sending an `ALTSVC` frame with a specific stream ID indicates that the alternate service is associated with the origin of the given `Http2Stream`.

The `alt` and origin string *must* contain only ASCII bytes and are strictly interpreted as a sequence of ASCII bytes. The special value `'clear'` may be passed to clear any previously set alternative service for a given domain.

When a string is passed for the `originOrStream` argument, it will be parsed as a URL and the origin will be derived. For instance, the origin for the HTTP URL `'https://example.org/foo/bar'` is the ASCII string `'https://example.org'`. An error will be thrown if either the given string

cannot be parsed as a URL or if a valid origin cannot be derived.

A `URL` object, or any object with an `origin` property, may be passed as `originOrStream`, in which case the value of the `origin` property will be used. The value of the `origin` property *must* be a properly serialized ASCII origin.

Specifying alternative services

The format of the `alt` parameter is strictly defined by RFC 7838 as an ASCII string containing a comma-delimited list of "alternative" protocols associated with a specific host and port.

For example, the value `'h2="example.org:81"'` indicates that the HTTP/2 protocol is available on the host `'example.org'` on TCP/IP port 81. The host and port *must* be contained within the quote (`"`) characters.

Multiple alternatives may be specified, for instance: `'h2="example.org:81", h2=":82"'`.

The protocol identifier (`'h2'` in the examples) may be any valid ALPN Protocol ID.

The syntax of these values is not validated by the Node.js implementation and are passed through as provided by the user or received from the peer.

serverhttp2session.origin(...origins)

- `origins` `<string>` | `<URL>` | `<Object>` One or more URL Strings passed as separate arguments.

Submits an `ORIGIN` frame (as defined by RFC 8336) to the connected client to advertise the set of origins for which the server is capable of providing authoritative responses.

```
import { createSecureServer } from 'node:http2';
const options = getSecureOptionsSomehow();
const server = createSecureServer(options);
server.on('stream', (stream) => {
  stream.respond();
  stream.end('ok');
});
server.on('session', (session) => {
  session.origin('https://example.com', 'https://example.org');
});
```

```
const http2 = require('node:http2');
const options = getSecureOptionsSomehow();
const server = http2.createSecureServer(options);
server.on('stream', (stream) => {
  stream.respond();
  stream.end('ok');
});
server.on('session', (session) => {
  session.origin('https://example.com', 'https://example.org');
});
```

COPY

When a string is passed as an `origin`, it will be parsed as a URL and the origin will be derived. For instance, the origin for the HTTP URL `'https://example.org/foo/bar'` is the ASCII string `'https://example.org'`. An error will be thrown if either the given string cannot be parsed as a URL or if a valid origin cannot be derived.

A `URL` object, or any object with an `origin` property, may be passed as an `origin`, in which case the value of the `origin` property will be used. The value of the `origin` property *must* be a properly serialized ASCII origin.

Alternatively, the `origins` option may be used when creating a new HTTP/2 server using the `http2.createSecureServer()` method:

```mjs
import { createSecureServer } from 'node:http2';
const options = getSecureOptionsSomehow();
options.origins = ['https://example.com', 'https://example.org'];
const server = createSecureServer(options);
server.on('stream', (stream) => {
  stream.respond();
  stream.end('ok');
});
```

```cjs
const http2 = require('node:http2');
const options = getSecureOptionsSomehow();
options.origins = ['https://example.com', 'https://example.org'];
const server = http2.createSecureServer(options);
server.on('stream', (stream) => {
  stream.respond();
  stream.end('ok');
});
```

COPY

Class: ClientHttp2Session

- Extends: <Http2Session>

Event: 'altsvc'

- `alt` <string>
- `origin` <string>
- `streamId` <number>

The `'altsvc'` event is emitted whenever an `ALTSVC` frame is received by the client. The event is emitted with the `ALTSVC` value, origin, and stream ID. If no `origin` is provided in the `ALTSVC` frame, `origin` will be an empty string.

```mjs
import { connect } from 'node:http2';
const client = connect('https://example.org');

client.on('altsvc', (alt, origin, streamId) => {
  console.log(alt);
  console.log(origin);
  console.log(streamId);
});
```

```cjs
const http2 = require('node:http2');
const client = http2.connect('https://example.org');

client.on('altsvc', (alt, origin, streamId) => {
```

```
  console.log(alt);
  console.log(origin);
  console.log(streamId);
});
```

COPY

Event: 'origin'

- `origins` <string[]>

The `'origin'` event is emitted whenever an `ORIGIN` frame is received by the client. The event is emitted with an array of `origin` strings. The `http2session.originSet` will be updated to include the received origins.

```
import { connect } from 'node:http2';
const client = connect('https://example.org');

client.on('origin', (origins) => {
  for (let n = 0; n < origins.length; n++)
    console.log(origins[n]);
});
```

```
const http2 = require('node:http2');
const client = http2.connect('https://example.org');

client.on('origin', (origins) => {
  for (let n = 0; n < origins.length; n++)
    console.log(origins[n]);
});
```

COPY

The `'origin'` event is only emitted when using a secure TLS connection.

clienthttp2session.request(headers[, options])

- `headers` <HTTP/2 Headers Object>
- `options` <Object>
  - `endStream` <boolean> `true` if the `Http2Stream` *writable* side should be closed initially, such as when sending a `GET` request that should not expect a payload body.
  - `exclusive` <boolean> When `true` and `parent` identifies a parent Stream, the created stream is made the sole direct dependency of the parent, with all other existing dependents made a dependent of the newly created stream. **Default:** `false`.
  - `parent` <number> Specifies the numeric identifier of a stream the newly created stream is dependent on.
  - `weight` <number> Specifies the relative dependency of a stream in relation to other streams with the same `parent`. The value is a number between `1` and `256` (inclusive).
  - `waitForTrailers` <boolean> When `true`, the `Http2Stream` will emit the `'wantTrailers'` event after the final `DATA` frame has been sent.
  - `signal` <AbortSignal> An AbortSignal that may be used to abort an ongoing request.
- Returns: <ClientHttp2Stream>

For HTTP/2 Client `Http2Session` instances only, the `http2session.request()` creates and returns an `Http2Stream` instance that can be used to send an HTTP/2 request to the connected server.

When a `ClientHttp2Session` is first created, the socket may not yet be connected. if `clienthttp2session.request()` is called during this time, the actual request will be deferred until the socket is ready to go. If the `session` is closed before the actual request be executed, an `ERR_HTTP2_GOAWAY_SESSION` is thrown.

This method is only available if `http2session.type` is equal to `http2.constants.NGHTTP2_SESSION_CLIENT`.

```
import { connect, constants } from 'node:http2';
const clientSession = connect('https://localhost:1234');
const {
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS,
} = constants;

const req = clientSession.request({ [HTTP2_HEADER_PATH]: '/' });
req.on('response', (headers) => {
  console.log(headers[HTTP2_HEADER_STATUS]);
  req.on('data', (chunk) => { /* .. */ });
  req.on('end', () => { /* .. */ });
});
```

```
const http2 = require('node:http2');
const clientSession = http2.connect('https://localhost:1234');
const {
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS,
} = http2.constants;

const req = clientSession.request({ [HTTP2_HEADER_PATH]: '/' });
req.on('response', (headers) => {
  console.log(headers[HTTP2_HEADER_STATUS]);
  req.on('data', (chunk) => { /* .. */ });
  req.on('end', () => { /* .. */ });
});
```

COPY

When the `options.waitForTrailers` option is set, the `'wantTrailers'` event is emitted immediately after queuing the last chunk of payload data to be sent. The `http2stream.sendTrailers()` method can then be called to send trailing headers to the peer.

When `options.waitForTrailers` is set, the `Http2Stream` will not automatically close when the final `DATA` frame is transmitted. User code must call either `http2stream.sendTrailers()` or `http2stream.close()` to close the `Http2Stream`.

When `options.signal` is set with an `AbortSignal` and then `abort` on the corresponding `AbortController` is called, the request will emit an `'error'` event with an `AbortError` error.

The `:method` and `:path` pseudo-headers are not specified within `headers`, they respectively default to:

- `:method = 'GET'`
- `:path = /`

Class: Http2Stream

- Extends: <stream.Duplex>

Each instance of the `Http2Stream` class represents a bidirectional HTTP/2 communications stream over an `Http2Session` instance. Any single `Http2Session` may have up to 2^{31}-1 `Http2Stream` instances over its lifetime.

User code will not construct `Http2Stream` instances directly. Rather, these are created, managed, and provided to user code through the `Http2Session` instance. On the server, `Http2Stream` instances are created either in response to an incoming HTTP request (and handed off to user code via the `'stream'` event), or in response to a call to the `http2stream.pushStream()` method. On the client, `Http2Stream` instances are created and returned when either the `http2session.request()` method is called, or in response to an incoming `'push'` event.

The `Http2Stream` class is a base for the `ServerHttp2Stream` and `ClientHttp2Stream` classes, each of which is used specifically by either the Server or Client side, respectively.

All `Http2Stream` instances are `Duplex` streams. The `Writable` side of the `Duplex` is used to send data to the connected peer, while the `Readable` side is used to receive data sent by the connected peer.

The default text character encoding for an `Http2Stream` is UTF-8. When using an `Http2Stream` to send text, use the `'content-type'` header to set the character encoding.

```
stream.respond({
  'content-type': 'text/html; charset=utf-8',
  ':status': 200,
});
```

COPY

Http2Stream Lifecycle

Creation

On the server side, instances of `ServerHttp2Stream` are created either when:

- A new HTTP/2 `HEADERS` frame with a previously unused stream ID is received;
- The `http2stream.pushStream()` method is called.

On the client side, instances of `ClientHttp2Stream` are created when the `http2session.request()` method is called.

On the client, the `Http2Stream` instance returned by `http2session.request()` may not be immediately ready for use if the parent `Http2Session` has not yet been fully established. In such cases, operations called on the `Http2Stream` will be buffered until the `'ready'` event is emitted. User code should rarely, if ever, need to handle the `'ready'` event directly. The ready status of an `Http2Stream` can be determined by checking the value of `http2stream.id` . If the value is `undefined` , the stream is not yet ready for use.

Destruction

All `Http2Stream` instances are destroyed either when:

- An `RST_STREAM` frame for the stream is received by the connected peer, and (for client streams only) pending data has been read.
- The `http2stream.close()` method is called, and (for client streams only) pending data has been read.
- The `http2stream.destroy()` or `http2session.destroy()` methods are called.

When an `Http2Stream` instance is destroyed, an attempt will be made to send an `RST_STREAM` frame to the connected peer.

When the `Http2Stream` instance is destroyed, the `'close'` event will be emitted. Because `Http2Stream` is an instance of `stream.Duplex` , the `'end'` event will also be emitted if the stream data is currently flowing. The `'error'` event may also be emitted if `http2stream.destroy()` was called with an `Error` passed as the first argument.

After the `Http2Stream` has been destroyed, the `http2stream.destroyed` property will be `true` and the `http2stream.rstCode` property will specify the `RST_STREAM` error code. The `Http2Stream` instance is no longer usable once destroyed.

Event: 'aborted'

The `'aborted'` event is emitted whenever a `Http2Stream` instance is abnormally aborted in mid-communication. Its listener does not expect any arguments.

The `'aborted'` event will only be emitted if the `Http2Stream` writable side has not been ended.

Event: 'close'

The `'close'` event is emitted when the `Http2Stream` is destroyed. Once this event is emitted, the `Http2Stream` instance is no longer usable.

The HTTP/2 error code used when closing the stream can be retrieved using the `http2stream.rstCode` property. If the code is any value other than `NGHTTP2_NO_ERROR` (`0`), an `'error'` event will have also been emitted.

Event: 'error'

- `error` <Error>

The `'error'` event is emitted when an error occurs during the processing of an `Http2Stream` .

Event: 'frameError'

- `type` <integer> The frame type.
- `code` <integer> The error code.
- `id` <integer> The stream id (or `0` if the frame isn't associated with a stream).

The `'frameError'` event is emitted when an error occurs while attempting to send a frame. When invoked, the handler function will receive an integer argument identifying the frame type, and an integer argument identifying the error code. The `Http2Stream` instance will be destroyed immediately after the `'frameError'` event is emitted.

Event: 'ready'

The `'ready'` event is emitted when the `Http2Stream` has been opened, has been assigned an `id` , and can be used. The listener does not expect any arguments.

Event: 'timeout'

The `'timeout'` event is emitted after no activity is received for this `Http2Stream` within the number of milliseconds set using `http2stream.setTimeout()` . Its listener does not expect any arguments.

Event: 'trailers'

- `headers` <HTTP/2 Headers Object> An object describing the headers
- `flags` <number> The associated numeric flags

The `'trailers'` event is emitted when a block of headers associated with trailing header fields is received. The listener callback is passed the HTTP/2 Headers Object and flags associated with the headers.

This event might not be emitted if `http2stream.end()` is called before trailers are received and the incoming data is not being read or listened for.

```
stream.on('trailers', (headers, flags) => {
  console.log(headers);
```

```
});
```
COPY

Event: 'wantTrailers'

The `'wantTrailers'` event is emitted when the `Http2Stream` has queued the final `DATA` frame to be sent on a frame and the `Http2Stream` is ready to send trailing headers. When initiating a request or response, the `waitForTrailers` option must be set for this event to be emitted.

http2stream.aborted

- `<boolean>`

Set to `true` if the `Http2Stream` instance was aborted abnormally. When set, the `'aborted'` event will have been emitted.

http2stream.bufferSize

- `<number>`

This property shows the number of characters currently buffered to be written. See `net.Socket.bufferSize` for details.

http2stream.close(code[, callback])

- `code` `<number>` Unsigned 32-bit integer identifying the error code. **Default:** `http2.constants.NGHTTP2_NO_ERROR` (`0x00`).
- `callback` `<Function>` An optional function registered to listen for the `'close'` event.

Closes the `Http2Stream` instance by sending an `RST_STREAM` frame to the connected HTTP/2 peer.

http2stream.closed

- `<boolean>`

Set to `true` if the `Http2Stream` instance has been closed.

http2stream.destroyed

- `<boolean>`

Set to `true` if the `Http2Stream` instance has been destroyed and is no longer usable.

http2stream.endAfterHeaders

- `<boolean>`

Set to `true` if the `END_STREAM` flag was set in the request or response HEADERS frame received, indicating that no additional data should be received and the readable side of the `Http2Stream` will be closed.

http2stream.id

- `<number>` | `<undefined>`

The numeric stream identifier of this `Http2Stream` instance. Set to `undefined` if the stream identifier has not yet been assigned.

http2stream.pending

- `<boolean>`

Set to `true` if the `Http2Stream` instance has not yet been assigned a numeric stream identifier.

http2stream.priority(options)

- `options` `<Object>`

- `exclusive` <boolean> When `true` and `parent` identifies a parent Stream, this stream is made the sole direct dependency of the parent, with all other existing dependents made a dependent of this stream. **Default:** `false`.
- `parent` <number> Specifies the numeric identifier of a stream this stream is dependent on.
- `weight` <number> Specifies the relative dependency of a stream in relation to other streams with the same `parent`. The value is a number between `1` and `256` (inclusive).
- `silent` <boolean> When `true`, changes the priority locally without sending a `PRIORITY` frame to the connected peer.

Updates the priority for this `Http2Stream` instance.

http2stream.rstCode

- <number>

Set to the `RST_STREAM` error code reported when the `Http2Stream` is destroyed after either receiving an `RST_STREAM` frame from the connected peer, calling `http2stream.close()`, or `http2stream.destroy()`. Will be `undefined` if the `Http2Stream` has not been closed.

http2stream.sentHeaders

- <HTTP/2 Headers Object>

An object containing the outbound headers sent for this `Http2Stream`.

http2stream.sentInfoHeaders

- <HTTP/2 Headers Object[]>

An array of objects containing the outbound informational (additional) headers sent for this `Http2Stream`.

http2stream.sentTrailers

- <HTTP/2 Headers Object>

An object containing the outbound trailers sent for this `HttpStream`.

http2stream.session

- <Http2Session>

A reference to the `Http2Session` instance that owns this `Http2Stream`. The value will be `undefined` after the `Http2Stream` instance is destroyed.

http2stream.setTimeout(msecs, callback)

- `msecs` <number>
- `callback` <Function>

```
import { connect, constants } from 'node:http2';
const client = connect('http://example.org:8000');
const { NGHTTP2_CANCEL } = constants;
const req = client.request({ ':path': '/' });

// Cancel the stream if there's no activity after 5 seconds
req.setTimeout(5000, () => req.close(NGHTTP2_CANCEL));
```

```
const http2 = require('node:http2');
const client = http2.connect('http://example.org:8000');
```

```
const { NGHTTP2_CANCEL } = http2.constants;
const req = client.request({ ':path': '/' });

// Cancel the stream if there's no activity after 5 seconds
req.setTimeout(5000, () => req.close(NGHTTP2_CANCEL));
```

COPY

http2stream.state

Provides miscellaneous information about the current state of the `Http2Stream`.

- <Object>
  - `localWindowSize` <number> The number of bytes the connected peer may send for this `Http2Stream` without receiving a `WINDOW_UPDATE`.
  - `state` <number> A flag indicating the low-level current state of the `Http2Stream` as determined by `nghttp2`.
  - `localClose` <number> `1` if this `Http2Stream` has been closed locally.
  - `remoteClose` <number> `1` if this `Http2Stream` has been closed remotely.
  - `sumDependencyWeight` <number> The sum weight of all `Http2Stream` instances that depend on this `Http2Stream` as specified using `PRIORITY` frames.
  - `weight` <number> The priority weight of this `Http2Stream`.

A current state of this `Http2Stream`.

http2stream.sendTrailers(headers)

- `headers` <HTTP/2 Headers Object>

Sends a trailing `HEADERS` frame to the connected HTTP/2 peer. This method will cause the `Http2Stream` to be immediately closed and must only be called after the `'wantTrailers'` event has been emitted. When sending a request or sending a response, the `options.waitForTrailers` option must be set in order to keep the `Http2Stream` open after the final `DATA` frame so that trailers can be sent.

```
import { createServer } from 'node:http2';
const server = createServer();
server.on('stream', (stream) => {
  stream.respond(undefined, { waitForTrailers: true });
  stream.on('wantTrailers', () => {
    stream.sendTrailers({ xyz: 'abc' });
  });
  stream.end('Hello World');
});
```

```
const http2 = require('node:http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respond(undefined, { waitForTrailers: true });
  stream.on('wantTrailers', () => {
    stream.sendTrailers({ xyz: 'abc' });
  });
  stream.end('Hello World');
});
```

COPY

The HTTP/1 specification forbids trailers from containing HTTP/2 pseudo-header fields (e.g. `':method'`, `':path'`, etc).

Class: ClientHttp2Stream

- Extends `<Http2Stream>`

The `ClientHttp2Stream` class is an extension of `Http2Stream` that is used exclusively on HTTP/2 Clients. `Http2Stream` instances on the client provide events such as `'response'` and `'push'` that are only relevant on the client.

Event: 'continue'

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

Event: 'headers'

- `headers` `<HTTP/2 Headers Object>`
- `flags` `<number>`

The `'headers'` event is emitted when an additional block of headers is received for a stream, such as when a block of `1xx` informational headers is received. The listener callback is passed the HTTP/2 Headers Object and flags associated with the headers.

```
stream.on('headers', (headers, flags) => {
  console.log(headers);
});
```

COPY

Event: 'push'

- `headers` `<HTTP/2 Headers Object>`
- `flags` `<number>`

The `'push'` event is emitted when response headers for a Server Push stream are received. The listener callback is passed the HTTP/2 Headers Object and flags associated with the headers.

```
stream.on('push', (headers, flags) => {
  console.log(headers);
});
```

COPY

Event: 'response'

- `headers` `<HTTP/2 Headers Object>`
- `flags` `<number>`

The `'response'` event is emitted when a response `HEADERS` frame has been received for this stream from the connected HTTP/2 server. The listener is invoked with two arguments: an `Object` containing the received HTTP/2 Headers Object, and flags associated with the headers.

```
import { connect } from 'node:http2';
const client = connect('https://localhost');
const req = client.request({ ':path': '/' });
req.on('response', (headers, flags) => {
  console.log(headers[':status']);
});
```

```
const http2 = require('node:http2');
const client = http2.connect('https://localhost');
const req = client.request({ ':path': '/' });
req.on('response', (headers, flags) => {
  console.log(headers[':status']);
});
```

COPY

Class: ServerHttp2Stream

- Extends: <Http2Stream>

The `ServerHttp2Stream` class is an extension of `Http2Stream` that is used exclusively on HTTP/2 Servers. `Http2Stream` instances on the server provide additional methods such as `http2stream.pushStream()` and `http2stream.respond()` that are only relevant on the server.

http2stream.additionalHeaders(headers)

- `headers` <HTTP/2 Headers Object>

Sends an additional informational `HEADERS` frame to the connected HTTP/2 peer.

http2stream.headersSent

- <boolean>

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

http2stream.pushAllowed

- <boolean>

Read-only property mapped to the `SETTINGS_ENABLE_PUSH` flag of the remote client's most recent `SETTINGS` frame. Will be `true` if the remote peer accepts push streams, `false` otherwise. Settings are the same for every `Http2Stream` in the same `Http2Session`.

http2stream.pushStream(headers[, options], callback)

- `headers` <HTTP/2 Headers Object>
- `options` <Object>
  - `exclusive` <boolean> When `true` and `parent` identifies a parent Stream, the created stream is made the sole direct dependency of the parent, with all other existing dependents made a dependent of the newly created stream. **Default:** `false`.
  - `parent` <number> Specifies the numeric identifier of a stream the newly created stream is dependent on.
- `callback` <Function> Callback that is called once the push stream has been initiated.
  - `err` <Error>
  - `pushStream` <ServerHttp2Stream> The returned `pushStream` object.
  - `headers` <HTTP/2 Headers Object> Headers object the `pushStream` was initiated with.

Initiates a push stream. The callback is invoked with the new `Http2Stream` instance created for the push stream passed as the second argument, or an `Error` passed as the first argument.

```
import { createServer } from 'node:http2';
const server = createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 });
  stream.pushStream({ ':path': '/' }, (err, pushStream, headers) => {
    if (err) throw err;
```

```
    pushStream.respond({ ':status': 200 });
    pushStream.end('some pushed data');
  });
  stream.end('some data');
});
```

```
const http2 = require('node:http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 });
  stream.pushStream({ ':path': '/' }, (err, pushStream, headers) => {
    if (err) throw err;
    pushStream.respond({ ':status': 200 });
    pushStream.end('some pushed data');
  });
  stream.end('some data');
});
```

COPY

Setting the weight of a push stream is not allowed in the `HEADERS` frame. Pass a `weight` value to `http2stream.priority` with the `silent` option set to `true` to enable server-side bandwidth balancing between concurrent streams.

Calling `http2stream.pushStream()` from within a pushed stream is not permitted and will throw an error.

http2stream.respond([headers[, options]])

- `headers` `<HTTP/2 Headers Object>`
- `options` `<Object>`
  - `endStream` `<boolean>` Set to `true` to indicate that the response will not include payload data.
  - `waitForTrailers` `<boolean>` When `true`, the `Http2Stream` will emit the `'wantTrailers'` event after the final `DATA` frame has been sent.

```
import { createServer } from 'node:http2';
const server = createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 });
  stream.end('some data');
});
```

```
const http2 = require('node:http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 });
  stream.end('some data');
});
```

COPY

Initiates a response. When the `options.waitForTrailers` option is set, the `'wantTrailers'` event will be emitted immediately after queuing the last chunk of payload data to be sent. The `http2stream.sendTrailers()` method can then be used to sent trailing header fields to the peer.

When `options.waitForTrailers` is set, the `Http2Stream` will not automatically close when the final `DATA` frame is transmitted. User code must call either `http2stream.sendTrailers()` or `http2stream.close()` to close the `Http2Stream`.

```
import { createServer } from 'node:http2';
const server = createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 }, { waitForTrailers: true });
  stream.on('wantTrailers', () => {
    stream.sendTrailers({ ABC: 'some value to send' });
  });
  stream.end('some data');
});
```

```
const http2 = require('node:http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respond({ ':status': 200 }, { waitForTrailers: true });
  stream.on('wantTrailers', () => {
    stream.sendTrailers({ ABC: 'some value to send' });
  });
  stream.end('some data');
});
```

COPY

http2stream.respondWithFD(fd[, headers[, options]])

- `fd` <number> | <FileHandle> A readable file descriptor.
- `headers` <HTTP/2 Headers Object>
- `options` <Object>
  - `statCheck` <Function>
  - `waitForTrailers` <boolean> When `true`, the `Http2Stream` will emit the `'wantTrailers'` event after the final `DATA` frame has been sent.
  - `offset` <number> The offset position at which to begin reading.
  - `length` <number> The amount of data from the fd to send.

Initiates a response whose data is read from the given file descriptor. No validation is performed on the given file descriptor. If an error occurs while attempting to read data using the file descriptor, the `Http2Stream` will be closed using an `RST_STREAM` frame using the standard `INTERNAL_ERROR` code.

When used, the `Http2Stream` object's `Duplex` interface will be closed automatically.

```
import { createServer } from 'node:http2';
import { openSync, fstatSync, closeSync } from 'node:fs';

const server = createServer();
server.on('stream', (stream) => {
  const fd = openSync('/some/file', 'r');

  const stat = fstatSync(fd);
  const headers = {
```

```
    'content-length': stat.size,
    'last-modified': stat.mtime.toUTCString(),
    'content-type': 'text/plain; charset=utf-8',
  };
  stream.respondWithFD(fd, headers);
  stream.on('close', () => closeSync(fd));
});
```

```
const http2 = require('node:http2');
const fs = require('node:fs');

const server = http2.createServer();
server.on('stream', (stream) => {
  const fd = fs.openSync('/some/file', 'r');

  const stat = fs.fstatSync(fd);
  const headers = {
    'content-length': stat.size,
    'last-modified': stat.mtime.toUTCString(),
    'content-type': 'text/plain; charset=utf-8',
  };
  stream.respondWithFD(fd, headers);
  stream.on('close', () => fs.closeSync(fd));
});
```

COPY

The optional `options.statCheck` function may be specified to give user code an opportunity to set additional content headers based on the `fs.Stat` details of the given fd. If the `statCheck` function is provided, the `http2stream.respondWithFD()` method will perform an `fs.fstat()` call to collect details on the provided file descriptor.

The `offset` and `length` options may be used to limit the response to a specific range subset. This can be used, for instance, to support HTTP Range requests.

The file descriptor or `FileHandle` is not closed when the stream is closed, so it will need to be closed manually once it is no longer needed. Using the same file descriptor concurrently for multiple streams is not supported and may result in data loss. Re-using a file descriptor after a stream has finished is supported.

When the `options.waitForTrailers` option is set, the `'wantTrailers'` event will be emitted immediately after queuing the last chunk of payload data to be sent. The `http2stream.sendTrailers()` method can then be used to sent trailing header fields to the peer.

When `options.waitForTrailers` is set, the `Http2Stream` will not automatically close when the final `DATA` frame is transmitted. User code *must* call either `http2stream.sendTrailers()` or `http2stream.close()` to close the `Http2Stream`.

```
import { createServer } from 'node:http2';
import { openSync, fstatSync, closeSync } from 'node:fs';

const server = createServer();
server.on('stream', (stream) => {
  const fd = openSync('/some/file', 'r');

  const stat = fstatSync(fd);
  const headers = {
```

```
    'content-length': stat.size,
    'last-modified': stat.mtime.toUTCString(),
    'content-type': 'text/plain; charset=utf-8',
  };
  stream.respondWithFD(fd, headers, { waitForTrailers: true });
  stream.on('wantTrailers', () => {
    stream.sendTrailers({ ABC: 'some value to send' });
  });

  stream.on('close', () => closeSync(fd));
});
```

```
const http2 = require('node:http2');
const fs = require('node:fs');

const server = http2.createServer();
server.on('stream', (stream) => {
  const fd = fs.openSync('/some/file', 'r');

  const stat = fs.fstatSync(fd);
  const headers = {
    'content-length': stat.size,
    'last-modified': stat.mtime.toUTCString(),
    'content-type': 'text/plain; charset=utf-8',
  };
  stream.respondWithFD(fd, headers, { waitForTrailers: true });
  stream.on('wantTrailers', () => {
    stream.sendTrailers({ ABC: 'some value to send' });
  });

  stream.on('close', () => fs.closeSync(fd));
});
```

COPY

http2stream.respondWithFile(path[, headers[, options]])

- `path` <string> | <Buffer> | <URL>
- `headers` <HTTP/2 Headers Object>
- `options` <Object>
  - `statCheck` <Function>
  - `onError` <Function> Callback function invoked in the case of an error before send.
  - `waitForTrailers` <boolean> When `true`, the `Http2Stream` will emit the `'wantTrailers'` event after the final `DATA` frame has been sent.
  - `offset` <number> The offset position at which to begin reading.
  - `length` <number> The amount of data from the fd to send.

Sends a regular file as the response. The `path` must specify a regular file or an `'error'` event will be emitted on the `Http2Stream` object.

When used, the `Http2Stream` object's `Duplex` interface will be closed automatically.

The optional `options.statCheck` function may be specified to give user code an opportunity to set additional content headers based on the `fs.Stat` details of the given file:

If an error occurs while attempting to read the file data, the `Http2Stream` will be closed using an `RST_STREAM` frame using the standard `INTERNAL_ERROR` code. If the `onError` callback is defined, then it will be called. Otherwise the stream will be destroyed.

Example using a file path:

```
import { createServer } from 'node:http2';
const server = createServer();
server.on('stream', (stream) => {
  function statCheck(stat, headers) {
    headers['last-modified'] = stat.mtime.toUTCString();
  }

  function onError(err) {
    // stream.respond() can throw if the stream has been destroyed by
    // the other side.
    try {
      if (err.code === 'ENOENT') {
        stream.respond({ ':status': 404 });
      } else {
        stream.respond({ ':status': 500 });
      }
    } catch (err) {
      // Perform actual error handling.
      console.error(err);
    }
    stream.end();
  }

  stream.respondWithFile('/some/file',
                         { 'content-type': 'text/plain; charset=utf-8' },
                         { statCheck, onError });
});
```

```
const http2 = require('node:http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  function statCheck(stat, headers) {
    headers['last-modified'] = stat.mtime.toUTCString();
  }

  function onError(err) {
    // stream.respond() can throw if the stream has been destroyed by
    // the other side.
    try {
      if (err.code === 'ENOENT') {
        stream.respond({ ':status': 404 });
      } else {
        stream.respond({ ':status': 500 });
      }
```

```
    } catch (err) {
      // Perform actual error handling.
      console.error(err);
    }
    stream.end();
  }

  stream.respondWithFile('/some/file',
                         { 'content-type': 'text/plain; charset=utf-8' },
                         { statCheck, onError });
});
```

COPY

The `options.statCheck` function may also be used to cancel the send operation by returning `false`. For instance, a conditional request may check the stat results to determine if the file has been modified to return an appropriate `304` response:

```
import { createServer } from 'node:http2';
const server = createServer();
server.on('stream', (stream) => {
  function statCheck(stat, headers) {
    // Check the stat here...
    stream.respond({ ':status': 304 });
    return false; // Cancel the send operation
  }
  stream.respondWithFile('/some/file',
                         { 'content-type': 'text/plain; charset=utf-8' },
                         { statCheck });
});
```

```
const http2 = require('node:http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  function statCheck(stat, headers) {
    // Check the stat here...
    stream.respond({ ':status': 304 });
    return false; // Cancel the send operation
  }
  stream.respondWithFile('/some/file',
                         { 'content-type': 'text/plain; charset=utf-8' },
                         { statCheck });
});
```

COPY

The `content-length` header field will be automatically set.

The `offset` and `length` options may be used to limit the response to a specific range subset. This can be used, for instance, to support HTTP Range requests.

The `options.onError` function may also be used to handle all the errors that could happen before the delivery of the file is initiated. The default behavior is to destroy the stream.

When the `options.waitForTrailers` option is set, the `'wantTrailers'` event will be emitted immediately after queuing the last chunk of payload data to be sent. The `http2stream.sendTrailers()` method can then be used to sent trailing header fields to the peer.

When `options.waitForTrailers` is set, the `Http2Stream` will not automatically close when the final `DATA` frame is transmitted. User code must call either `http2stream.sendTrailers()` or `http2stream.close()` to close the `Http2Stream`.

```
import { createServer } from 'node:http2';
const server = createServer();
server.on('stream', (stream) => {
  stream.respondWithFile('/some/file',
                         { 'content-type': 'text/plain; charset=utf-8' },
                         { waitForTrailers: true });
  stream.on('wantTrailers', () => {
    stream.sendTrailers({ ABC: 'some value to send' });
  });
});
```

```
const http2 = require('node:http2');
const server = http2.createServer();
server.on('stream', (stream) => {
  stream.respondWithFile('/some/file',
                         { 'content-type': 'text/plain; charset=utf-8' },
                         { waitForTrailers: true });
  stream.on('wantTrailers', () => {
    stream.sendTrailers({ ABC: 'some value to send' });
  });
});
```

COPY

Class: Http2Server

- Extends: <net.Server>

Instances of `Http2Server` are created using the `http2.createServer()` function. The `Http2Server` class is not exported directly by the `node:http2` module.

Event: 'checkContinue'

- `request` <http2.Http2ServerRequest>
- `response` <http2.Http2ServerResponse>

If a `'request'` listener is registered or `http2.createServer()` is supplied a callback function, the `'checkContinue'` event is 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 status `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: '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.

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.

Event: 'request'

- `request` `<http2.Http2ServerRequest>`
- `response` `<http2.Http2ServerResponse>`

Emitted each time there is a request. There may be multiple requests per session. See the Compatibility API.

Event: 'session'

- `session` `<ServerHttp2Session>`

The `'session'` event is emitted when a new `Http2Session` is created by the `Http2Server`.

Event: 'sessionError'

- `error` `<Error>`
- `session` `<ServerHttp2Session>`

The `'sessionError'` event is emitted when an `'error'` event is emitted by an `Http2Session` object associated with the `Http2Server`.

Event: 'stream'

- `stream` `<Http2Stream>` A reference to the stream
- `headers` `<HTTP/2 Headers Object>` An object describing the headers
- `flags` `<number>` The associated numeric flags
- `rawHeaders` `<Array>` An array containing the raw header names followed by their respective values.

The `'stream'` event is emitted when a `'stream'` event has been emitted by an `Http2Session` associated with the server.

See also `Http2Session`'s `'stream'` event.

```
import { createServer, constants } from 'node:http2';
const {
  HTTP2_HEADER_METHOD,
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS,
  HTTP2_HEADER_CONTENT_TYPE,
} = constants;

const server = createServer();
server.on('stream', (stream, headers, flags) => {
  const method = headers[HTTP2_HEADER_METHOD];
  const path = headers[HTTP2_HEADER_PATH];
  // ...
  stream.respond({
    [HTTP2_HEADER_STATUS]: 200,
    [HTTP2_HEADER_CONTENT_TYPE]: 'text/plain; charset=utf-8',
  });
  stream.write('hello ');
```

```
  stream.end('world');
});
```

```
const http2 = require('node:http2');
const {
  HTTP2_HEADER_METHOD,
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS,
  HTTP2_HEADER_CONTENT_TYPE,
} = http2.constants;

const server = http2.createServer();
server.on('stream', (stream, headers, flags) => {
  const method = headers[HTTP2_HEADER_METHOD];
  const path = headers[HTTP2_HEADER_PATH];
  // ...
  stream.respond({
    [HTTP2_HEADER_STATUS]: 200,
    [HTTP2_HEADER_CONTENT_TYPE]: 'text/plain; charset=utf-8',
  });
  stream.write('hello ');
  stream.end('world');
});
```

COPY

Event: 'timeout'

The `'timeout'` event is emitted when there is no activity on the Server for a given number of milliseconds set using `http2server.setTimeout()`. **Default:** 0 (no timeout)

server.close([callback])

- `callback` `<Function>`

Stops the server from establishing new sessions. This does not prevent new request streams from being created due to the persistent nature of HTTP/2 sessions. To gracefully shut down the server, call `http2session.close()` on all active sessions.

If `callback` is provided, it is not invoked until all active sessions have been closed, although the server has already stopped allowing new sessions. See `net.Server.close()` for more details.

server[Symbol.asyncDispose]()

Stability: 1 - Experimental

Calls `server.close()` and returns a promise that fulfills when the server has closed.

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

- `msecs` `<number>` **Default:** 0 (no timeout)
- `callback` `<Function>`
- Returns: `<Http2Server>`

Used to set the timeout value for http2 server requests, and sets a callback function that is called when there is no activity on the `Http2Server` after `msecs` milliseconds.

The given callback is registered as a listener on the `'timeout'` event.

In case if `callback` is not a function, a new `ERR_INVALID_ARG_TYPE` error will be thrown.

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.updateSettings([settings])

- `settings` `<HTTP/2 Settings Object>`

Used to update the server with the provided settings.

Throws `ERR_HTTP2_INVALID_SETTING_VALUE` for invalid `settings` values.

Throws `ERR_INVALID_ARG_TYPE` for invalid `settings` argument.

Class: Http2SecureServer

- Extends: `<tls.Server>`

Instances of `Http2SecureServer` are created using the `http2.createSecureServer()` function. The `Http2SecureServer` class is not exported directly by the `node:http2` module.

Event: 'checkContinue'

- `request` `<http2.Http2ServerRequest>`
- `response` `<http2.Http2ServerResponse>`

If a `'request'` listener is registered or `http2.createSecureServer()` is supplied a callback function, the `'checkContinue'` event is 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 status `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: 'connection'

- `socket` `<stream.Duplex>`

This event is emitted when a new TCP stream is established, before the TLS handshake begins. `socket` is typically an object of type `net.Socket`. Usually users will not want to access this event.

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.

Event: 'request'

- `request` <http2.Http2ServerRequest>
- `response` <http2.Http2ServerResponse>

Emitted each time there is a request. There may be multiple requests per session. See the Compatibility API .

Event: 'session'

- `session` <ServerHttp2Session>

The `'session'` event is emitted when a new `Http2Session` is created by the `Http2SecureServer` .

Event: 'sessionError'

- `error` <Error>
- `session` <ServerHttp2Session>

The `'sessionError'` event is emitted when an `'error'` event is emitted by an `Http2Session` object associated with the `Http2SecureServer` .

Event: 'stream'

- `stream` <Http2Stream> A reference to the stream
- `headers` <HTTP/2 Headers Object> An object describing the headers
- `flags` <number> The associated numeric flags
- `rawHeaders` <Array> An array containing the raw header names followed by their respective values.

The `'stream'` event is emitted when a `'stream'` event has been emitted by an `Http2Session` associated with the server.

See also `Http2Session`'s `'stream'` event .

```
import { createSecureServer, constants } from 'node:http2';
const {
  HTTP2_HEADER_METHOD,
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS,
  HTTP2_HEADER_CONTENT_TYPE,
} = constants;

const options = getOptionsSomehow();

const server = createSecureServer(options);
server.on('stream', (stream, headers, flags) => {
  const method = headers[HTTP2_HEADER_METHOD];
  const path = headers[HTTP2_HEADER_PATH];
  // ...
  stream.respond({
    [HTTP2_HEADER_STATUS]: 200,
    [HTTP2_HEADER_CONTENT_TYPE]: 'text/plain; charset=utf-8',
  });
  stream.write('hello ');
  stream.end('world');
});
```

```
const http2 = require('node:http2');
const {
```

```
  HTTP2_HEADER_METHOD,
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_STATUS,
  HTTP2_HEADER_CONTENT_TYPE,
} = http2.constants;

const options = getOptionsSomehow();

const server = http2.createSecureServer(options);
server.on('stream', (stream, headers, flags) => {
  const method = headers[HTTP2_HEADER_METHOD];
  const path = headers[HTTP2_HEADER_PATH];
  // ...
  stream.respond({
    [HTTP2_HEADER_STATUS]: 200,
    [HTTP2_HEADER_CONTENT_TYPE]: 'text/plain; charset=utf-8',
  });
  stream.write('hello ');
  stream.end('world');
});
```

COPY

Event: 'timeout'

The `'timeout'` event is emitted when there is no activity on the Server for a given number of milliseconds set using `http2secureServer.setTimeout()`. **Default:** 2 minutes.

Event: 'unknownProtocol'

- `socket` `<stream.Duplex>`

The `'unknownProtocol'` event is emitted when a connecting client fails to negotiate an allowed protocol (i.e. HTTP/2 or HTTP/1.1). The event handler receives the socket for handling. If no listener is registered for this event, the connection is terminated. A timeout may be specified using the `'unknownProtocolTimeout'` option passed to `http2.createSecureServer()`.

In earlier versions of Node.js, this event would be emitted if `allowHTTP1` is `false` and, during the TLS handshake, the client either does not send an ALPN extension or sends an ALPN extension that does not include HTTP/2 (`h2`). Newer versions of Node.js only emit this event if `allowHTTP1` is `false` and the client does not send an ALPN extension. If the client sends an ALPN extension that does not include HTTP/2 (or HTTP/1.1 if `allowHTTP1` is `true`), the TLS handshake will fail and no secure connection will be established.

See the Compatibility API.

server.close([callback])

- `callback` `<Function>`

Stops the server from establishing new sessions. This does not prevent new request streams from being created due to the persistent nature of HTTP/2 sessions. To gracefully shut down the server, call `http2session.close()` on all active sessions.

If `callback` is provided, it is not invoked until all active sessions have been closed, although the server has already stopped allowing new sessions. See `tls.Server.close()` for more details.

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

- `msecs` `<number>` **Default:** `120000` (2 minutes)
- `callback` `<Function>`

- Returns: `<Http2SecureServer>`

Used to set the timeout value for http2 secure server requests, and sets a callback function that is called when there is no activity on the `Http2SecureServer` after `msecs` milliseconds.

The given callback is registered as a listener on the `'timeout'` event.

In case if `callback` is not a function, a new `ERR_INVALID_ARG_TYPE` error will be thrown.

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.updateSettings([settings])

- `settings` `<HTTP/2 Settings Object>`

Used to update the server with the provided settings.

Throws `ERR_HTTP2_INVALID_SETTING_VALUE` for invalid `settings` values.

Throws `ERR_INVALID_ARG_TYPE` for invalid `settings` argument.

http2.createServer([options][, onRequestHandler])

- `options` `<Object>`
  - `maxDeflateDynamicTableSize` `<number>` Sets the maximum dynamic table size for deflating header fields. **Default:** `4Kib` .
  - `maxSettings` `<number>` Sets the maximum number of settings entries per `SETTINGS` frame. The minimum value allowed is `1` . **Default:** `32` .
  - `maxSessionMemory` `<number>` Sets the maximum memory that the `Http2Session` is permitted to use. The value is expressed in terms of number of megabytes, e.g. `1` equal 1 megabyte. The minimum value allowed is `1` . This is a credit based limit, existing `Http2Stream` s may cause this limit to be exceeded, but new `Http2Stream` instances will be rejected while this limit is exceeded. The current number of `Http2Stream` sessions, the current memory use of the header compression tables, current data queued to be sent, and unacknowledged `PING` and `SETTINGS` frames are all counted towards the current limit. **Default:** `10` .
  - `maxHeaderListPairs` `<number>` Sets the maximum number of header entries. This is similar to `server.maxHeadersCount` or `request.maxHeadersCount` in the `node:http` module. The minimum value is `4` . **Default:** `128` .
  - `maxOutstandingPings` `<number>` Sets the maximum number of outstanding, unacknowledged pings. **Default:** `10` .
  - `maxSendHeaderBlockLength` `<number>` Sets the maximum allowed size for a serialized, compressed block of headers. Attempts to send headers that exceed this limit will result in a `'frameError'` event being emitted and the stream being closed and destroyed. While this sets the maximum allowed size to the entire block of headers, `nghttp2` (the internal http2 library) has a limit of `65536` for each decompressed key/value pair.
  - `paddingStrategy` `<number>` The strategy used for determining the amount of padding to use for `HEADERS` and `DATA` frames. **Default:** `http2.constants.PADDING_STRATEGY_NONE` . Value may be one of:
    - `http2.constants.PADDING_STRATEGY_NONE` : No padding is applied.
    - `http2.constants.PADDING_STRATEGY_MAX` : The maximum amount of padding, determined by the internal implementation, is applied.
    - `http2.constants.PADDING_STRATEGY_ALIGNED` : Attempts to apply enough padding to ensure that the total frame length, including the 9-byte header, is a multiple of 8. For each frame, there is a maximum allowed number of padding bytes that is

determined by current flow control state and settings. If this maximum is less than the calculated amount needed to ensure alignment, the maximum is used and the total frame length is not necessarily aligned at 8 bytes.

- `peerMaxConcurrentStreams` `<number>` Sets the maximum number of concurrent streams for the remote peer as if a `SETTINGS` frame had been received. Will be overridden if the remote peer sets its own value for `maxConcurrentStreams`. **Default:** `100`.
- `maxSessionInvalidFrames` `<integer>` Sets the maximum number of invalid frames that will be tolerated before the session is closed. **Default:** `1000`.
- `maxSessionRejectedStreams` `<integer>` Sets the maximum number of rejected upon creation streams that will be tolerated before the session is closed. Each rejection is associated with an `NGHTTP2_ENHANCE_YOUR_CALM` error that should tell the peer to not open any more streams, continuing to open streams is therefore regarded as a sign of a misbehaving peer. **Default:** `100`.
- `settings` `<HTTP/2 Settings Object>` The initial settings to send to the remote peer upon connection.
- `remoteCustomSettings` `<Array>` The array of integer values determines the settings types, which are included in the `CustomSettings`-property of the received remoteSettings. Please see the `CustomSettings`-property of the `Http2Settings` object for more information, on the allowed setting types.
- `Http1IncomingMessage` `<http.IncomingMessage>` Specifies the `IncomingMessage` class to used for HTTP/1 fallback. Useful for extending the original `http.IncomingMessage`. **Default:** `http.IncomingMessage`.
- `Http1ServerResponse` `<http.ServerResponse>` Specifies the `ServerResponse` class to used for HTTP/1 fallback. Useful for extending the original `http.ServerResponse`. **Default:** `http.ServerResponse`.
- `Http2ServerRequest` `<http2.Http2ServerRequest>` Specifies the `Http2ServerRequest` class to use. Useful for extending the original `Http2ServerRequest`. **Default:** `Http2ServerRequest`.
- `Http2ServerResponse` `<http2.Http2ServerResponse>` Specifies the `Http2ServerResponse` class to use. Useful for extending the original `Http2ServerResponse`. **Default:** `Http2ServerResponse`.
- `unknownProtocolTimeout` `<number>` Specifies a timeout in milliseconds that a server should wait when an `'unknownProtocol'` is emitted. If the socket has not been destroyed by that time the server will destroy it. **Default:** `10000`.
- ...: Any `net.createServer()` option can be provided.

- `onRequestHandler` `<Function>` See Compatibility API
- Returns: `<Http2Server>`

Returns a `net.Server` instance that creates and manages `Http2Session` instances.

Since there are no browsers known that support unencrypted HTTP/2, the use of `http2.createSecureServer()` is necessary when communicating with browser clients.

```
import { createServer } from 'node:http2';

// Create an unencrypted HTTP/2 server.
// Since there are no browsers known that support
// unencrypted HTTP/2, the use of `createSecureServer()`
// is necessary when communicating with browser clients.
const server = createServer();

server.on('stream', (stream, headers) => {
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200,
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(8000);
```

```
const http2 = require('node:http2');

// Create an unencrypted HTTP/2 server.
// Since there are no browsers known that support
// unencrypted HTTP/2, the use of `http2.createSecureServer()`
// is necessary when communicating with browser clients.
const server = http2.createServer();

server.on('stream', (stream, headers) => {
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200,
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(8000);
```

COPY

http2.createSecureServer(options[, onRequestHandler])

- `options` <Object>
  - `allowHTTP1` <boolean> Incoming client connections that do not support HTTP/2 will be downgraded to HTTP/1.x when set to `true`. See the `'unknownProtocol'` event. See ALPN negotiation. **Default:** `false`.
  - `maxDeflateDynamicTableSize` <number> Sets the maximum dynamic table size for deflating header fields. **Default:** `4Kib`.
  - `maxSettings` <number> Sets the maximum number of settings entries per `SETTINGS` frame. The minimum value allowed is `1`. **Default:** `32`.
  - `maxSessionMemory` <number> Sets the maximum memory that the `Http2Session` is permitted to use. The value is expressed in terms of number of megabytes, e.g. `1` equal 1 megabyte. The minimum value allowed is `1`. This is a credit based limit, existing `Http2Stream`s may cause this limit to be exceeded, but new `Http2Stream` instances will be rejected while this limit is exceeded. The current number of `Http2Stream` sessions, the current memory use of the header compression tables, current data queued to be sent, and unacknowledged `PING` and `SETTINGS` frames are all counted towards the current limit. **Default:** `10`.
  - `maxHeaderListPairs` <number> Sets the maximum number of header entries. This is similar to `server.maxHeadersCount` or `request.maxHeadersCount` in the `node:http` module. The minimum value is `4`. **Default:** `128`.
  - `maxOutstandingPings` <number> Sets the maximum number of outstanding, unacknowledged pings. **Default:** `10`.
  - `maxSendHeaderBlockLength` <number> Sets the maximum allowed size for a serialized, compressed block of headers. Attempts to send headers that exceed this limit will result in a `'frameError'` event being emitted and the stream being closed and destroyed.
  - `paddingStrategy` <number> Strategy used for determining the amount of padding to use for `HEADERS` and `DATA` frames. **Default:** `http2.constants.PADDING_STRATEGY_NONE`. Value may be one of:
    - `http2.constants.PADDING_STRATEGY_NONE`: No padding is applied.
    - `http2.constants.PADDING_STRATEGY_MAX`: The maximum amount of padding, determined by the internal implementation, is applied.
    - `http2.constants.PADDING_STRATEGY_ALIGNED`: Attempts to apply enough padding to ensure that the total frame length, including the 9-byte header, is a multiple of 8. For each frame, there is a maximum allowed number of padding bytes that is determined by current flow control state and settings. If this maximum is less than the calculated amount needed to ensure alignment, the maximum is used and the total frame length is not necessarily aligned at 8 bytes.
  - `peerMaxConcurrentStreams` <number> Sets the maximum number of concurrent streams for the remote peer as if a `SETTINGS` frame had been received. Will be overridden if the remote peer sets its own value for `maxConcurrentStreams`. **Default:** `100`.
  - `maxSessionInvalidFrames` <integer> Sets the maximum number of invalid frames that will be tolerated before the session is closed. **Default:** `1000`.

- `maxSessionRejectedStreams` <integer> Sets the maximum number of rejected upon creation streams that will be tolerated before the session is closed. Each rejection is associated with an `NGHTTP2_ENHANCE_YOUR_CALM` error that should tell the peer to not open any more streams, continuing to open streams is therefore regarded as a sign of a misbehaving peer. **Default:** `100`.
- `settings` <HTTP/2 Settings Object> The initial settings to send to the remote peer upon connection.
- `remoteCustomSettings` <Array> The array of integer values determines the settings types, which are included in the `customSettings`-property of the received remoteSettings. Please see the `customSettings`-property of the `Http2Settings` object for more information, on the allowed setting types.
- ...: Any `tls.createServer()` options can be provided. For servers, the identity options (`pfx` or `key`/`cert`) are usually required.
- `origins` <string[]> An array of origin strings to send within an `ORIGIN` frame immediately following creation of a new server `Http2Session`.
- `unknownProtocolTimeout` <number> Specifies a timeout in milliseconds that a server should wait when an `'unknownProtocol'` event is emitted. If the socket has not been destroyed by that time the server will destroy it. **Default:** `10000`.

- `onRequestHandler` <Function> See Compatibility API
- Returns: <Http2SecureServer>

Returns a `tls.Server` instance that creates and manages `Http2Session` instances.

```
import { createSecureServer } from 'node:http2';
import { readFileSync } from 'node:fs';

const options = {
  key: readFileSync('server-key.pem'),
  cert: readFileSync('server-cert.pem'),
};

// Create a secure HTTP/2 server
const server = createSecureServer(options);

server.on('stream', (stream, headers) => {
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200,
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(8443);
```

```
const http2 = require('node:http2');
const fs = require('node:fs');

const options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem'),
};

// Create a secure HTTP/2 server
const server = http2.createSecureServer(options);

server.on('stream', (stream, headers) => {
```

```
  stream.respond({
    'content-type': 'text/html; charset=utf-8',
    ':status': 200,
  });
  stream.end('<h1>Hello World</h1>');
});

server.listen(8443);
```

COPY

http2.connect(authority[, options][, listener])

- `authority` <string> | <URL> The remote HTTP/2 server to connect to. This must be in the form of a minimal, valid URL with the `http://` or `https://` prefix, host name, and IP port (if a non-default port is used). Userinfo (user ID and password), path, querystring, and fragment details in the URL will be ignored.
- `options` <Object>
  - `maxDeflateDynamicTableSize` <number> Sets the maximum dynamic table size for deflating header fields. **Default:** `4Kib` .
  - `maxSettings` <number> Sets the maximum number of settings entries per `SETTINGS` frame. The minimum value allowed is `1` . **Default:** `32` .
  - `maxSessionMemory` <number> Sets the maximum memory that the `Http2Session` is permitted to use. The value is expressed in terms of number of megabytes, e.g. `1` equal 1 megabyte. The minimum value allowed is `1` . This is a credit based limit, existing `Http2Stream` s may cause this limit to be exceeded, but new `Http2Stream` instances will be rejected while this limit is exceeded. The current number of `Http2Stream` sessions, the current memory use of the header compression tables, current data queued to be sent, and unacknowledged `PING` and `SETTINGS` frames are all counted towards the current limit. **Default:** `10` .
  - `maxHeaderListPairs` <number> Sets the maximum number of header entries. This is similar to `server.maxHeadersCount` or `request.maxHeadersCount` in the `node:http` module. The minimum value is `1` . **Default:** `128` .
  - `maxOutstandingPings` <number> Sets the maximum number of outstanding, unacknowledged pings. **Default:** `10` .
  - `maxReservedRemoteStreams` <number> Sets the maximum number of reserved push streams the client will accept at any given time. Once the current number of currently reserved push streams exceeds reaches this limit, new push streams sent by the server will be automatically rejected. The minimum allowed value is 0. The maximum allowed value is 2^{32}-1. A negative value sets this option to the maximum allowed value. **Default:** `200` .
  - `maxSendHeaderBlockLength` <number> Sets the maximum allowed size for a serialized, compressed block of headers. Attempts to send headers that exceed this limit will result in a `'frameError'` event being emitted and the stream being closed and destroyed.
  - `paddingStrategy` <number> Strategy used for determining the amount of padding to use for `HEADERS` and `DATA` frames. **Default:** `http2.constants.PADDING_STRATEGY_NONE` . Value may be one of:
    - `http2.constants.PADDING_STRATEGY_NONE` : No padding is applied.
    - `http2.constants.PADDING_STRATEGY_MAX` : The maximum amount of padding, determined by the internal implementation, is applied.
    - `http2.constants.PADDING_STRATEGY_ALIGNED` : Attempts to apply enough padding to ensure that the total frame length, including the 9-byte header, is a multiple of 8. For each frame, there is a maximum allowed number of padding bytes that is determined by current flow control state and settings. If this maximum is less than the calculated amount needed to ensure alignment, the maximum is used and the total frame length is not necessarily aligned at 8 bytes.
  - `peerMaxConcurrentStreams` <number> Sets the maximum number of concurrent streams for the remote peer as if a `SETTINGS` frame had been received. Will be overridden if the remote peer sets its own value for `maxConcurrentStreams` . **Default:** `100` .
  - `protocol` <string> The protocol to connect with, if not set in the `authority` . Value may be either `'http:'` or `'https:'` . **Default:** `'https:'`
  - `settings` <HTTP/2 Settings Object> The initial settings to send to the remote peer upon connection.
  - `remoteCustomSettings` <Array> The array of integer values determines the settings types, which are included in the `CustomSettings` -property of the received remoteSettings. Please see the `CustomSettings` -property of the `Http2Settings` object for more information, on the allowed setting types.

- `createConnection` <Function> An optional callback that receives the `URL` instance passed to `connect` and the `options` object, and returns any `Duplex` stream that is to be used as the connection for this session.
- ...: Any `net.connect()` or `tls.connect()` options can be provided.
- `unknownProtocolTimeout` <number> Specifies a timeout in milliseconds that a server should wait when an `'unknownProtocol'` event is emitted. If the socket has not been destroyed by that time the server will destroy it. **Default:** `10000`.

- `listener` <Function> Will be registered as a one-time listener of the `'connect'` event.
- Returns: <ClientHttp2Session>

Returns a `ClientHttp2Session` instance.

```
import { connect } from 'node:http2';
const client = connect('https://localhost:1234');

/* Use the client */

client.close();
```

```
const http2 = require('node:http2');
const client = http2.connect('https://localhost:1234');

/* Use the client */

client.close();
```

COPY

http2.constants

Error codes for RST_STREAM and GOAWAY

Value	Name	Constant
`0x00`	No Error	`http2.constants.NGHTTP2_NO_ERROR`
`0x01`	Protocol Error	`http2.constants.NGHTTP2_PROTOCOL_ERROR`
`0x02`	Internal Error	`http2.constants.NGHTTP2_INTERNAL_ERROR`
`0x03`	Flow Control Error	`http2.constants.NGHTTP2_FLOW_CONTROL_ERROR`
`0x04`	Settings Timeout	`http2.constants.NGHTTP2_SETTINGS_TIMEOUT`
`0x05`	Stream Closed	`http2.constants.NGHTTP2_STREAM_CLOSED`
`0x06`	Frame Size Error	`http2.constants.NGHTTP2_FRAME_SIZE_ERROR`
`0x07`	Refused Stream	`http2.constants.NGHTTP2_REFUSED_STREAM`
`0x08`	Cancel	`http2.constants.NGHTTP2_CANCEL`
`0x09`	Compression Error	`http2.constants.NGHTTP2_COMPRESSION_ERROR`
`0x0a`	Connect Error	`http2.constants.NGHTTP2_CONNECT_ERROR`

Value	Name	Constant
`0x0b`	Enhance Your Calm	`http2.constants.NGHTTP2_ENHANCE_YOUR_CALM`
`0x0c`	Inadequate Security	`http2.constants.NGHTTP2_INADEQUATE_SECURITY`
`0x0d`	HTTP/1.1 Required	`http2.constants.NGHTTP2_HTTP_1_1_REQUIRED`

The `'timeout'` event is emitted when there is no activity on the Server for a given number of milliseconds set using `http2server.setTimeout()`.

http2.getDefaultSettings()

- Returns: `<HTTP/2 Settings Object>`

Returns an object containing the default settings for an `Http2Session` instance. This method returns a new object instance every time it is called so instances returned may be safely modified for use.

http2.getPackedSettings([settings])

- `settings` `<HTTP/2 Settings Object>`
- Returns: `<Buffer>`

Returns a `Buffer` instance containing serialized representation of the given HTTP/2 settings as specified in the HTTP/2 specification. This is intended for use with the `HTTP2-Settings` header field.

```
import { getPackedSettings } from 'node:http2';

const packed = getPackedSettings({ enablePush: false });

console.log(packed.toString('base64'));
// Prints: AAIAAAAA
```

```
const http2 = require('node:http2');

const packed = http2.getPackedSettings({ enablePush: false });

console.log(packed.toString('base64'));
// Prints: AAIAAAAA
```

COPY

http2.getUnpackedSettings(buf)

- `buf` `<Buffer>` | `<TypedArray>` The packed settings.
- Returns: `<HTTP/2 Settings Object>`

Returns a HTTP/2 Settings Object containing the deserialized settings from the given `Buffer` as generated by `http2.getPackedSettings()`.

http2.performServerHandshake(socket[, options])

- `socket` `<stream.Duplex>`
- `options` `<Object>`
  - ...: Any `http2.createServer()` option can be provided.

- Returns: `<ServerHttp2Session>`

Create an HTTP/2 server session from an existing socket.

http2.sensitiveHeaders

- `<symbol>`

This symbol can be set as a property on the HTTP/2 headers object with an array value in order to provide a list of headers considered sensitive. See Sensitive headers for more details.

Headers object

Headers are represented as own-properties on JavaScript objects. The property keys will be serialized to lower-case. Property values should be strings (if they are not they will be coerced to strings) or an `Array` of strings (in order to send more than one value per header field).

```
const headers = {
  ':status': '200',
  'content-type': 'text-plain',
  'ABC': ['has', 'more', 'than', 'one', 'value'],
};

stream.respond(headers);
```

COPY

Header objects passed to callback functions will have a `null` prototype. This means that normal JavaScript object methods such as `Object.prototype.toString()` and `Object.prototype.hasOwnProperty()` will not work.

For incoming headers:

- The `:status` header is converted to `number`.
- Duplicates of `:status`, `:method`, `:authority`, `:scheme`, `:path`, `:protocol`, `age`, `authorization`, `access-control-allow-credentials`, `access-control-max-age`, `access-control-request-method`, `content-encoding`, `content-language`, `content-length`, `content-location`, `content-md5`, `content-range`, `content-type`, `date`, `dnt`, `etag`, `expires`, `from`, `host`, `if-match`, `if-modified-since`, `if-none-match`, `if-range`, `if-unmodified-since`, `last-modified`, `location`, `max-forwards`, `proxy-authorization`, `range`, `referer`, `retry-after`, `tk`, `upgrade-insecure-requests`, `user-agent` or `x-content-type-options` 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 ', '.

```
import { createServer } from 'node:http2';
const server = createServer();
server.on('stream', (stream, headers) => {
  console.log(headers[':path']);
  console.log(headers.ABC);
});
```

```
const http2 = require('node:http2');
const server = http2.createServer();
server.on('stream', (stream, headers) => {
  console.log(headers[':path']);
  console.log(headers.ABC);
});
```

COPY

Sensitive headers

HTTP2 headers can be marked as sensitive, which means that the HTTP/2 header compression algorithm will never index them. This can make sense for header values with low entropy and that may be considered valuable to an attacker, for example `Cookie` or `Authorization`. To achieve this, add the header name to the `[http2.sensitiveHeaders]` property as an array:

```
const headers = {
  ':status': '200',
  'content-type': 'text-plain',
  'cookie': 'some-cookie',
  'other-sensitive-header': 'very secret data',
  [http2.sensitiveHeaders]: ['cookie', 'other-sensitive-header'],
};

stream.respond(headers);
```

COPY

For some headers, such as `Authorization` and short `Cookie` headers, this flag is set automatically.

This property is also set for received headers. It will contain the names of all headers marked as sensitive, including ones marked that way automatically.

Settings object

The `http2.getDefaultSettings()`, `http2.getPackedSettings()`, `http2.createServer()`, `http2.createSecureServer()`, `http2session.settings()`, `http2session.localSettings`, and `http2session.remoteSettings` APIs either return or receive as input an object that defines configuration settings for an `Http2Session` object. These objects are ordinary JavaScript objects containing the following properties.

- `headerTableSize` `<number>` Specifies the maximum number of bytes used for header compression. The minimum allowed value is 0. The maximum allowed value is 2^{32}-1. **Default:** `4096`.
- `enablePush` `<boolean>` Specifies `true` if HTTP/2 Push Streams are to be permitted on the `Http2Session` instances. **Default:** `true`.
- `initialWindowSize` `<number>` Specifies the *sender's* initial window size in bytes for stream-level flow control. The minimum allowed value is 0. The maximum allowed value is 2^{32}-1. **Default:** `65535`.
- `maxFrameSize` `<number>` Specifies the size in bytes of the largest frame payload. The minimum allowed value is 16,384. The maximum allowed value is 2^{24}-1. **Default:** `16384`.
- `maxConcurrentStreams` `<number>` Specifies the maximum number of concurrent streams permitted on an `Http2Session`. There is no default value which implies, at least theoretically, 2^{32}-1 streams may be open concurrently at any given time in an `Http2Session`. The minimum value is 0. The maximum allowed value is 2^{32}-1. **Default:** `4294967295`.
- `maxHeaderListSize` `<number>` Specifies the maximum size (uncompressed octets) of header list that will be accepted. The minimum allowed value is 0. The maximum allowed value is 2^{32}-1. **Default:** `65535`.
- `maxHeaderSize` `<number>` Alias for `maxHeaderListSize`.
- `enableConnectProtocol` `<boolean>` Specifies `true` if the "Extended Connect Protocol" defined by RFC 8441 is to be enabled. This setting is only meaningful if sent by the server. Once the `enableConnectProtocol` setting has been enabled for a given `Http2Session`, it cannot be disabled. **Default:** `false`.
- `customSettings` `<Object>` Specifies additional settings, yet not implemented in node and the underlying libraries. The key of the object defines the numeric value of the settings type (as defined in the "HTTP/2 SETTINGS" registry established by [RFC 7540]) and the values the actual numeric value of the settings. The settings type has to be an integer in the range from 1 to 2^16-1. It should not be a settings type already handled by node, i.e. currently it should be greater than 6, although it is not an error. The values need to be unsigned integers

in the range from 0 to 2^32-1. Currently, a maximum of up 10 custom settings is supported. It is only supported for sending SETTINGS, or for receiving settings values specified in the `remoteCustomSettings` options of the server or client object. Do not mix the `customSettings` -mechanism for a settings id with interfaces for the natively handled settings, in case a setting becomes natively supported in a future node version.

All additional properties on the settings object are ignored.

Error handling

There are several types of error conditions that may arise when using the `node:http2` module:

Validation errors occur when an incorrect argument, option, or setting value is passed in. These will always be reported by a synchronous `throw` .

State errors occur when an action is attempted at an incorrect time (for instance, attempting to send data on a stream after it has closed). These will be reported using either a synchronous `throw` or via an `'error'` event on the `Http2Stream` , `Http2Session` or HTTP/2 Server objects, depending on where and when the error occurs.

Internal errors occur when an HTTP/2 session fails unexpectedly. These will be reported via an `'error'` event on the `Http2Session` or HTTP/2 Server objects.

Protocol errors occur when various HTTP/2 protocol constraints are violated. These will be reported using either a synchronous `throw` or via an `'error'` event on the `Http2Stream` , `Http2Session` or HTTP/2 Server objects, depending on where and when the error occurs.

Invalid character handling in header names and values

The HTTP/2 implementation applies stricter handling of invalid characters in HTTP header names and values than the HTTP/1 implementation.

Header field names are *case-insensitive* and are transmitted over the wire strictly as lower-case strings. The API provided by Node.js allows header names to be set as mixed-case strings (e.g. `Content-Type`) but will convert those to lower-case (e.g. `content-type`) upon transmission.

Header field-names *must only* contain one or more of the following ASCII characters: `a` - `z` , `A` - `Z` , `0` - `9` , `!` , `#` , `$` , `%` , `&` , `'` , `*` , `+` , `-` , `.` , `^` , `_` , `` ` `` (backtick), `|` , and `~` .

Using invalid characters within an HTTP header field name will cause the stream to be closed with a protocol error being reported.

Header field values are handled with more leniency but *should* not contain new-line or carriage return characters and *should* be limited to US-ASCII characters, per the requirements of the HTTP specification.

Push streams on the client

To receive pushed streams on the client, set a listener for the `'stream'` event on the `ClientHttp2Session` :

```
import { connect } from 'node:http2';

const client = connect('http://localhost');

client.on('stream', (pushedStream, requestHeaders) => {
  pushedStream.on('push', (responseHeaders) => {
    // Process response headers
  });
  pushedStream.on('data', (chunk) => { /* handle pushed data */ });
});

const req = client.request({ ':path': '/' });
```

```
const http2 = require('node:http2');

const client = http2.connect('http://localhost');

client.on('stream', (pushedStream, requestHeaders) => {
  pushedStream.on('push', (responseHeaders) => {
    // Process response headers
  });
  pushedStream.on('data', (chunk) => { /* handle pushed data */ });
});

const req = client.request({ ':path': '/' });
```

COPY

Supporting the `CONNECT` method

The `CONNECT` method is used to allow an HTTP/2 server to be used as a proxy for TCP/IP connections.

A simple TCP Server:

```
import { createServer } from 'node:net';

const server = createServer((socket) => {
  let name = '';
  socket.setEncoding('utf8');
  socket.on('data', (chunk) => name += chunk);
  socket.on('end', () => socket.end(`hello ${name}`));
});

server.listen(8000);
```

```
const net = require('node:net');

const server = net.createServer((socket) => {
  let name = '';
  socket.setEncoding('utf8');
  socket.on('data', (chunk) => name += chunk);
  socket.on('end', () => socket.end(`hello ${name}`));
});

server.listen(8000);
```

COPY

An HTTP/2 CONNECT proxy:

```
import { createServer, constants } from 'node:http2';
const { NGHTTP2_REFUSED_STREAM, NGHTTP2_CONNECT_ERROR } = constants;
import { connect } from 'node:net';

const proxy = createServer();
proxy.on('stream', (stream, headers) => {
```

```
  if (headers[':method'] !== 'CONNECT') {
    // Only accept CONNECT requests
    stream.close(NGHTTP2_REFUSED_STREAM);
    return;
  }
  const auth = new URL(`tcp://${headers[':authority']}`);
  // It's a very good idea to verify that hostname and port are
  // things this proxy should be connecting to.
  const socket = connect(auth.port, auth.hostname, () => {
    stream.respond();
    socket.pipe(stream);
    stream.pipe(socket);
  });
  socket.on('error', (error) => {
    stream.close(NGHTTP2_CONNECT_ERROR);
  });
});

proxy.listen(8001);
```

```
const http2 = require('node:http2');
const { NGHTTP2_REFUSED_STREAM } = http2.constants;
const net = require('node:net');

const proxy = http2.createServer();
proxy.on('stream', (stream, headers) => {
  if (headers[':method'] !== 'CONNECT') {
    // Only accept CONNECT requests
    stream.close(NGHTTP2_REFUSED_STREAM);
    return;
  }
  const auth = new URL(`tcp://${headers[':authority']}`);
  // It's a very good idea to verify that hostname and port are
  // things this proxy should be connecting to.
  const socket = net.connect(auth.port, auth.hostname, () => {
    stream.respond();
    socket.pipe(stream);
    stream.pipe(socket);
  });
  socket.on('error', (error) => {
    stream.close(http2.constants.NGHTTP2_CONNECT_ERROR);
  });
});

proxy.listen(8001);
```

COPY

An HTTP/2 CONNECT client:

```
import { connect, constants } from 'node:http2';

const client = connect('http://localhost:8001');
```

```
// Must not specify the ':path' and ':scheme' headers
// for CONNECT requests or an error will be thrown.
const req = client.request({
  ':method': 'CONNECT',
  ':authority': 'localhost:8000',
});

req.on('response', (headers) => {
  console.log(headers[constants.HTTP2_HEADER_STATUS]);
});
let data = '';
req.setEncoding('utf8');
req.on('data', (chunk) => data += chunk);
req.on('end', () => {
  console.log(`The server says: ${data}`);
  client.close();
});
req.end('Jane');
```

```
const http2 = require('node:http2');

const client = http2.connect('http://localhost:8001');

// Must not specify the ':path' and ':scheme' headers
// for CONNECT requests or an error will be thrown.
const req = client.request({
  ':method': 'CONNECT',
  ':authority': 'localhost:8000',
});

req.on('response', (headers) => {
  console.log(headers[http2.constants.HTTP2_HEADER_STATUS]);
});
let data = '';
req.setEncoding('utf8');
req.on('data', (chunk) => data += chunk);
req.on('end', () => {
  console.log(`The server says: ${data}`);
  client.close();
});
req.end('Jane');
```

COPY

The extended `CONNECT` protocol

RFC 8441 defines an "Extended CONNECT Protocol" extension to HTTP/2 that may be used to bootstrap the use of an `Http2Stream` using the `CONNECT` method as a tunnel for other communication protocols (such as WebSockets).

The use of the Extended CONNECT Protocol is enabled by HTTP/2 servers by using the `enableConnectProtocol` setting:

```mjs
import { createServer } from 'node:http2';
const settings = { enableConnectProtocol: true };
const server = createServer({ settings });
```

```cjs
const http2 = require('node:http2');
const settings = { enableConnectProtocol: true };
const server = http2.createServer({ settings });
```

COPY

Once the client receives the `SETTINGS` frame from the server indicating that the extended CONNECT may be used, it may send `CONNECT` requests that use the `':protocol'` HTTP/2 pseudo-header:

```mjs
import { connect } from 'node:http2';
const client = connect('http://localhost:8080');
client.on('remoteSettings', (settings) => {
  if (settings.enableConnectProtocol) {
    const req = client.request({ ':method': 'CONNECT', ':protocol': 'foo' });
    // ...
  }
});
```

```cjs
const http2 = require('node:http2');
const client = http2.connect('http://localhost:8080');
client.on('remoteSettings', (settings) => {
  if (settings.enableConnectProtocol) {
    const req = client.request({ ':method': 'CONNECT', ':protocol': 'foo' });
    // ...
  }
});
```

COPY

Compatibility API

The Compatibility API has the goal of providing a similar developer experience of HTTP/1 when using HTTP/2, making it possible to develop applications that support both HTTP/1 and HTTP/2. This API targets only the **public API** of the HTTP/1 . However many modules use internal methods or state, and those *are not supported* as it is a completely different implementation.

The following example creates an HTTP/2 server using the compatibility API:

```mjs
import { createServer } from 'node:http2';
const server = createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
  res.end('ok');
});
```

```
const http2 = require('node:http2');
const server = http2.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
  res.end('ok');
});
```

COPY

In order to create a mixed HTTPS and HTTP/2 server, refer to the ALPN negotiation section. Upgrading from non-tls HTTP/1 servers is not supported.

The HTTP/2 compatibility API is composed of `Http2ServerRequest` and `Http2ServerResponse`. They aim at API compatibility with HTTP/1, but they do not hide the differences between the protocols. As an example, the status message for HTTP codes is ignored.

ALPN negotiation

ALPN negotiation allows supporting both HTTPS and HTTP/2 over the same socket. The `req` and `res` objects can be either HTTP/1 or HTTP/2, and an application **must** restrict itself to the public API of HTTP/1, and detect if it is possible to use the more advanced features of HTTP/2.

The following example creates a server that supports both protocols:

```
import { createSecureServer } from 'node:http2';
import { readFileSync } from 'node:fs';

const cert = readFileSync('./cert.pem');
const key = readFileSync('./key.pem');

const server = createSecureServer(
  { cert, key, allowHTTP1: true },
  onRequest,
).listen(8000);

function onRequest(req, res) {
  // Detects if it is a HTTPS request or HTTP/2
  const { socket: { alpnProtocol } } = req.httpVersion === '2.0' ?
    req.stream.session : req;
  res.writeHead(200, { 'content-type': 'application/json' });
  res.end(JSON.stringify({
    alpnProtocol,
    httpVersion: req.httpVersion,
  }));
}
```

```
const { createSecureServer } = require('node:http2');
const { readFileSync } = require('node:fs');

const cert = readFileSync('./cert.pem');
const key = readFileSync('./key.pem');

const server = createSecureServer(
```

```
  { cert, key, allowHTTP1: true },
  onRequest,
).listen(4443);

function onRequest(req, res) {
  // Detects if it is a HTTPS request or HTTP/2
  const { socket: { alpnProtocol } } = req.httpVersion === '2.0' ?
    req.stream.session : req;
  res.writeHead(200, { 'content-type': 'application/json' });
  res.end(JSON.stringify({
    alpnProtocol,
    httpVersion: req.httpVersion,
  }));
}
```

COPY

The `'request'` event works identically on both HTTPS and HTTP/2.

Class: http2.Http2ServerRequest

- Extends: `<stream.Readable>`

A `Http2ServerRequest` object is created by `http2.Server` or `http2.SecureServer` and passed as the first argument to the `'request'` event. It may be used to access a request status, headers, and data.

Event: 'aborted'

The `'aborted'` event is emitted whenever a `Http2ServerRequest` instance is abnormally aborted in mid-communication.

The `'aborted'` event will only be emitted if the `Http2ServerRequest` writable side has not been ended.

Event: 'close'

Indicates that the underlying `Http2Stream` was closed. Just like `'end'`, this event occurs only once per response.

request.aborted

- `<boolean>`

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

request.authority

- `<string>`

The request authority pseudo header field. Because HTTP/2 allows requests to set either `:authority` or `host`, this value is derived from `req.headers[':authority']` if present. Otherwise, it is derived from `req.headers['host']`.

request.complete

- `<boolean>`

The `request.complete` property will be `true` if the request has been completed, aborted, or destroyed.

request.connection

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

- `<net.Socket>` | `<tls.TLSSocket>`

See `request.socket`.

request.destroy([error])

- `error` `<Error>`

Calls `destroy()` on the `Http2Stream` that received the `Http2ServerRequest`. If `error` is provided, an `'error'` event is emitted and `error` is passed as an argument to any listeners on the event.

It does nothing if the stream was already destroyed.

request.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.headers);
```

COPY

See HTTP/2 Headers Object.

In HTTP/2, the request path, host name, protocol, and method are represented as special headers prefixed with the `:` character (e.g. `':path'`). These special headers will be included in the `request.headers` object. Care must be taken not to inadvertently modify these special headers or errors may occur. For instance, removing all headers from the request will cause errors to occur:

```
removeAllHeaders(request.headers);
assert(request.url);   // Fails because the :path header has been removed
```

COPY

request.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. Returns `'2.0'`.

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

request.method

- `<string>`

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

request.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);
```

COPY

request.rawTrailers

- <string[]>

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

request.scheme

- <string>

The request scheme pseudo header field indicating the scheme portion of the target URL.

request.setTimeout(msecs, callback)

- `msecs` <number>
- `callback` <Function>
- Returns: <http2.Http2ServerRequest>

Sets the `Http2Stream` '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 `Http2Stream` s 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.

request.socket

- <net.Socket> | <tls.TLSSocket>

Returns a `Proxy` object that acts as a `net.Socket` (or `tls.TLSSocket`) but applies getters, setters, and methods based on HTTP/2 logic.

`destroyed` , `readable` , and `writable` properties will be retrieved from and set on `request.stream` .

`destroy` , `emit` , `end` , `on` and `once` methods will be called on `request.stream` .

`setTimeout` method will be called on `request.stream.session` .

`pause`, `read`, `resume`, and `write` will throw an error with code `ERR_HTTP2_NO_SOCKET_MANIPULATION`. See `Http2Session and Sockets` for more information.

All other interactions will be routed directly to the socket. With TLS support, use `request.socket.getPeerCertificate()` to obtain the client's authentication details.

request.stream

- `<Http2Stream>`

The `Http2Stream` object backing the request.

request.trailers

- `<Object>`

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

request.url

- `<string>`

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
Accept: text/plain
```

COPY

Then `request.url` will be:

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

COPY

To parse the url into its parts, `new URL()` can be used:

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

COPY

Class: http2.Http2ServerResponse

- 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 underlying `Http2Stream` was terminated before `response.end()` was called or able to flush.

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 HTTP/2 multiplexing 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)

- `headers` `<Object>`

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

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

response.appendHeader(name, value)

- `name` `<string>`
- `value` `<string>` | `<string[]>`

Append a single header value to the header object.

If the value is an array, this is equivalent to calling this method multiple times.

If there were no previous values for the header, this is equivalent to calling `response.setHeader()`.

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

```
// Returns headers including "set-cookie: a" and "set-cookie: b"
const server = http2.createServer((req, res) => {
  res.setHeader('set-cookie', 'a');
  res.appendHeader('set-cookie', 'b');
  res.writeHead(200);
  res.end('ok');
});
```

COPY

response.connection

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

- `<net.Socket>` | `<tls.TLSSocket>`

See `response.socket`.

response.createPushResponse(headers, callback)

- `headers` `<HTTP/2 Headers Object>` An object describing the headers
- `callback` `<Function>` Called once `http2stream.pushStream()` is finished, or either when the attempt to create the pushed `Http2Stream` has failed or has been rejected, or the state of `Http2ServerRequest` is closed prior to calling the `http2stream.pushStream()` method
  - `err` `<Error>`

- `res` <http2.Http2ServerResponse> The newly-created `Http2ServerResponse` object

Call `http2stream.pushStream()` with the given headers, and wrap the given `Http2Stream` on a newly created `Http2ServerResponse` as the callback parameter if successful. When `Http2ServerRequest` is closed, the callback is called with an error `ERR_HTTP2_INVALID_STREAM`.

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

- `data` <string> | <Buffer> | <Uint8Array>
- `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 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

Stability: 0 - Deprecated. Use `response.writableEnded`.

- <boolean>

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

response.getHeader(name)

- `name` <string>
- Returns: <string>

Reads out a header that has already been queued but not sent to the client. The name is case-insensitive.

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

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']
```
COPY

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'] }
```

COPY

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');
```

COPY

response.headersSent

- <boolean>

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

response.removeHeader(name)

- `name` <string>

Removes a header that has been queued for implicit sending.

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

COPY

response.req

- <http2.Http2ServerRequest>

A reference to the original HTTP2 `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` <string> | <string[]>

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.

```
response.setHeader('Content-Type', 'text/html; charset=utf-8');
```
COPY

or

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

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 = http2.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html; charset=utf-8');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
  res.end('ok');
});
```
COPY

response.setTimeout(msecs[, callback])

- `msecs` `<number>`
- `callback` `<Function>`
- Returns: `<http2.Http2ServerResponse>`

Sets the `Http2Stream` '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 `Http2Stream` s 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

- `<net.Socket>` | `<tls.TLSSocket>`

Returns a `Proxy` object that acts as a `net.Socket` (or `tls.TLSSocket`) but applies getters, setters, and methods based on HTTP/2 logic.

`destroyed`, `readable`, and `writable` properties will be retrieved from and set on `response.stream`.

`destroy`, `emit`, `end`, `on` and `once` methods will be called on `response.stream`.

`setTimeout` method will be called on `response.stream.session`.

`pause`, `read`, `resume`, and `write` will throw an error with code `ERR_HTTP2_NO_SOCKET_MANIPULATION`. See `Http2Session and Sockets` for more information.

All other interactions will be routed directly to the socket.

```
import { createServer } from 'node:http2';
const server = createServer((req, res) => {
  const ip = req.socket.remoteAddress;
  const port = req.socket.remotePort;
```

```
  res.end(`Your IP address is ${ip} and your source port is ${port}.`);
}).listen(3000);
```

```
const http2 = require('node:http2');
const server = http2.createServer((req, res) => {
  const ip = req.socket.remoteAddress;
  const port = req.socket.remotePort;
  res.end(`Your IP address is ${ip} and your source port is ${port}.`);
}).listen(3000);
```

COPY

response.statusCode

- <number>

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;
```

COPY

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

response.statusMessage

- <string>

Status message is not supported by HTTP/2 (RFC 7540 8.1.2.4). It returns an empty string.

response.stream

- <Http2Stream>

The Http2Stream object backing the response.

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 writable.writableFinished instead.

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

- `chunk` <string> | <Buffer> | <Uint8Array>
- `encoding` <string>
- `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 `node: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. By default the `encoding` is `'utf8'`. `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 status `100 Continue` to the client, indicating that the request body should be sent. See the `'checkContinue'` event on `Http2Server` and `Http2SecureServer`.

response.writeEarlyHints(hints)

- `hints` `<Object>`

Sends a status `103 Early Hints` to the client with a Link header, indicating that the user agent can preload/preconnect the linked resources. The `hints` is an object containing the values of headers to be sent with early hints message.

Example

```
const earlyHintsLink = '</styles.css>; rel=preload; as=style';
response.writeEarlyHints({
  'link': earlyHintsLink,
});

const earlyHintsLinks = [
  '</styles.css>; rel=preload; as=style',
  '</scripts.js>; rel=preload; as=script',
];
response.writeEarlyHints({
  'link': earlyHintsLinks,
});
```

COPY

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

- `statusCode` `<number>`
- `statusMessage` `<string>`
- `headers` `<Object>` | `<Array>`
- Returns: `<http2.Http2ServerResponse>`

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.

Returns a reference to the `Http2ServerResponse`, so that calls can be chained.

For compatibility with HTTP/1, a human-readable `statusMessage` may be passed as the second argument. However, because the `statusMessage` has no meaning within HTTP/2, the argument will have no effect and a process warning will be emitted.

```
const body = 'hello world';
response.writeHead(200, {
  'Content-Length': Buffer.byteLength(body),
  'Content-Type': 'text/plain; charset=utf-8',
});
```
COPY

`Content-Length` is given in bytes not characters. The `Buffer.byteLength()` API may be used to determine the number of bytes in a given encoding. On outbound messages, Node.js does not check if Content-Length and the length of the body being transmitted are equal or not. However, when receiving messages, Node.js will automatically reject messages when the `Content-Length` does not match the actual payload size.

This method may be called at most one time on a message 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.

```
// Returns content-type = text/plain
const server = http2.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html; charset=utf-8');
  res.setHeader('X-Foo', 'bar');
  res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
  res.end('ok');
});
```
COPY

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

Collecting HTTP/2 performance metrics

The Performance Observer API can be used to collect basic performance metrics for each `Http2Session` and `Http2Stream` instance.

```
import { PerformanceObserver } from 'node:perf_hooks';

const obs = new PerformanceObserver((items) => {
  const entry = items.getEntries()[0];
  console.log(entry.entryType);  // prints 'http2'
  if (entry.name === 'Http2Session') {
    // Entry contains statistics about the Http2Session
  } else if (entry.name === 'Http2Stream') {
    // Entry contains statistics about the Http2Stream
  }
});
obs.observe({ entryTypes: ['http2'] });
```

```
const { PerformanceObserver } = require('node:perf_hooks');

const obs = new PerformanceObserver((items) => {
  const entry = items.getEntries()[0];
  console.log(entry.entryType);  // prints 'http2'
  if (entry.name === 'Http2Session') {
```

```
    // Entry contains statistics about the Http2Session
  } else if (entry.name === 'Http2Stream') {
    // Entry contains statistics about the Http2Stream
  }
});
obs.observe({ entryTypes: ['http2'] });
```

COPY

The `entryType` property of the `PerformanceEntry` will be equal to `'http2'`.

The `name` property of the `PerformanceEntry` will be equal to either `'Http2Stream'` or `'Http2Session'`.

If `name` is equal to `Http2Stream`, the `PerformanceEntry` will contain the following additional properties:

- `bytesRead` <number> The number of `DATA` frame bytes received for this `Http2Stream`.
- `bytesWritten` <number> The number of `DATA` frame bytes sent for this `Http2Stream`.
- `id` <number> The identifier of the associated `Http2Stream`
- `timeToFirstByte` <number> The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and the reception of the first `DATA` frame.
- `timeToFirstByteSent` <number> The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and sending of the first `DATA` frame.
- `timeToFirstHeader` <number> The number of milliseconds elapsed between the `PerformanceEntry` `startTime` and the reception of the first header.

If `name` is equal to `Http2Session`, the `PerformanceEntry` will contain the following additional properties:

- `bytesRead` <number> The number of bytes received for this `Http2Session`.
- `bytesWritten` <number> The number of bytes sent for this `Http2Session`.
- `framesReceived` <number> The number of HTTP/2 frames received by the `Http2Session`.
- `framesSent` <number> The number of HTTP/2 frames sent by the `Http2Session`.
- `maxConcurrentStreams` <number> The maximum number of streams concurrently open during the lifetime of the `Http2Session`.
- `pingRTT` <number> The number of milliseconds elapsed since the transmission of a `PING` frame and the reception of its acknowledgment. Only present if a `PING` frame has been sent on the `Http2Session`.
- `streamAverageDuration` <number> The average duration (in milliseconds) for all `Http2Stream` instances.
- `streamCount` <number> The number of `Http2Stream` instances processed by the `Http2Session`.
- `type` <string> Either `'server'` or `'client'` to identify the type of `Http2Session`.

Note on `:authority` and `host`

HTTP/2 requires requests to have either the `:authority` pseudo-header or the `host` header. Prefer `:authority` when constructing an HTTP/2 request directly, and `host` when converting from HTTP/1 (in proxies, for instance).

The compatibility API falls back to `host` if `:authority` is not present. See `request.authority` for more information. However, if you don't use the compatibility API (or use `req.headers` directly), you need to implement any fall-back behavior yourself.