Handling HTTP and WebSocket Protocol

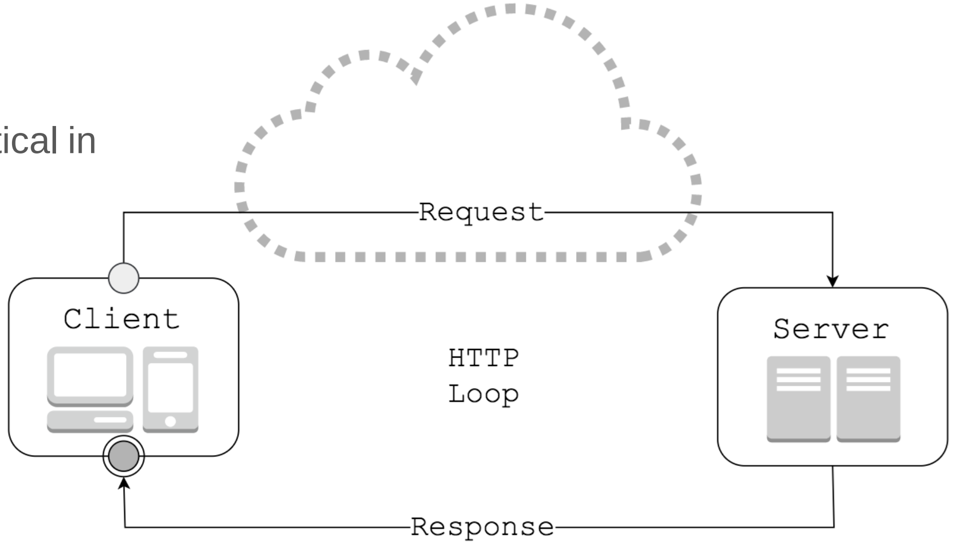
Mykhailo Miroshnikov, Oct 2017

Agenda

- Hypertext Transfer Protocol (HTTP) Overview
- `require(‘http’)`
    - Server
    - Request
    - Response
    - Static Server
    - Error handling
    - HTTP Client
- HTTPS Overview
- HTTP/2 Overview
- WebSocket Protocol
    - Socket.io
    - WebSocket

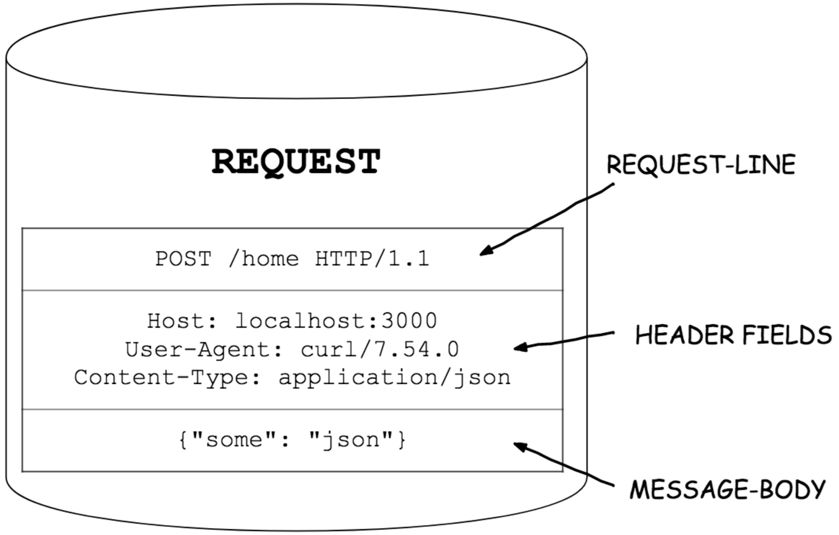
HTTP - Hypertext Transfer Protocol

- `Client` - requests via `Request`
- `Server` - responds via `Response`
- `Request` and `Response` are almost identical in terms of structure
- `Client` initiates communication



HTTP Request

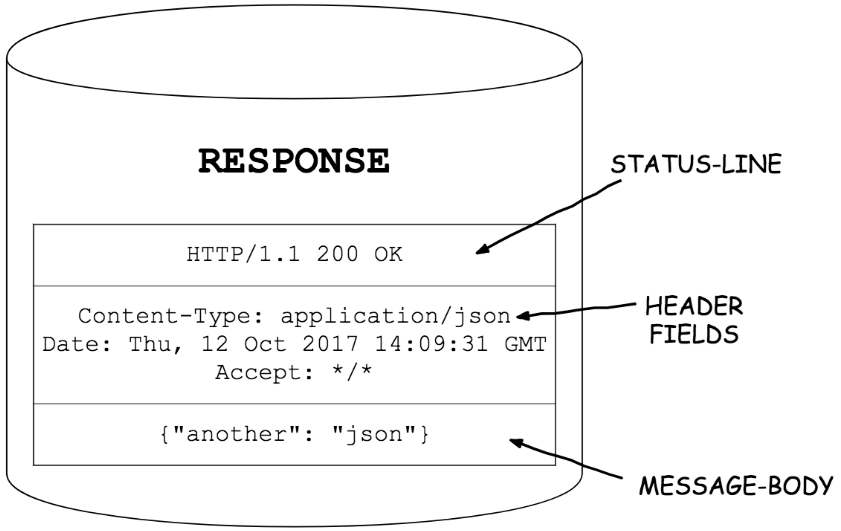
- Request-line
  - HTTP Method
  - URI
  - Protocol Version
- Header Fields
- Message-body



```
POST /home HTTP/1.1

Host: localhost:3000
User-Agent: curl/7.54.0
Content-Type: application/json

{"some": "json"}
```

HTTP Response

- Status-line
  - Protocol Version
  - Status Code
  - Reason Phrase
- Header Fields
- Message-body

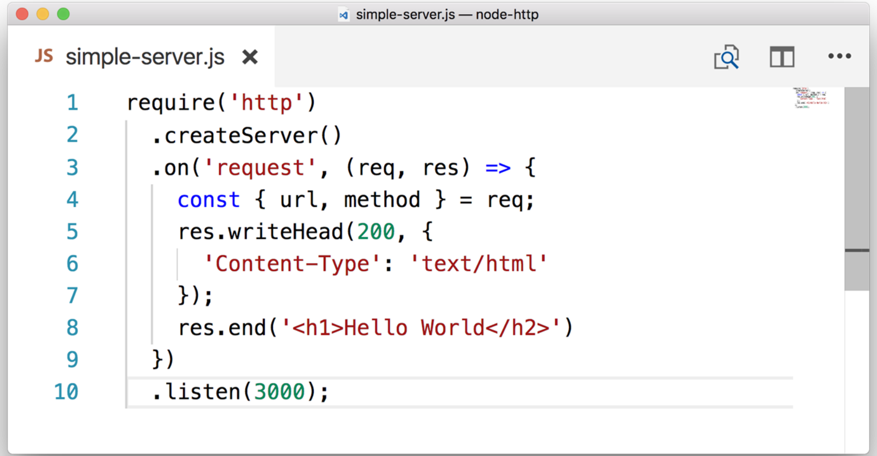


```
HTTP/1.1 200 OK

Content-Type: application/json
Date: Thu, 12 Oct 2017 14:09:31 GMT
Accept: */*

{"another": "json"}
```

HTTP/HTTPS in Node

```
require('http')
```

Agent
ClientRequest
Server
createServer()
ServerResponse
IncomingMessage
METHODS
STATUS_CODES
get()
globalAgent
request()



```js
require('http')
  .createServer()
  .on('request', (req, res) => {
    const { url, method } = req;
    res.writeHead(200, {
      'Content-Type': 'text/html'
    });
    res.end('<h1>Hello World</h2>')
  })
  .listen(3000);
```

`http.Server`
`http.createServer()`

The cornerstone feature of Node.

Minimum setup:

1. Create an instance of `Server`
2. Attach request handler to it
3. Make it listen on port

`http.Server`

Is good for:

- Fast file uploading
- WebSocket server
- Data streaming
- Ad servers
- Stock exchange software

May not be so good for:

- Static server
- CPU-heavy operations

But really, it still can be done.

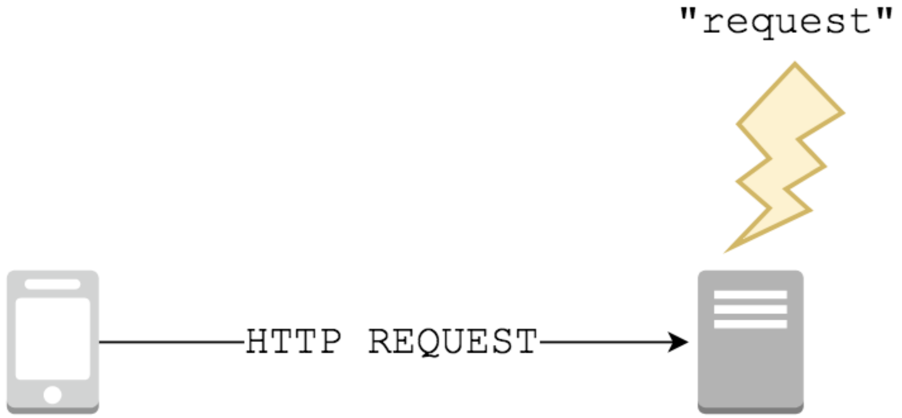
`http.Server` as `EventEmitter`

`“request”`

Emitted each time there is a request.

There can be multiple requests done over a single connection in case of Keep-Alive HTTP connection.

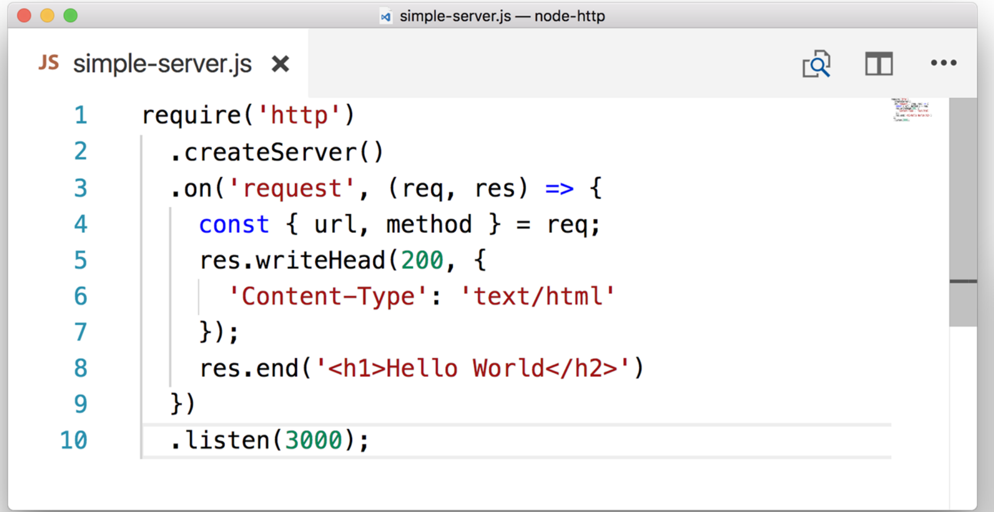
```
server.on(“request”, cb);
```



`http.Server` as `EventEmitter`

Event	**Emitted when...**	**Useful when...**
`“checkContinue”`	`Expect: 100-continue` header is received	Need to validate before receiving the body
`“checkExpectation”`	`Expect` header is received (but not `100-continue`)	…never?
`“clientError”`	Client connection emits `error` event	Need to override default `400 Bad Request` response
`“close”`	Server closes	Need to clean up
`“connect”`	Client requests `CONNECT` method	Need to handle `CONNECT` method, otherwise will close connection
`“connection”`	TCP stream is established	…never
`“upgrade”`	`Upgrade` header is received	Need to switch protocols

Hello World



simple-server.js — node-http

```javascript
require('http')
  .createServer()
  .on('request', (req, res) => {
    const { url, method } = req;
    res.writeHead(200, {
      'Content-Type': 'text/html'
    });
    res.end('<h1>Hello World</h2>')
  })
  .listen(3000);
```

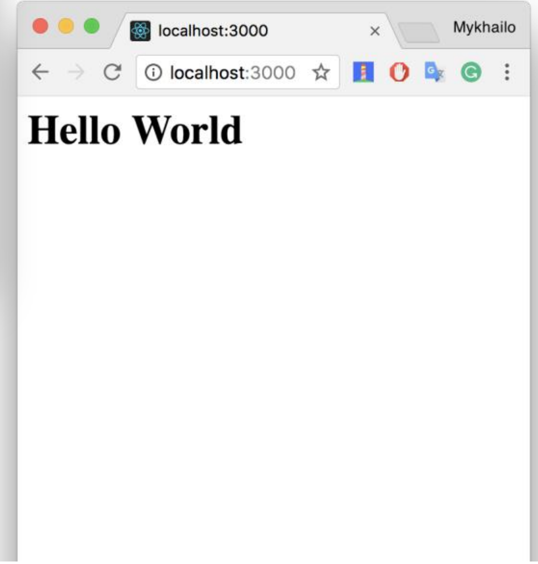
```
new http.Server([requestHandler])
```

OR

```
http.createServer([requestHandler]);
```

OR

```
server.on('request', requestHandler);
```

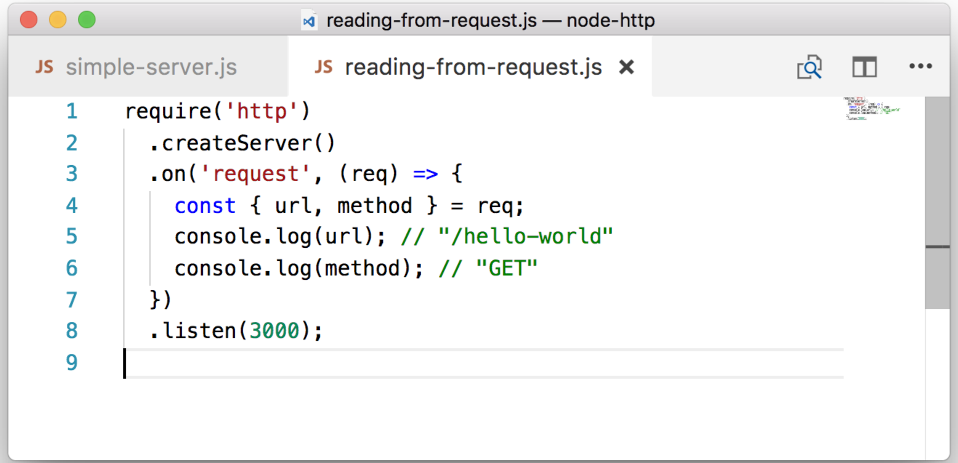


localhost:3000

Hello World

Reading from Request

Reading `url` and `method`

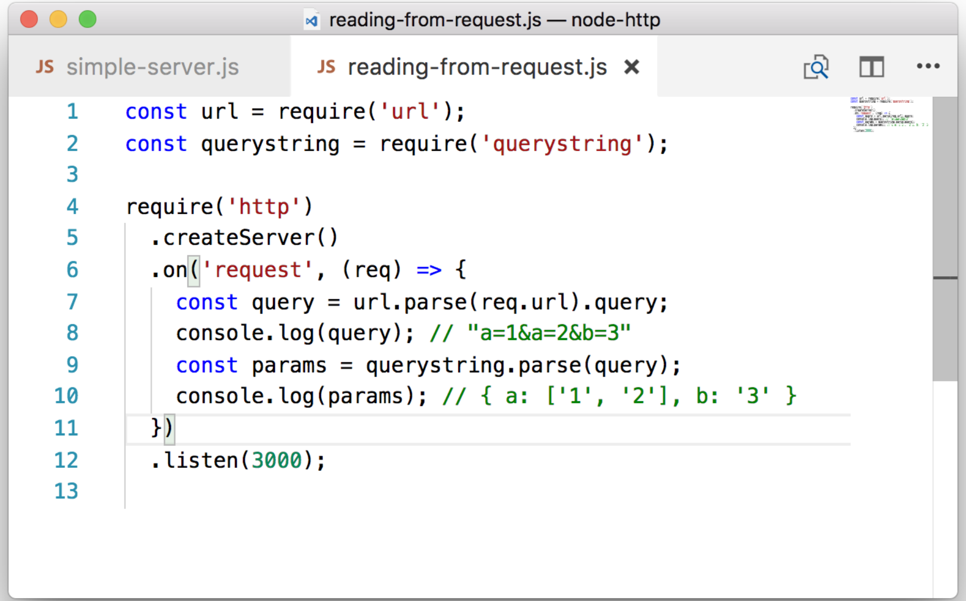


```
require('http')
  .createServer()
  .on('request', (req) => {
    const { url, method } = req;
    console.log(url); // "/hello-world"
    console.log(method); // "GET"
  })
  .listen(3000);
```

```
curl localhost:3000/hello-world
```

Reading from Request

Use `require('querystring')` and `require('url')` to parse query string params.



reading-from-request.js — node-http

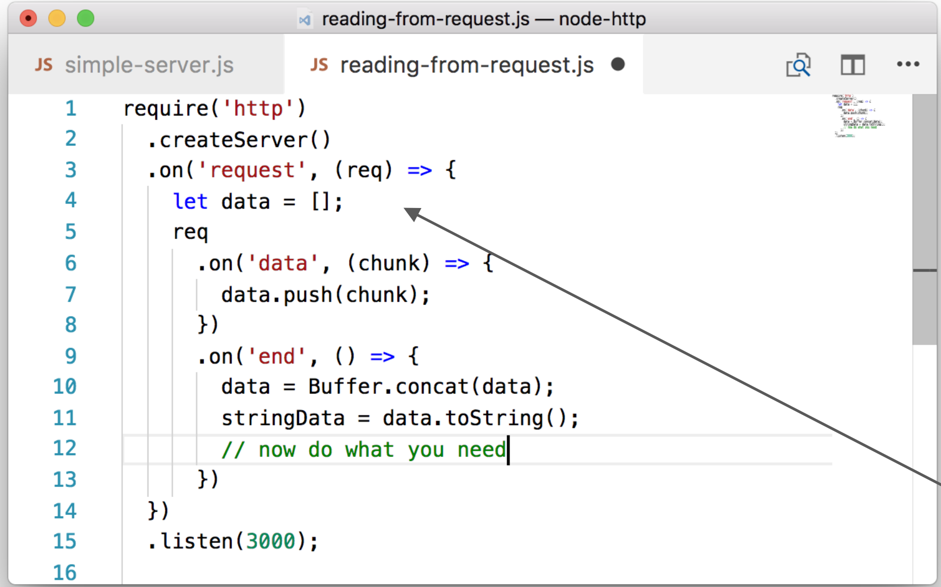
simple-server.js | reading-from-request.js

```javascript
const url = require('url');
const querystring = require('querystring');

require('http')
  .createServer()
  .on('request', (req) => {
    const query = url.parse(req.url).query;
    console.log(query); // "a=1&a=2&b=3"
    const params = querystring.parse(query);
    console.log(params); // { a: ['1', '2'], b: '3' }
  })
  .listen(3000);

```

1. Parse `req.url` and extract query string
2. Parse query string and extract params

Reading from Request

Reading `body`



```
require('http')
  .createServer()
  .on('request', (req) => {
    let data = [];
    req
      .on('data', (chunk) => {
        data.push(chunk);
      })
      .on('end', () => {
        data = Buffer.concat(data);
        stringData = data.toString();
        // now do what you need
      })
  })
  .listen(3000);
```

`request` is a `Readable` stream.

`if (req.method === 'POST')` can be used to check for particular HTTP Methods.

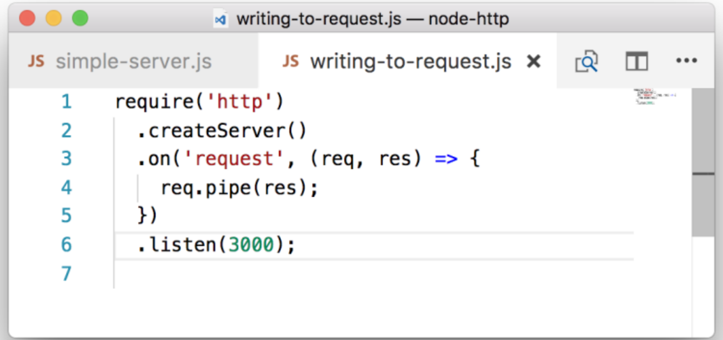
Writing to Response

`response` is a `Writable` stream

`write()` `end()` `pipe()` `writev()`

Echo

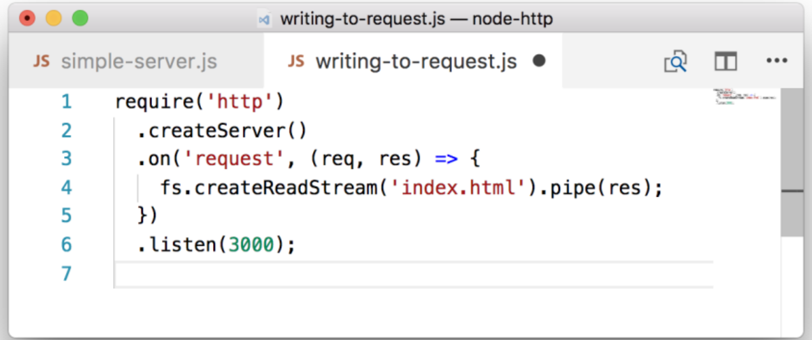
Static



writing-to-request.js — node-http

simple-server.js | writing-to-request.js

```
require('http')
  .createServer()
  .on('request', (req, res) => {
    req.pipe(res);
  })
  .listen(3000);
```



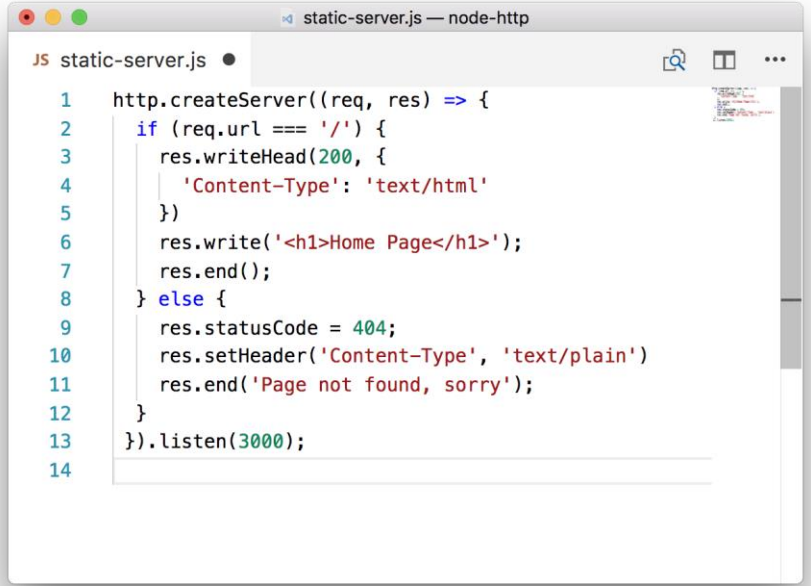
writing-to-request.js — node-http

simple-server.js | writing-to-request.js

```
require('http')
  .createServer()
  .on('request', (req, res) => {
    fs.createReadStream('index.html').pipe(res);
  })
  .listen(3000);
```

Writing to Response

Make sure to write headers before `response.write()` is called



```js
http.createServer((req, res) => {
  if (req.url === '/') {
    res.writeHead(200, {
      'Content-Type': 'text/html'
    })
    res.write('<h1>Home Page</h1>');
    res.end();
  } else {
    res.statusCode = 404;
    res.setHeader('Content-Type', 'text/plain')
    res.end('Page not found, sorry');
  }
}).listen(3000);
```

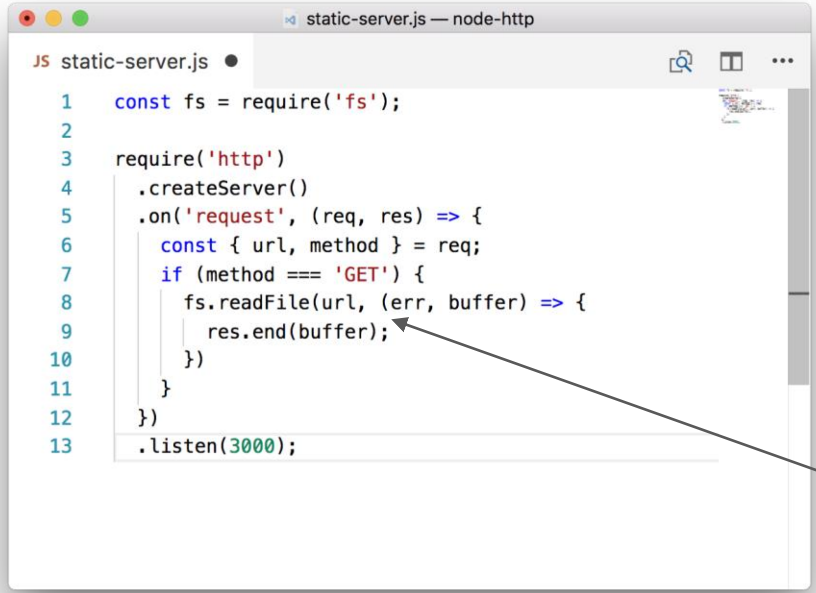
`res.statusCode + res.setHeader()`

`===`

`res.writeHead()`

Static Server

Static server is aimed for serving static resources - i.e. files.



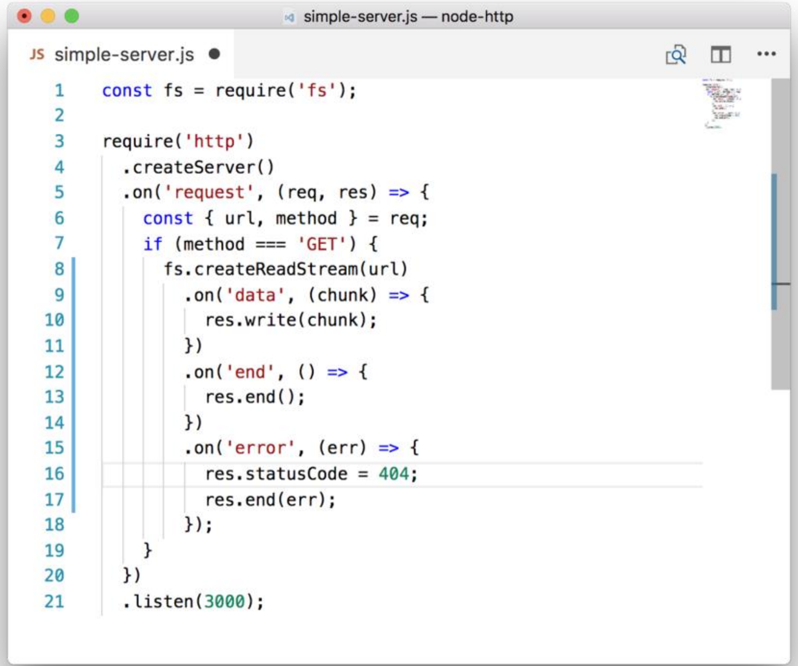
```
const fs = require('fs');

require('http')
  .createServer()
  .on('request', (req, res) => {
    const { url, method } = req;
    if (method === 'GET') {
      fs.readFile(url, (err, buffer) => {
        res.end(buffer);
      })
    }
  })
  .listen(3000);
```

Problem: huge memory consumption
Solution: streaming files

This could be a disaster for large files

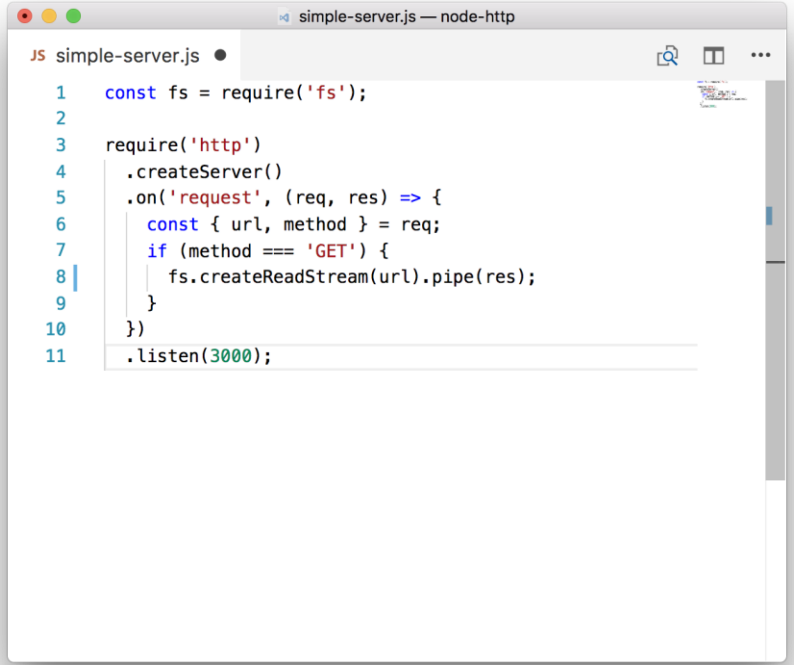
Static Server on `Stream`

Good



```js
const fs = require('fs');

require('http')
  .createServer()
  .on('request', (req, res) => {
    const { url, method } = req;
    if (method === 'GET') {
      fs.createReadStream(url)
        .on('data', (chunk) => {
          res.write(chunk);
        })
        .on('end', () => {
          res.end();
        })
        .on('error', (err) => {
          res.statusCode = 404;
          res.end(err);
        });
    }
  })
  .listen(3000);
```

Better

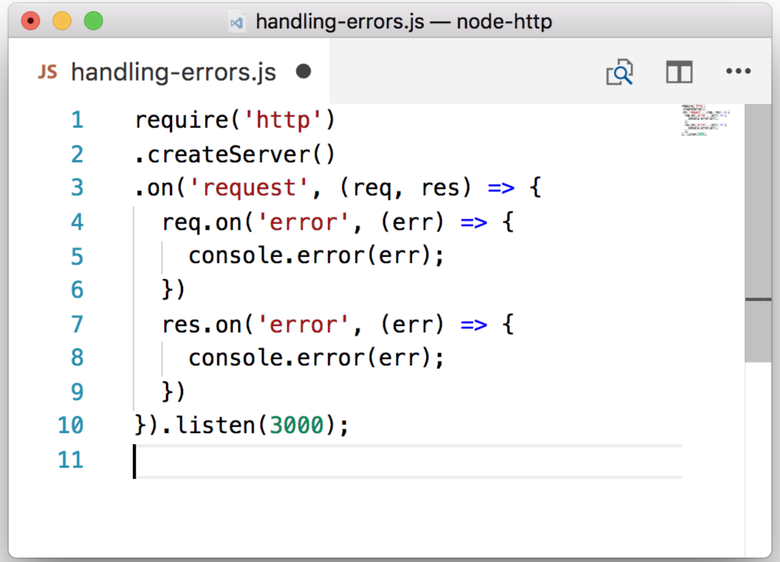


```js
const fs = require('fs');

require('http')
  .createServer()
  .on('request', (req, res) => {
    const { url, method } = req;
    if (method === 'GET') {
      fs.createReadStream(url).pipe(res);
    }
  })
  .listen(3000);
```

Error Handling

If unhandled, errors will `throw` and down your server.

Handle `error` event for both `req` and `res`.

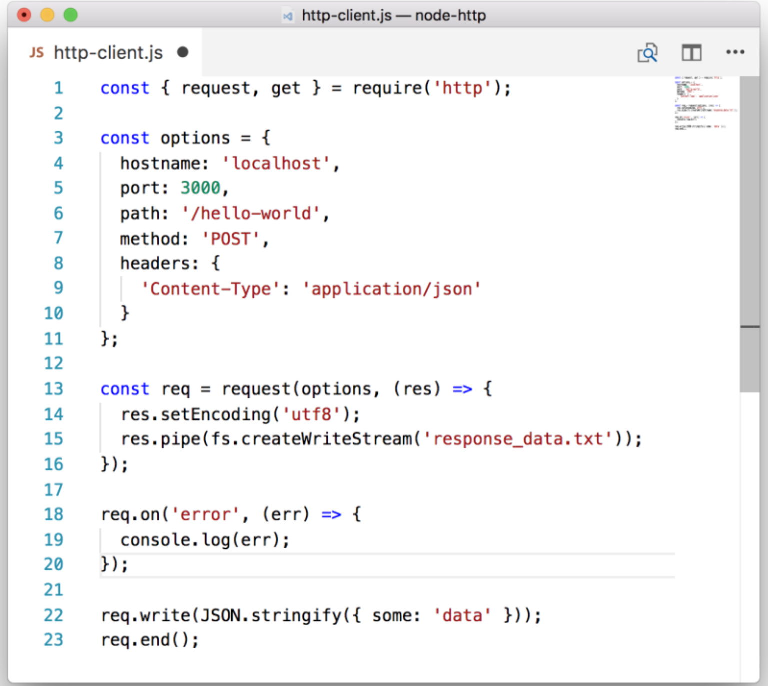

handling-errors.js — node-http

handling-errors.js

```
require('http')
.createServer()
.on('request', (req, res) => {
  req.on('error', (err) => {
    console.error(err);
  })
  res.on('error', (err) => {
    console.error(err);
  })
}).listen(3000);
```

HTTP Client

Use `http.request()`



```javascript
const { request, get } = require('http');

const options = {
  hostname: 'localhost',
  port: 3000,
  path: '/hello-world',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  }
};

const req = request(options, (res) => {
  res.setEncoding('utf8');
  res.pipe(fs.createWriteStream('response_data.txt'));
});

req.on('error', (err) => {
  console.log(err);
});

req.write(JSON.stringify({ some: 'data' }));
req.end();
```

- `request` is a `Writable` stream
- `response` is a `Readable` stream
- use `http.get()` as a shortcut for GET requests

Handling HTTPS

Issue CSR (certificate signing request) and private key:

```
openssl req -new -newkey rsa:2048 -nodes -out mydomain.csr -keyout private.key
```

THEN

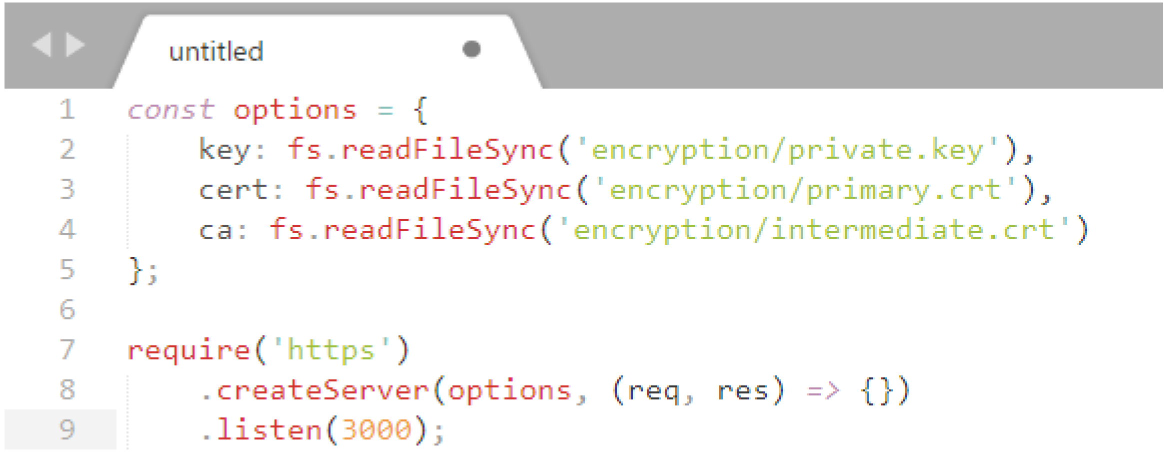
Use CSR to generate your SSL Certificates (primary and intermediate) from issuing authority.

THEN

Run your HTTPS NodeJS server.

Handling HTTPS

Running HTTPS Server:



untitled

```
const options = {
    key: fs.readFileSync('encryption/private.key'),
    cert: fs.readFileSync('encryption/primary.crt'),
    ca: fs.readFileSync('encryption/intermediate.crt')
};

require('https')
    .createServer(options, (req, res) => {})
    .listen(3000);
```

Benefits and Drawbacks

+ Handles **many** concurrent connections with ease
+ Utilizes streams to the full capacity
+ Good for WebSocket servers
+ Strong solution for file upload server (due to streams)
+ Code can be reused on the client side
+ Large community and NPM
- Needs special care to handle CPU-heavy computations
- Only takes advantage of a single core, by default

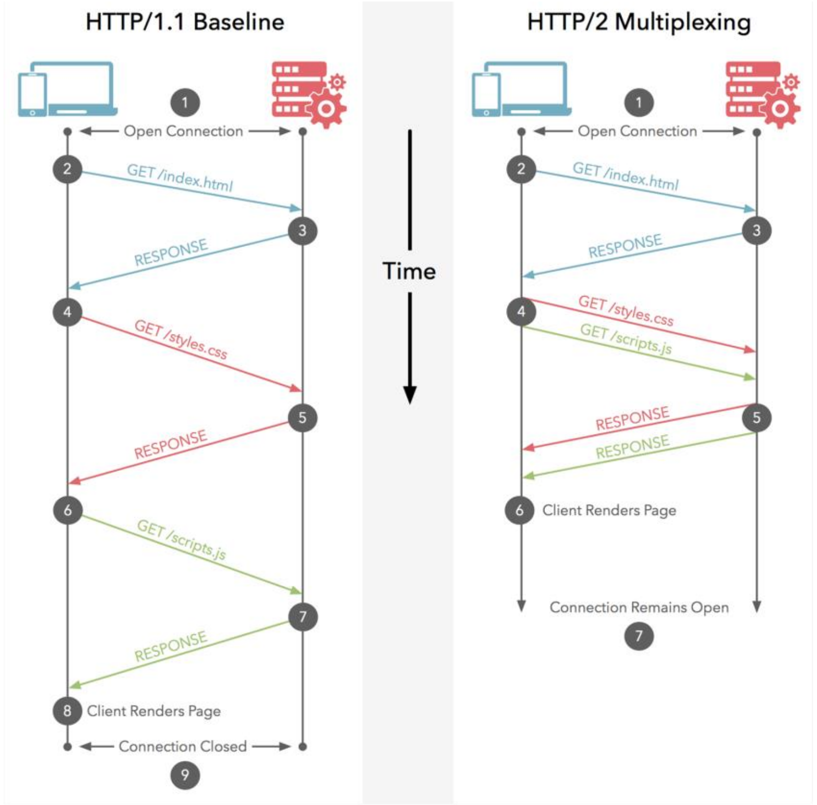
HTTP/2

Motivation:

More efficient use of network resources and reduced perception of latency

How?

- Header field compression
- Multiple concurrent exchanges on the same connection (multiplexing)
- Allow servers to “push” responses proactively into client caches
- Binary over textual
- Single TCP connection



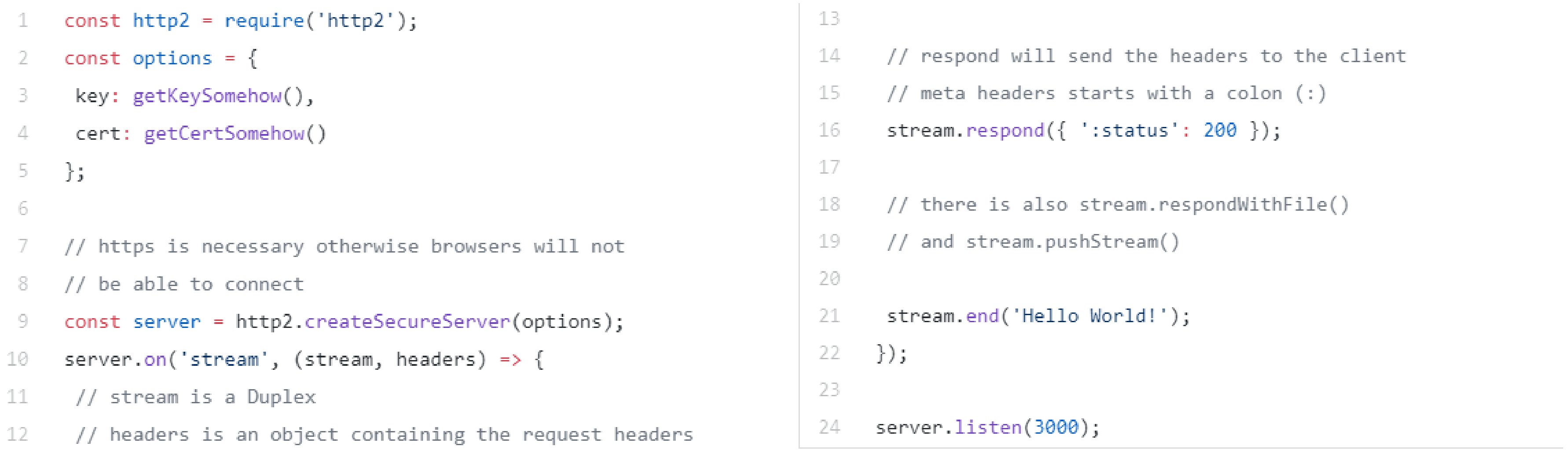
HTTP/2 Connection Establishment

- ALPN (When the client connects to the server it sends a list of supported protocols)
- HTTP upgrade based connection (`Upgrade: h2c`)

HTTP/2 in Node

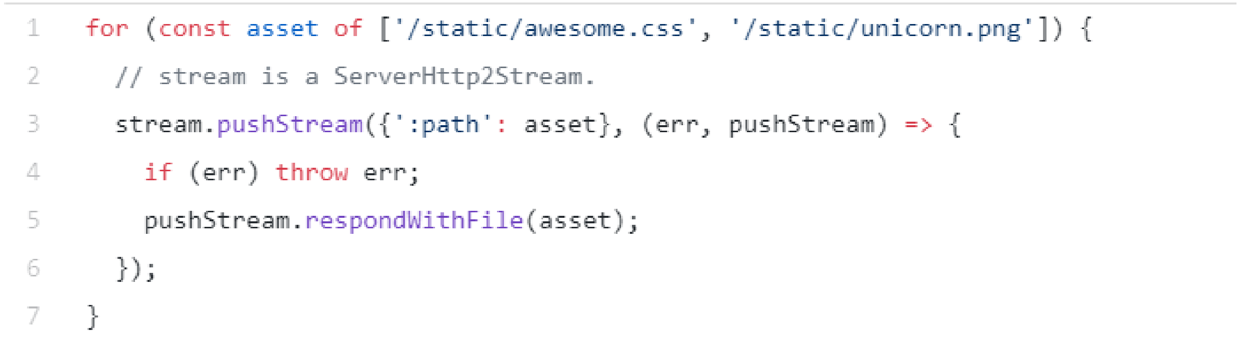
- `require('http2')` API is on **Experimental** stage
- Use `--expose-http2` flag to use it
- Not compatible with `require(‘http’)` API
- There are several well-supported libraries that provide a http-like API

HTTP/2 API Example



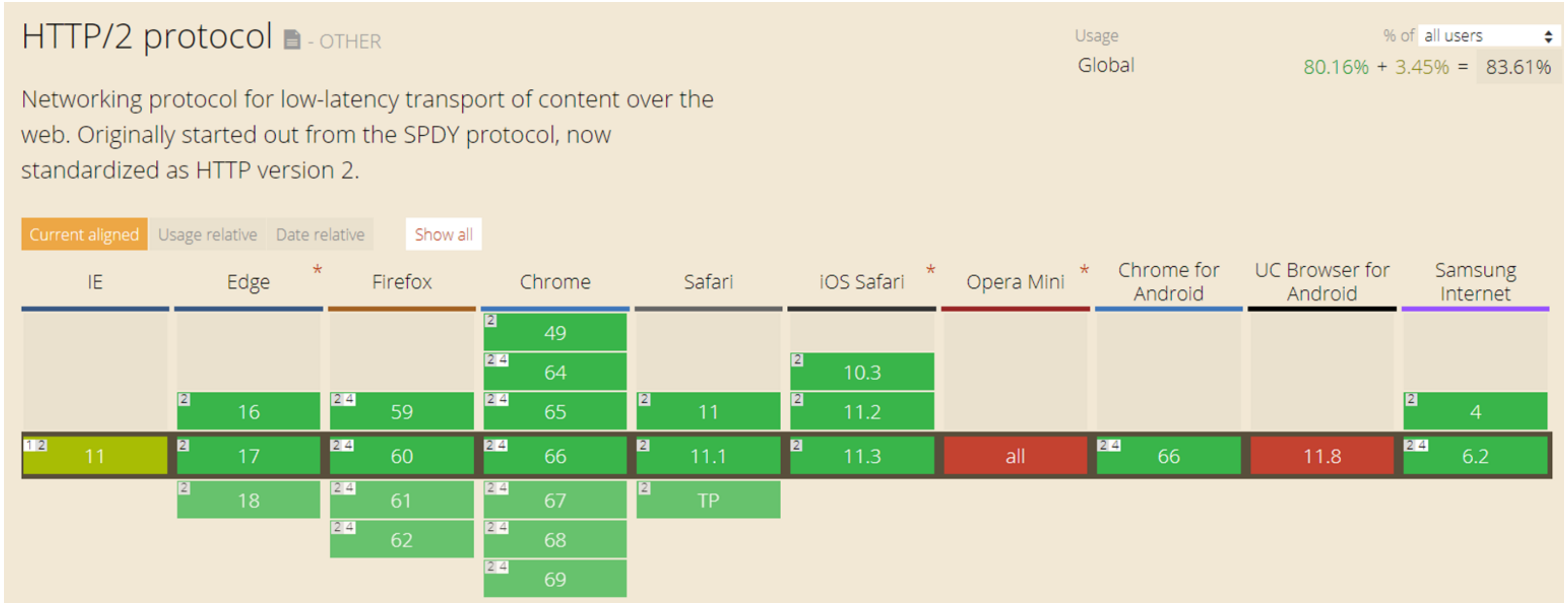
```
const http2 = require('http2');
const options = {
 key: getKeySomehow(),
 cert: getCertSomehow()
};

// https is necessary otherwise browsers will not
// be able to connect
const server = http2.createSecureServer(options);
server.on('stream', (stream, headers) => {
 // stream is a Duplex
 // headers is an object containing the request headers

 // respond will send the headers to the client
 // meta headers starts with a colon (:)
 stream.respond({ ':status': 200 });

 // there is also stream.respondWithFile()
 // and stream.pushStream()

 stream.end('Hello World!');
});

server.listen(3000);
```

HTTP/2 API Example



```
for (const asset of ['/static/awesome.css', '/static/unicorn.png']) {
  // stream is a ServerHttp2Stream.
  stream.pushStream({':path': asset}, (err, pushStream) => {
    if (err) throw err;
    pushStream.respondWithFile(asset);
  });
}
```

CanIUse



HTTP/2 protocol - OTHER

Usage Global

% of all users

80.16% + 3.45% = 83.61%

Networking protocol for low-latency transport of content over the web. Originally started out from the SPDY protocol, now standardized as HTTP version 2.

Current aligned | Usage relative | Date relative | Show all

IE	Edge *	Firefox	Chrome	Safari	iOS Safari *	Opera Mini *	Chrome for Android	UC Browser for Android	Samsung Internet
			2 49						
			2 4 64		2 10.3				
	2 16	2 4 59	2 4 65	2 11	2 11.2				2 4
1 2 11	2 17	2 4 60	2 4 66	2 11.1	2 11.3	all	2 4 66	11.8	2 4 6.2
	2 18	2 4 61	2 4 67	2 TP					
		2 4 62	2 4 68						
			2 4 69						

HTTP/2 Summary

HTTP/2 is introduced to fix issues of HTTP/1.x, but it has its drawbacks.

It’s supported in current releases of Edge, Safari, Firefox and Chrome

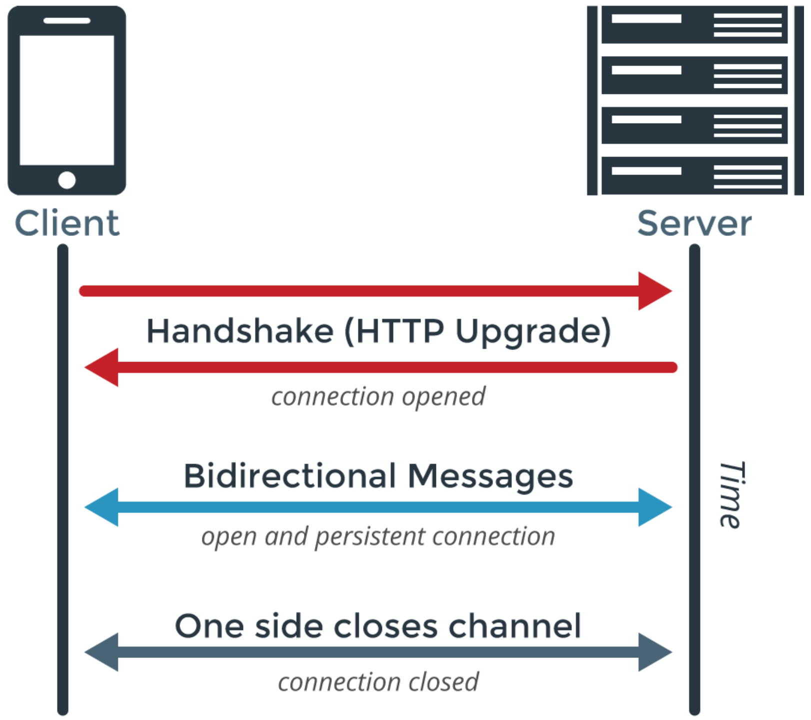
Reading:

- Spec https://http2.github.io/http2-spec/
- Node API https://nodejs.org/api/http2.html
- HTTP/2 FAQ https://http2.github.io/faq
- HTTP/2 Guide

WebSocket Protocol

WebSocket is a computer communications protocol, providing **full-duplex communication** channels over a single TCP connection.

Spec https://tools.ietf.org/html/rfc6455



WebSocket Handshake

Handshake Request from the Client:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key:
dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Handshake Response from the Server:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept:
s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

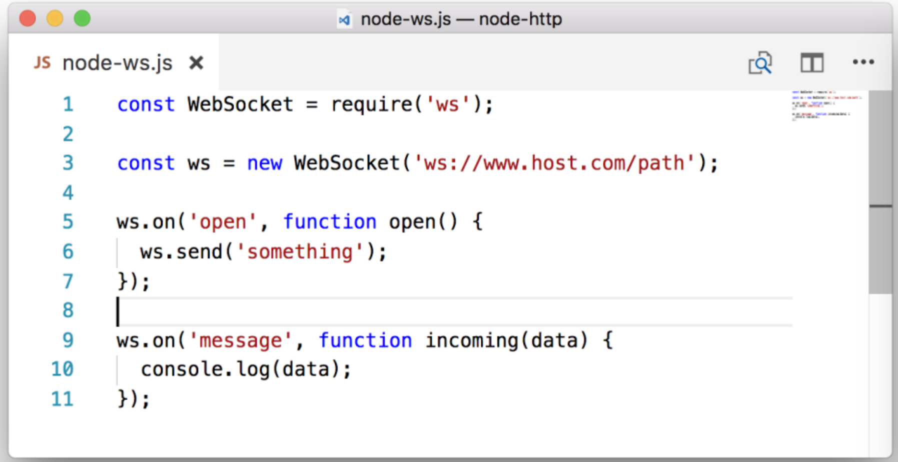
When handshake is complete, **data transfer part** starts (UTF8 or Binary)

How to WebSocket

Library	Comments	Stars
`ws`	One of the fastest WebSocket server and client for Node	★ Star 6,399
`websocket-node`	WebSocket server and client for Node	★ Star 1,880
`socket.io`	WebSocket server and client for Node + client for browsers + channels + fallbacks	★ Star 36,402 ★ Star 4,936
`sockjs`	WebSocket server and client for Node and others + client for browsers + fallbacks	★ Star 1,474 ★ Star 5,222
`faye`	WebSocket server and client for Node and others + client for browsers + fallbacks	★ Star 3,946
And more...		

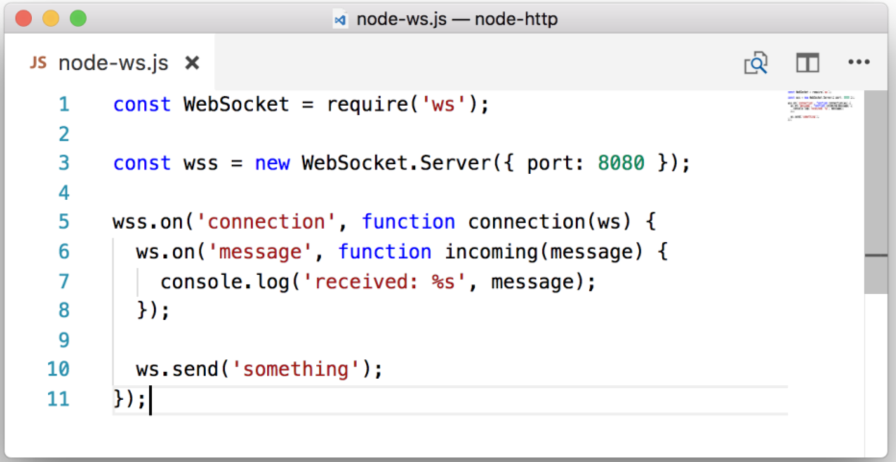
`ws` Example

Client



```
const WebSocket = require('ws');

const ws = new WebSocket('ws://www.host.com/path');

ws.on('open', function open() {
  ws.send('something');
});

ws.on('message', function incoming(data) {
  console.log(data);
});
```

Server

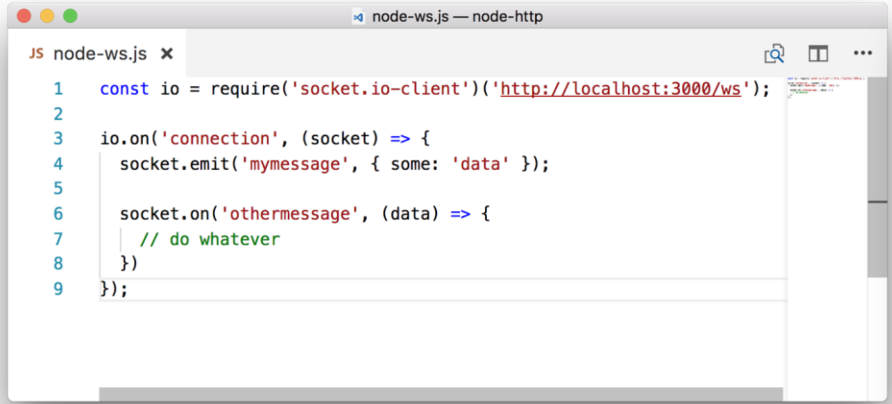


```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', function connection(ws) {
  ws.on('message', function incoming(message) {
    console.log('received: %s', message);
  });

  ws.send('something');
});
```

No browser client.

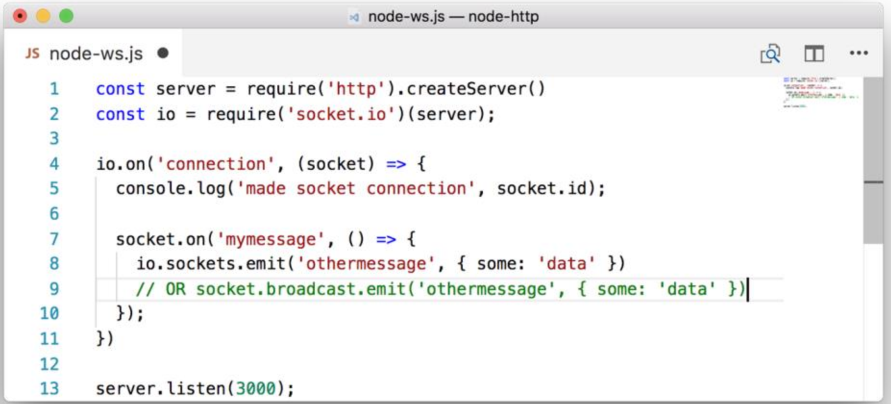
`socket.io` Example

Client



```
const io = require('socket.io-client')('http://localhost:3000/ws');

io.on('connection', (socket) => {
  socket.emit('mymessage', { some: 'data' });

  socket.on('othermessage', (data) => {
    // do whatever
  })
});
```

Server



```
const server = require('http').createServer()
const io = require('socket.io')(server);

io.on('connection', (socket) => {
  console.log('made socket connection', socket.id);

  socket.on('mymessage', () => {
    io.sockets.emit('othermessage', { some: 'data' })
    // OR socket.broadcast.emit('othermessage', { some: 'data' })
  });
})

server.listen(3000);
```

WebSocket Summary

- WebSocket is a bidirectional communication protocol
- There are various Server and Client implementations on JavaScript
- One should take into account Browser compatibility if using native WebSocket
- Both textual and binary data can be sent over WebSocket
- Connection can be closed from both - server and client - sides
- WebSocket is often used to handle real-time web applications

Questions?

Thanks