Node and HTTP/2

HTTP/2 is the later evolution of the HTTP protocol that powers the web. It's main goal is to improve performance and latency over existing HTTP 1.1 implementations. For more details see Ilyia Gregorik's High Performance Browser Networking chapter on HTTP/2

HTTP/2 landed in Node.js 8.4 behind a flag. With Node.js 9, it became an experimental part of Node core, still at stability 1. This is the version we'll be working with. Do not run production workloads using this API yet, since it might change. Check the Node HTTP/2 API documentation for more information

Getting an SSL certificate for localhost

> While it's possible use HTTP/2 without TSL no current browser supports it. If you want to serve content through HTTP/2 to web browsers you must do it securely through HTTPS.

To run the example above, you have to generate a private key and a certificate for your server. To do so, run this command:

```
openssl req -newkey rsa:2048 -new -nodes -x509 -days 3650 -keyout key.pem
```

When it asks for the common name, make sure to enter `localhost`.

We will use these keys in the following examples.

Do not use self-signed keys in production servers! For production servers you can purchase a certificate or use let's encrypt to generate free certificates.

Also be aware the most browsers will indicate that the site is insecure. You can safely ignore this warning if you're working on your local development machine.

Basic Server Example

The most basic example using this API will serve a stream to the client.

We first require the necessary modules. Because both of them are part of the core Node.js system we don't need to install them.

```
const http2 = require('http2')
const fs = require('fs')
```

Next, we creaate our securre server. We pass two parameters that are read synchronously: the location of our SSL key and certificate. We do it synchronously because we need the read to be complete before we move forward with the rest of the process.

```
const server = http2.createSecureServer({
  key: fs.readFileSync('./key.pem'),
  cert: fs.readFileSync('./cert.pem')
})
```

Remember that this is a low level API. We're working through a connection between a http2SecureServer object and one underlying socket.

We listen for three events between the socket and the server:

- error: Happens when an error occurs during the processing of an Http2Session
- sockerError: Triggered when an 'error' is emitted on the Socket instance bound to the Http2Session. If this event is not handled, the 'error' event will be re-emitted on the Socket
- stream: fired when a new Http2Stream is created. When invoked, the handler function will receive a reference to the Http2Stream object, a Headers Object, and numeric flags associated with the creation of the stream

The stream will respond with the content type of the response, the status code and the payload of the response. Because the stream is a Duplex, read and write enabled, we use the end property to write our content. See the description of the

writable stream's end method for more information and to see what additional tricks you have at your disposal.

```
server.on('error', (err) => console.error(err))
server.on('socketError', (e'socketError'e.error(err))
server.on('stream', (stream, headers) => {
  'stream'// stream is a Duplex
  stream.respond({
    'content-type': 'text/html',
    ':status': 200
  })
  stream.end('<h1>Hello World</h1>')
})
```

The final task is to set the server to listen in the specified port. In a production environment I would put the value for the port in a configuration file or in `package.json`

```
server.listen(7300)
```

Static File Server

> This file server is taken from How to create a zero dependency HTTP/2 static file server with Node.js (with examples)

As usual we require the files that we need. `mime-types` is not part of core Node so you must install with with npm (`npm i mime-types`) before continuing.

```
const http2 = require('http2');
const fs = require('fs');
const path = require('path');
const mime = require('mime-types');
```

The next block of code handles configuration of the server.

- We define http2.constants to make it easier to work with error codes.
- Next, we create an object holding the location of the certificate and key
- We set up the server using the options object as a parameter
- The location of the server root; in this case the `public` directory.

```
const {
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_METHOD,
  HTTP_STATUS_NOT_FOUND,
  HTTP_STATUS_INTERNAL_SERVER_ERROR
} = http2.constants;

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

const server = http2.createSecureServer(options);

const serverRoot = "./public";
```

`respondToStreamError` is a functions that will handle 400 (`HTTP_STATUS_NOT_FOUND`) and 500 (`HTTP_INTERNAL_SERVER_ERROR`) error codes.

```
function respondToStreamError(err, stream) {
  console.log(err);
  if (err.code === 'ENOENT') {
      stream.respond({ ":status": HTTP_STATUS_NOT_FOUND });
  } else {
      stream.respond({ ":status": HTTP_STATUS_INTERNAL_SERVER_ERROR });
  }
  stream.end();
}
```

The stream method is where we make all the changes. We first set up variables to hold the following information about the request

- Request Path
- Request Method
- The full path to the requested item
- The mime type of the objject we're responding with

We then use `respondWithFile` to return the file with the appropriate mime type and use `respondToStreamError` to provide an error if appropriate.

```
server.on('stream', (stream, headers) => {
  const reqPath = headers[HTTP2_HEADER_PATH];
  const reqMethod = headers[HTTP2_HEADER_METHOD];

  const fullPath = path.join(serverRoot, reqPath);
  const responseMimeType = mime.lookup(fullPath);

  stream.respondWithFile(fullPath, {
      'content-type': responseMimeType
  }, {
      onError: (err) => respondToStreamError(err, stream)
  });
});
```

As always we listen in the specified port.

```
server.listen(7350);
```

Pushing Resources: Push

One of the best new features, and one that is very hard to use correctly, is server push. The idea is that, if we know that the current page or another page on the site will use a resource, we can have the server push the resource to the browser before it requests it.

The configuration and `respondToStreamError` are the same as the static file server.

```
const http2 = require('http2');
const fs = require('fs');
const path = require('path');
const mime = require('mime-types');

const {
  HTTP2_HEADER_PATH,
  HTTP2_HEADER_METHOD,
  HTTP_STATUS_NOT_FOUND,
  HTTP_STATUS_INTERNAL_SERVER_ERROR
} = http2.constants;

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

const server = http2.createSecureServer(options);

const serverRoot = "./public";

function respondToStreamError(err, stream) {
  console.log(err);
  if (err.code === 'ENOENT') {
      stream.respond({ ":status": HTTP_STATUS_NOT_FOUND });
  } else {
      stream.respond({ ":status": HTTP_STATUS_INTERNAL_SERVER_ERROR });
  }
  stream.end();
}

server.on('stream', (stream, headers) => {
  const reqPath = headers[HTTP2_HEADER_PATH];
  const reqMethod = headers[HTTP2_HEADER_METHOD];

  const fullPath = path.join(serverRoot, reqPath);
  const responseMimeType = mime.lookup(fullPath);
```

It's inside the stream event that we make our changes. First, if the file ends with `.html` we respond with the file matching the name and assign the correct value to the `content-type` header. If there's an error we respond with `respondToStreamError`.

We could use the same system to generate responses for other content types that we know we'll serve on our pages.

```
if (fullPath.endsWith(".html")) {
    console.log('html');
    'html'// handle HTML file
    stream.respondWithFile(fullPath, {
      "content-type": "text/html"
    }, {
      onError: (err) => {
        respondToStreamError(err, stream);
      }
    });
```

We use `pushStream` to initate a push event. We know that we want to push `font.woff` and we know the id of the parent stream. All that is lleft is to add the resource to the push stream send it to the client.

As usual, if we get an error we use `respondToStreamError` to provide a response.

```
    stream.pushStream({ ":path": "/font.woff" }, { parent: stream.id },
      console.log('pushing');
      pushStream.respondWithFile(path.join(serverRoot, "/font.woff"), {
        'content-type': "text/css"
      }, {
          onError: (err) => {
            respondToStreamError(err, pushStream);
          }
      });
    });
```

If it's not an HTML file then we just serve it normally, using `respondWithFile` to

return the resource and using `responseMimeType` as the value for the `content-type` header.

```
  } else {
    // handle static file
    console.log(fullPath);
    stream.respondWithFile(fullPath, {
      'content-type': responseMimeType
   'content-type'Error: (err) => respondToStreamError(err, stream)
    });
  }

});
```

The last step is to hear for requests on the specified port.

```
server.listen(7350);
```

Express

Express has plans to support the native Node.js implementation on version 5.0, which is currently in alpha release and without support for . If you're interested there is a tracking issue for Express 5.0 in Github.

If you can't wait

If you really think you must implement HTTP/2 in your production application right now there are implementations of HTTP/2 and SPDY (the framework that HTTP/2 is based on) available.

I've played with the node-spdy module. It provides a more mature implementation of HTTP/2 in Node and also supports the Google propietary SPDY server extensions that were the basis for HTTP/2.

After installing the spdy module (`npm i spdy`) my starting point (taken from the module's README) looks like this

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

const options = {
 'fs': fs.readFileSync('./key.pem'),
  cert: fs.readFileSync('./cert.pem'),

  spdy: {
    protocols: ['./key.pem'/3.1', 'http/1.1'],
    plain: false,
    'x-forwarded-for': false,

    connection: {
      windowSize: 1024 * 1024, ', '// Server's window size
      autoSpdy31: false
    }
  }
};

const server = spdy.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end('hello world!');
});

server.listen(3000);
```

Links and Resources

- HTTP/2 Specs
    - Hypertext Transfer Protocol version 2 - RFC7540
    - HPACK - Header Compression for HTTP/2 - RFC7541
- HTTP/2 FAQ
- Introduction to HTTP/2
- Rules of Thumb for HTTP/2 Push
- Node.js HTTP/2 documentation
- node-spdy module