# HTTP

require("dotenv/config");

const fs = require("fs");

const path = require("path");

const mime = require("mime");

const http = require("http");

const port = process.env.PORT || 3000;

const server = http.createServer((req, res) => {

  const absPath = path.join(\_\_dirname, "public", "index.html");

  res.writeHead(200, {

    "Content-Type": mime.getType(absPath),

  });

  console.log(absPath);

  console.log(mime.getType(absPath));

  console.log(mime.getExtension(mime.getType(absPath)));

  const file = fs.readFileSync(absPath, "utf-8");

  res.end(file);

});

server.listen(port, () => {

  console.log(`Server running at port ${port}`);

});

Download

require("dotenv/config");

const fs = require("fs");

const path = require("path");

const mime = require("mime");

const http = require("http");

const port = process.env.PORT || 3000;

const server = http.createServer((req, res) => {

  const absPath = path.join(\_\_dirname, "public", "videos", "tuts.mp4");

  res.writeHead(200, {

    "Content-Type": mime.getType(absPath),

    "Content-Disposition": `attachment; filename=${path.basename(absPath)}`,

  });

  console.log(`Path : ${absPath}`);

  console.log(`Filename : ${path.basename(absPath)}`);

  console.log(`MimeType : ${mime.getType(absPath)}`);

  console.log(`Extension : ${mime.getExtension(mime.getType(absPath))}`);

  const readStream = fs.createReadStream(absPath);

  readStream.pipe(res);

});

server.listen(port, () => {

  console.log(`Server running at port ${port}`);

});

Render index.html + Static Files

require("dotenv/config");

const fs = require("fs");

const path = require("path");

const mime = require("mime");

const http = require("http");

const port = process.env.PORT || 3000;

const server = http.createServer((req, res) => {

  if (req.url === "/") {

    /\* Hiển thị index \*/

    const absPath = path.join(\_\_dirname, "public", "index.html");

    res.writeHead(200, {

      "Content-Type": mime.getType(absPath),

    });

    const file = fs.readFileSync(absPath);

    res.end(file);

  } else {

    /\* Load static \*/

    const paths = req.url.split("/");

    let absPath = path.join(\_\_dirname, "public");

    paths.forEach((p) => {

      absPath = path.join(absPath, p);

    });

    try {

      res.writeHead(200, {

        "Content-Type": mime.getType(absPath),

      });

      res.end(fs.readFileSync(absPath));

    } catch (err) {

      res.writeHead(404);

    }

  }

});

server.listen(port, () => {

  console.log(`Server running at port ${port}`);

});

Http

|  |
| --- |
| * Class: http.Agent * new Agent([options]) * agent.createConnection(options[, callback]) * agent.destroy() * agent.freeSockets * agent.getName(options) * agent.maxFreeSockets * agent.maxSockets * agent.maxTotalSockets * agent.requests * agent.sockets |

|  |
| --- |
| * Class: http.ClientRequest * Event: 'abort' * Event: 'connect' * Event: 'continue' * Event: 'information' * Event: 'response' * Event: 'socket' * Event: 'timeout' * Event: 'upgrade' * request.abort() * request.aborted * request.connection * request.end([data[, encoding]][, callback]) * request.destroy([error]) * request.destroyed * request.finished * request.flushHeaders() * request.getHeader(name) * request.maxHeadersCount * request.path * request.method * request.host * request.protocol * request.removeHeader(name) * request.reusedSocket * request.setHeader(name, value) * request.setNoDelay([noDelay]) * request.setSocketKeepAlive([enable][, initialDelay]) * request.setTimeout(timeout[, callback]) * request.socket * request.writableEnded * request.writableFinished * request.write(chunk[, encoding][, callback])   request.setHeader('content-type', 'text/html');  request.setHeader('Content-Length', Buffer.byteLength(body));  request.setHeader('Cookie', ['type=ninja', 'language=javascript']);  const contentType = request.getHeader('Content-Type');  // 'contentType' is 'text/html'  const contentLength = request.getHeader('Content-Length');  // 'contentLength' is of type number  const cookie = request.getHeader('Cookie');  // 'cookie' is of type string[] |

|  |
| --- |
| * Class: http.Server * Event: 'checkContinue' * Event: 'checkExpectation' * Event: 'clientError' * Event: 'close' * Event: 'connect' * Event: 'connection' * Event: 'request' * Event: 'upgrade' * server.close([callback]) * server.headersTimeout * server.listen() * server.listening * server.maxHeadersCount * server.requestTimeout * server.setTimeout([msecs][, callback]) * server.timeout * server.keepAliveTimeout |

|  |
| --- |
| * Class: http.ServerResponse * Event: 'close' * Event: 'finish' * response.addTrailers(headers) * response.connection * response.cork() * response.end([data[, encoding]][, callback]) * response.finished * response.flushHeaders() * response.getHeader(name) * response.getHeaderNames() * response.getHeaders() * response.hasHeader(name) * response.headersSent * response.removeHeader(name) * response.sendDate * response.setHeader(name, value) * response.setTimeout(msecs[, callback]) * response.socket * response.statusCode * response.statusMessage * response.uncork() * response.writableEnded * response.writableFinished * response.write(chunk[, encoding][, callback]) * response.writeContinue() * response.writeHead(statusCode[, statusMessage][, headers]) * response.writeProcessing()   response.setHeader('Content-Type', 'text/html');  response.setHeader('Content-Length', Buffer.byteLength(body));  response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);  const contentType = response.getHeader('content-type');  // contentType is 'text/html'  const contentLength = response.getHeader('Content-Length');  // contentLength is of type number  const setCookie = response.getHeader('set-cookie');  // setCookie is of type string[]  response.setHeader('Foo', 'bar');  response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);  const headerNames = response.getHeaderNames();  // headerNames === ['foo', 'set-cookie']  const headers = response.getHeaders();  // headers === { foo: 'bar', 'set-cookie': ['foo=bar', 'bar=baz'] }  response.removeHeader('Content-Encoding');  response.statusCode = 404;  response.statusMessage = 'Not found';  // Returns content-type = text/plain  const server = http.createServer((req, res) => {      res.setHeader('Content-Type', 'text/html');      res.setHeader('X-Foo', 'bar');      res.writeHead(200, { 'Content-Type': 'text/plain' });      res.end('ok');    }); |

|  |
| --- |
| * Class: http.IncomingMessage * Event: 'aborted' * Event: 'close' * message.aborted * message.complete * message.destroy([error]) * message.headers * message.httpVersion * message.method * message.rawHeaders * message.rawTrailers * message.setTimeout(msecs[, callback]) * message.socket * message.statusCode * message.statusMessage * message.trailers * message.url |

|  |
| --- |
| * http.METHODS * http.STATUS\_CODES * http.createServer([options][, requestListener]) * http.get(options[, callback]) * http.get(url[, options][, callback]) * http.globalAgent * http.maxHeaderSize * http.request(options[, callback]) * http.request(url[, options][, callback]) * http.validateHeaderName(name) * http.validateHeaderValue(name, value)   const http = require('http');  const agent = new http.Agent({ keepAlive: true });  // Server has a 5 seconds keep-alive timeout by default  http    .createServer((req, res) => {      res.write('hello\n');      res.end();    })    .listen(3000);  setInterval(() => {    // Adapting a keep-alive agent    http.get('http://localhost:3000', { agent }, (res) => {      res.on('data', (data) => {        // Do nothing      })        res.on('error', (err) => {          // Check if retry is needed          if (req.reusedSocket && err.code === 'ECONNRESET') {          retriableRequest();          }      });    });  }, 5000); // Sending request on 5s interval so it's easy to hit idle timeout |