CS572 Modern Web Applications Programming
Maharishi University of Management
Department of Computer Science
Assistant Professor Asaad Saad

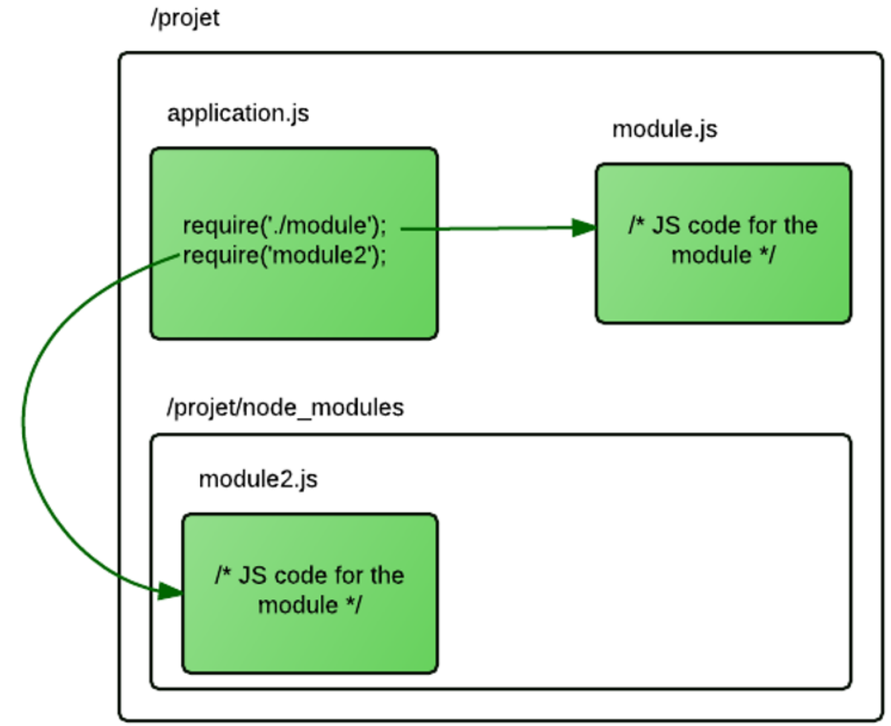
PREV NEXT

OUTLINE

Search...

Modules in NodeJS

- Node implements **CommonJS** Modules specs.
- If we simply create a normal JS file it will be a module *(without affecting the rest of other JS files and without messing with the global scope).*



https://openclassrooms.com

Explore Module and Require

```
> module
        Module { id: '<repl>',
                 exports: {},
                 parent: undefined,
                 filename: null,
                 loaded: false,
                 children: [],
                 paths: [ ... ]
    }
    > require
          { [Function: require]
          resolve: [Function: resolve],
          main: undefined,
          extensions: { '.js': [Function], '.json': [Function], '.node': [Function] },
          cache: {}
    }
```

Look at differences in the extensions functions:

```
require.extensions['.js'].toString()
require.extensions['.json'].toString()
require.extensions['.node'].toString()
```

How `require()` works

1. Resolve
2. Load
3. Wrap
4. Evaluate (execute and implicitly return **`module.exports`**)
5. Cache

Note: Node core modules return immediately (no resolve)

What's the wrapper?

```
node -p "require('module').wrapper"
```

1. Node will wrap my code into:

```
(function (exports, require, module, __filename, __dirname){
    // mycode.js
    // this is why I can use exports and module objects.. etc in my code
    without any problem, because Node is going to initialize these for me and
    pass them to me as parameters through this wrapper function
});
```

2. Node will run the function using .apply()

3. Node will return the following:

```
return module.exports;
```


PREV NEXT

`require(path)`

Encapsulates my JS code in IIFE (protected) and run it and returns what we **explicitly** attach to **`module.exports`** object.

- Add **`./`** to make it look into **local location** for the the JS file
- Passing simply a **`filename`** means requiring a **node core module or dependency**
- If we pass a **`folder`** without **`filename`** will look for file with name **`index.js`**

require() has **cachedModules** that returns the same object when called more than one time. That is very efficient to require many files in my application efficiently.

We can **ignore file extension** (.js). But if we want to require files with an extension other than .js then we must mention the it. If the file is json, it will be automatically parsed for me as an object.

```
const ref = require (__dirname + '/module');
const data = require('./data.json');
```


PREV NEXT

module.exports

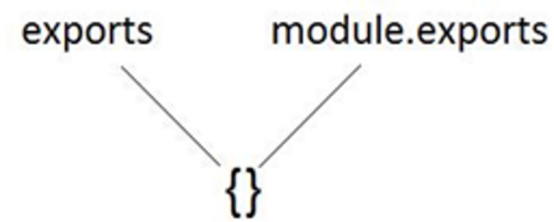
Think about this object as **return** statement, this is why we should change the way we "require" the module to be contained in a variable.

```
// helloModule.js
var sayHi = function(){
                console.log('hi');
            }

module.exports = sayHi;
```

```
// app.js
var hello = require('./helloModule');

hello();
```

`exports` vs `module.exports`

`exports` object is a reference to the `module.exports` that is shorter to type



Both of them are references to the same (empty) object at the beggining. (But only module.exports will be returned!)

Be careful when using `exports`, a code like this will make it point to another object and in the end `module.exports` will be returned.

```
exports = function Something() {
    console.log('blah blah');
}
```

Create your own module

play folder

```
// play/violin.js
const play = function() { console.log("First Violin is playing!"); }
module.exports = play;
```

```
// play/clarinet.js
const play = function() { console.log("Clarinet is playing!"); }
module.exports = play;
```

```
// play/index.js
const violin = require('./violin');
const clarinet = require('./clarinet');
module.exports = { 'violin': violin, 'clarinet': clarinet };
```

```
// app.js
const play = require('./play');
play.violin();
play.clarinet();
```

Using Modules

```
// Pattern1 - pattern1.js
module.exports = function() {
                    console.log('CS572');
                };
```

```
// app.js
const getCourseName = require('./pattern1');
getCourseName(); // CS572
```

Using Modules

```
// Pattern2 - pattern2.js
module.exports.getCourseName = function() {
                                  console.log('CS572');
                               };
```

```
// app.js
const getCourseName = require('./pattern2').getCourseName;
getCourseName(); // CS572
```

OR

```
// app.js
const myCourse = require('./pattern2');
myCourse.getCourseName(); // CS572
```


PREV NEXT

Using Modules

```
// Pattern3 - pattern3.js
function Course() {
        this.courseName = 'CS472';
        this.getCourseName = function() {
                                        console.log(this.courseName);
                                }
}
module.exports = new Course();
```



```
// app.js
let courseInstance = require('./pattern3');
courseInstance.getCourseName(); // CS472
courseInstance.courseName = 'CS572';
let courseInstanceTwo = require('./pattern3'); // cached
courseInstanceTwo.getCourseName(); // CS572
```

Using Modules

```
// Pattern4 - pattern4.js
function Course() {
        this.courseName = 'CS572';
        this.getCourseName = function() {
                                        console.log(this.courseName);
                                }
}
module.exports = Course;
```

```
// app.js
const courseConstructor = require('./pattern4');
const courseInstance = new courseConstructor();
courseInstance.getCourseName(); // CS572
courseInstance.courseName = 'x';
const courseConstructor2 = require('./pattern4');
const courseInstance2 = new courseConstructor2();
courseInstance2.getCourseName(); // CS572
```

Using Modules

```
// Pattern5 - pattern5.js
const courseName = 'CS572';
function getCourseName() {
            console.log(courseName); // closure
}
module.exports = {
        getCourseName: getCourseName // closure
}
```



```
// app.js
const getCourseNameMethod = require('./pattern5').getCourseName;
getCourseNameMethod(); // CS572
```

Node Core Libraries

Node provides me with many core libraries (nodejs.org > API)

```
const util = require('util'); // We do not use ./ before the filename
const path = require('path');
const sayHi = util.format("Hi, %s", 'Asaad');
util.log(sayHi); // Hi, Asaad
```

We should read the API documentation to know how to use each one of these.

What will happen if I create a module and I name it the same as one of the core modules?

Events

- We should understand two different kinds of events:
  - **Custom Events**, Objects using EventEmitter from Node.
  - **System Events** and that comes from C++ core (libuv) – Examples: I *finished reading a file, the file is open, I received data from the internet (things JS doesn't have)* - Because Node code is wrapping calls to the C++ side. when something happened it generates JS events.

Custom Event Emitter

```
function Emitter() {
        this.events = {}; // Empty Object
}
```

```
events = {
        'onSomeoneClick' : [function1, function2],
        'onDataAvailable' : [function1]
}
```

```
Emitter.prototype.on = function(type, listener) {
        this.events[type] = this.events[type] || [];
        this.events[type].push(listener);
}
Emitter.prototype.emit = function(type) {
        if (this.events[type]) {
                this.events[type].forEach(function(listener) {
                                                listener();
                                        });
        }
}
module.exports = Emitter;
```

A simple Emitter object is just an object that contains a key of "type", its value will be an array of listeners.

Using our Custom Basic Emitter

```
var Emitter = require('./emitter');
var emtr = new Emitter();

emtr.on('greet', function() {
        console.log('Hi from CS572');
});
console.log('Hello!');
emtr.emit('greet');
```

However, Node Emitter's more reliable, flexible and advanced than the one we built!

`Events` core library

- Find full documentation here: https://nodejs.org/api/events.html
- It has **.emit()** on the prototype using .call() .apply() to invoke all the listeners functions.
- It has **.on()** which is a shortcut to addListener() function also on the prototype.
- It has better logic for adding the listeners into the object/array.
- It controls the memory in a better way!

Using Event Emitter

```
var EventEmitter = require('events');

class ComproStudent extends EventEmitter {
    constructor(){
        super();
        this.message = 'New Student!';
    }
    visit() {
        console.log(this.message);
        this.emit('newStudent', 'Asaad');
    }
}

var student = new ComproStudent();
student.on('newStudent', function(name){console.log(`Welcome ${name}`)});
```

Inheritance with ES6

```
function Welcome() {
        EventEmitter.call(this);
        this.message = 'New Student!';
}

util.inherits(Welcome, EventEmitter);

Welcome.prototype.visit = function() {
             console.log(this.message);
              this.emit('newStu');
}
```

```
'use strict';

class Welcome extends EventEmitter {
    constructor(){
        super();
        this.message = 'New Student!';
    }
    visit() {
        console.log(this.message);
        this.emit('newStu');
    }
}
```

ES5 ES6

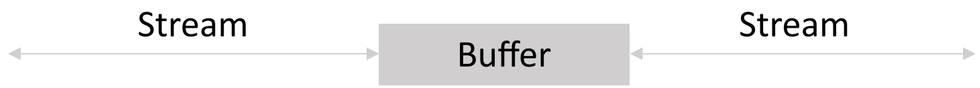
process inherits from Event Emitter

```
process.on('exit', (code) => {
        // do one final synchronous operation before the node process terminates
        console.log(`About to exit with code: ${code}`);
});

process.on('uncaughtException', (err) => {
        // Do any cleanup and exit anyway!
        console.error(err);
        // FORCE exit the process too.
        process.exit(1);
});
```

Buffers and Streams

Buffer: A chunk of memory allocated outside V8 (intentionally limited in size, we cannot resize an allocated buffer)

Stream: a sequence of data made available over time. Pieces of data that eventually combine into a whole.



Example: If you stream a movie over the internet, you are processing the data as being received. there will be a buffer inside your browser that receive the data from the server, process it when enough data is available (few seconds of pictures) , and stream it out to your monitor.

Buffer is a Super Data Type

- **`Buffer`** is a Node.js addition to primitives (boolean, string, number..etc) and all other objects (Array, Function.. etc).
- We can think of buffers as extremely efficient data stores. In fact, Node.js will try to use buffers any time it can, most Node API is built using buffers behind the scenes, for example: reading from file system, receiving packets over the network.
- Generally, you can pass buffers on every Node API requiring data to be sent (response object). They are very helpful when reading files and images and send streams.

Character set vs. Encoding

Character set: A representation of characters as numbers, each character gets a number. Unicode and ASCII are character sets. Where character get a number assigned to them.

Encoding: How characters are stored in binary, the numbers (code points) are converted and stored in binary.

	h	e	l	l	o
Unicode character set	104	101	108	108	111
UTF-8 encoding	01101000	01100101	01101100	01101100	01101111

Remember in HTML when a binary response comes from the server it's mandatory in HTML5 to specify the encoding in the header <meta charset='utf-8'> Specify the character encoding for the HTML document.

36

Buffers Encoding

Buffers data are saved in binary, so it can be interpreted in many ways depending on the length of characters. This is why we need to specify the char-encoding when we read data from Buffers. If we don't specify the encoding then we will receive a Buffer object with the data represented in a hex format in the console.

PREV NEXT

Buffer Examples

```
var buf = Buffer.alloc(8); // allocate a Buffer with 8 bytes
var buf = Buffer.allocUnsafe(8); allocated 8 bytes Buffer with random data
var buf = Buffer.allocUnsafe(8).fill(); // fill buffer with 0
var buf = Buffer.from('Hello'); // create Buffer of 5 bytes and fill it
var buf = new Buffer('Hello', 'utf8');
console.log(buf);
// 48 65 6c 6c 6f stored internally in binary but displayed as hex in CLI

console.log(buf[2]); // 108 (Charset)

console.log(buf.toString()); // Hello

console.log(buf.toJSON());
// {type: 'Buffer', data = [72, 101, 108, 108, 111 ] } Unicode characterset

buf.write('wo'); // overwrite data in the buffer without changing its size
console.log(buf.toString()); // wollo
```


PREV NEXT

Slice a buffer

A buffer can be sliced into a smaller buffer by using the slice() method:

```
var buffer = new Buffer('this is the string in my buffer');
var slice = buffer.slice(10, 20);
```

Here we are slicing the original buffer that has 31 bytes into a new buffer that has 10 bytes equal to the 10th to 20th bytes on the original buffer.

Note that the slice function does not create new buffer memory: it uses the original untouched buffer underneath.

Tip: If you are afraid you will be wasting precious memory by keeping the old buffer around when slicing it, you can copy it into another

Copy a buffer

You can copy a part of a buffer into another pre-allocated buffer like this:

```
var buffer = new Buffer('this is the string in my buffer');
var targetBuffer = new Buffer(10);
var targetStart = 0,
    sourceStart = 10,
    sourceEnd = 20;
buffer.copy(targetBuffer, targetStart, sourceStart, sourceEnd);
```

Here we are copying part of buffer (positions 10 through 20) into new buffer.

Files

```
var fs = require('fs');
var path = require('path');

var greet = fs.readFileSync(path.join(__dirname, 'greet.txt'), 'utf8');
console.log(greet);
```

```
var greet2 = fs.readFile(path.join(__dirname, 'greet.txt'), 'utf8',
                         function(err, data) { console.log(data);} );
console.log('Done!');

// Hello
// Done!
// Hello
```

Notice the Node Applications Design: any async function accepts a **callback as a last parameter** and the **callback function accepts error as a first parameter** (null will be passed if there is no error).

Notice: data here is a buffer object. We can convert it with toString or add the encoding to the command

Example Read/Write Files

```
var fs = require('fs');
var path = require('path');

// Reading from a file:
fs.readFile(path.join(__dirname, 'data/students.csv'),
                {encoding: 'utf-8'}, function (err, data) {
                                        if (err) throw err;
                                        console.log(data);
                                    });
// Writing to a file:
fs.writeFile('students.txt', 'Hello World!', function (err) {
                                        if (err) throw err;
                                        console.log('Done');
                                    });
```

What's the problem with the code above?

Revisit the Event Loop

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

fs.readFile(path.join(__dirname, 'greet.txt'), 'utf8', function(err, data) {
        setTimeout(() => { console.log('timeout'); }, 0);
        setImmediate(() => { console.log('immediate'); });
        process.nextTick(()=> console.log('nexttick'));
});
```

Based on your understanding to the Event Loop, what is the output of the code?

Streams

Collection of data that might not be available all at once and don't have to fit in memory. Streaming the data means an application processes the data while it's still receiving it. This is useful for extra large datasets, like video or database migrations.

Stream types:

- **Readable (fs.createReadStream)**
- **Writable (fs.createWriteStream)**
- **Duplex (net.Socket)** both readable/writable streams
- **Transform (zlib.createGzip)** Duplex streams that can modify/transform the data

Streams inherit from `EvenEmitter`. So they have access to `on()` and `emit()`.

Streams

Readable Streams	Writable Streams
HTTP responses, on the client	HTTP requests, on the client
HTTP requests, on the server	HTTP responses, on the server
fs read streams	fs write streams
zlib streams	zlib streams
crypto streams	crypto streams
TCP sockets	TCP sockets
child process stdout and stderr	child process stdin
process.stdin	process.stdout, process.stderr

All streams are EventEmitters

Streams example

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

// Stream will read the file in chunks
// if file size is bigger than the chunk then it will read a chunk and emit a
'data' event.
// Use encoding to convert data to String of hex
// Use highWaterMark to set the size of the chunk

var readable = fs.createReadStream(path.join(__dirname, 'sourceFile.txt'),
                                   { encoding: 'utf8', highWaterMark: 16 * 1024 });
var writable = fs.createWriteStream(path.join(__dirname, 'destinationFile.txt'));

readable.on('data', function(chunk) {
                                   console.log(chunk.length);
                                   writable.write(chunk);
                    });
```

highWaterMark - specifies the total number of bytes/objects (buffer)

Pipes: src.pipe(dst);

To connect two streams, Node provides a method called pipe() available on all readable streams. Pipes hide the complexity of listening to the stream events.

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

var readable = fs.createReadStream(__dirname + 'sourceFile.txt');
var writable = fs.createWriteStream(__dirname + 'destinationFile.txt');


readable.pipe(writable);

// note that pipe return the destination, this is why I can pipe it again to another stream if the destination was readable in this case it has to be DUPLEX because I'm going to write to it first, then read it and pipe it again to another writable stream.
```

Zip file using streams

```
var fs = require('fs');
var zlib = require('zlib');
var gzip = zlib.createGzip();
// this is a readable & writable stream and it returns a zipped stream

var readable = fs.createReadStream(__dirname + 'source.txt');
var compressed = fs.createWriteStream(__dirname + 'destination.txt.gz');

readable.pipe(gzip).pipe(compressed);
```

A key goal of the stream API, is to limit the buffering of data to acceptable levels such that sources and destinations of differing speeds will not overwhelm the available memory.

Node as a Web Server

Node started as a Web server and evolved into a much more generalized framework.

Node **`http`** core module is designed with streaming and low latency in mind.

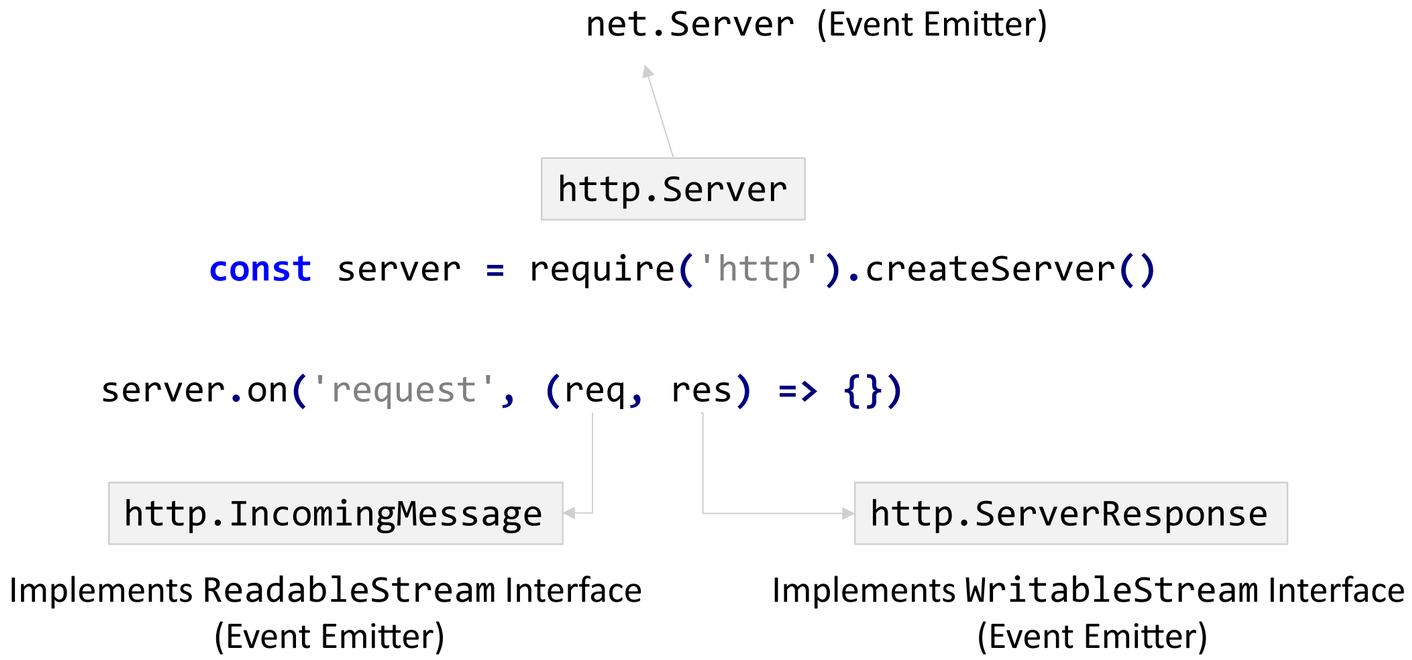
Node is very popular today to create and run Web servers.

PREV NEXT

OUTLINE

Search...

Class Structure



Web Server

```
var http = require('http');
var server = http.createServer();

server.on('request', function(req, res) {
                        res.writeHead(200, {'Content-Type': 'text/plain'});
                        res.write('Hello World!');
                        res.end();
                    });
server.listen(4000);
```

Node treats TCP Packets as Stream

After we run this code. The node program doesn't stop.. it keeps waiting for requests. Why?

Web Server Shortcut

```
require('http').createServer(function(req, res) {
                        res.writeHead(200, {'Content-Type': 'text/plain'});
                        res.end('Hello World!');
}).listen(4000, ()=> console.log('listening on 4000'));
```

Passing a callback function to createServer() is a shortcut for listening to "request" event.

Create HTTPS Server

```
const fs = require('fs');
const server = require('https')
        .createServer({
                key: fs.readFileSync('./key.pem'),
                cert: fs.readFileSync('./cert.pem'),
        });
server.on('request', (req, res) => {
        res.writeHead(200, { 'content-type': 'text/plain' });
        res.end('Hello from my HTTPS Web server!\n');
});

server.listen(443);
```

To create a key and certificate for your server you may use OpenSSL:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -node
```

https://blog.didierstevens.com/2015/03/30/howto-make-your-own-cert-with-openssl-on-windows/

Reactive Server with RxJs

```
const { Subject } = require('rxjs');
const subject = new Subject();

function sendHello(reqres) {
        reqres.res.end('Hello \n');
}

subject.subscribe(sendHello)
subject.subscribe(FilterIP)
subject.subscribe(LogToDB)

const http = require('http');
http.createServer((req, res) => {
        subject.next({ req: req, res: res });
}).listen(1337, ()=> console.log('127.0.0.1'));
```

The benefit of using a reactive server is that many components can listen to the subject and update themselves accordingly

Notice an interesting fact about the variables in the example: all variables (streams) were declared using **const**. This is very common in reactive programming, it is the events that move inside the streams, invisible to us.

Reactive benefits

We can add logging, buffer, filter, merge and many more operations on events, because of the large number of operations in RxJS library.

We can handle errors and memory deallocation easily.

Reactive programming makes reasoning about asynchronous events, controlling their timing, and combining them simpler. It is much simpler than using callbacks, and even simpler than using Promises. Since the server logic naturally deals with multiple identical request events, a stream where the events can flow and be processed is a better mental model compared to single-execution Promise abstraction.

Building an API and send JSON

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

http.createServer(function(req, res) {
                res.writeHead(200, { 'Content-Type': 'application/json' });
                var obj = {
                        firstname: 'Asaad',
                        lastname: 'Saad'
                };
                res.end(JSON.stringify(obj));
}).listen(1337, ()=> console.log('listening on 1337'));
```

Send out an HTML file

```
var http = require('http');
var fs = require('fs');
http.createServer(function(req, res) {
                res.writeHead(200, { 'Content-Type': 'text/html' });
                var html = fs.readFileSync(__dirname + 'index.html', 'utf8');
                var message = 'Hello CS572!';
                html = html.replace('{Message}', message);
                res.end(html);
}).listen(1337, ()=> console.log('listening on 1337'));
```

What's the problem with the code above?

```
<html>
        <head></head>
        <body>
                <h1>{Message}</h1>
        </body>
</html>
```

index.html

What can we do?

```
require('http').createServer(function(req, res) {
        var src = fs.createReadStream('/path/to/big/file');
        src.on('data', function(data) {
                if (!res.write(data)) {
                        src.pause();
                }
        });
        res.on('drain', function() {
                src.resume();
        });
        src.on('end', function() {
                res.end();
        });
});
```

Seriously!

A Simpler solution

```
var http = require('http');
var fs = require('fs');

http.createServer(function(req, res) {
   var rs = fs.createReadStream('/big/file').pipe(res);
});
```

We can simply use stream.pipe(), which does exactly what we described.

util.promisify()

It converts a callback-based function to a Promise-based one.

```
const { promisify } = require('util');
const fs = require('fs');

const readFileAsync = promisify(fs.readFile);

readFileAsync('./path_to_file', {encoding: 'utf8'})
        .then((text) => { console.log('CONTENT:', text); })
        .catch((err) => { console.log('ERROR:', err); });
```

Using util.promisify with async/await

```
const {promisify} = require('util');
const fs = require('fs');

const readFileAsync = promisify(fs.readFile);

async function main() {
    try {
        const text = await readFileAsync('filePath', {encoding: 'utf8'});
        console.log('CONTENT:', text);
    } catch (err) {
        console.log('ERROR:', err);
    }
}

main();
```