# 5 Wielding Web Protocols

This chapter covers the following recipes

* Creating an HTTP server
* Receiving POST data
* Handling file uploads
* Making an HTTP POST request
* Communicating with WebSockets
* Creating an SMTP server

## Introduction

One of the great qualities of Node is the simplicity it provides around low-level system operations.

Unlike template-centric languages such as PHP or ASP we have fine grain controlled over the behavior we want without sacrificing easy content control.

With Node we can create the server, customize it, and deliver content all at the code level.

Starting with a focus on core API's and low-level implementation then working our way up to more complex protocols with third party libraries, this chapter demonstrates how to create various clients and servers in the "Application Layer" of the TCP/IP stack.

### From Protocols to Frameworks

This chapter focuses on Node's direct relationship with Network protocols. It's intended to develop understanding of fundamental concepts. For creating more extensive and enterprise focused HTTP infrastructure check out **Chapter 7. Working with Web Frameworks**.

## Creating an HTTP server

HTTP is the most prolific protocol in the Application Layer of the Internet Protocol Suite. Node comes bundled with the core http module which providers both client and server capabilities.

In this recipe we're going to create an HTTP server from scratch.

### Getting Ready

To keep things interesting, our server is actually going to be a RESTful HTTP server, so let's create a folder called rest-server, containing a file named index.js.

### How to do it

We only need one module, the http module.

Let's require it:

const http = require('http')

Now we'll define a host and port which our HTTP server will attach to:

const host = process.env.HOST || '0.0.0.0'  
const port = process.env.PORT || 8080

Next, let's create the actual HTTP server:

const server = http.createServer((req, res) => {  
 if (req.method !== 'GET') return error(res, 405)  
 if (req.url === '/users') return users(res)  
 if (req.url === '/') return index(res)  
 error(res, 404)  
})

In the request handling function we passed to the http.createServer method, we reference three other functions, error, users and index.

First let's write our route handling functions:

function users (res) {  
 res.end('{"data": [{"id": 1, "first\_name": "Bob", "second\_name": "Smith"}]}')   
}  
  
function index (res) {  
 res.end('{"name": "my-rest-server", "version": 0}')  
}

Next we'll write our error function:

function error (res, code) {  
 res.statusCode = code  
 res.end(`{"error": "${http.STATUS\_CODES[code]}"}`)  
}

Finally we'll tell our server to listen on the previously defined port and host.

server.listen(port, host)

We can now try out our server.

We start our server like so:

$ node index.js

Then in another terminal, we can use curl to check the routes:

$ curl http://localhost:8080/users  
{"data": [{"id": 1, "first\_name": "Bob", "second\_name": "Smith"}]}

$ curl http://localhost:8080/  
{"name": "my-rest-server", "version": 0}

### How it works

Node's core http module sits on top of the core net module (just as the HTTP protocol is layer over the TCP protocol).

When we call http.createServer it returns an object (which we call server) that has a listen method. The listen method, binds our server to a given port and host.

Our port and host assignments at the top of the file check process.env.PORT and process.env.HOST, before defaulting to port 8080 and host '0.0.0.0'. This is good practice since it allows us (or a deployment orchestrator) to inject the desired settings into our server at execution time.

The http.createServer method takes a function, this is known as the request handler. Every time a client makes a request of our server, this request handler function is called, and passed a Request (req) object and a Response (res) object.

Our request handler sets up logic paths based on the req.url and req.method properties. The first thing we do is check req.method, which holds the HTTP verb. In our case we're only supporting GET request, so anything other than GET receives a 405 response (Method not Allowed). From a puritanical perspective we should only be giving a 405 to any unsupported yet recognized HTTP method, and 400 (Bad Request) to nonsensical methods. However, for our purposes we simply don't care.

The url property of the Request object tends to be a misnomer, since it generally relates more to the relative path (the route) than the full URL. This is because in an HTTP request, a client will usually only specify the relative path (the full URL is known because the client has connected to the domain already at this point). Whether the requested route is / or /users we delegate to a specific route handling function, passing it the res object and then in each route handling function we call res.end with the desired content.

Finally if no conditional checks match, we never explicitly return from the request handler function, and reach the final error function call, which sends a 404 (Not Found) HTTP status code. The error function uses the http.STATUS\_CODES constants to map HTTP codes to their equivalent descriptions.

### There's more

How can we bind to a any free port? what about serving dynamic content by handling more complex URL patterns?

#### Bind to a random free port

To assign the server to a random free port, we simply set the port number to 0.

Let's copy the rest-server folder and call the new folder rest-server-random-port.

Now in the rest-server-random-port/index.js file let's change our port reference near the top of the file to the following:

const port = process.env.PORT || 0

Next we'll change our server.listen statement at the bottom of the file like so:

server.listen(port, host, () => console.log(JSON.stringify(server.address())))

We've added a third callback argument to server.listen. The server binding process is asychronous, so we won't know which port we're bound to immediately. The callback is triggered once the server has bound to a port, then we can use the server.address method to get the port, host and IP family (IPv4 or IPv6). We JSON.stringify the object returned from server.address, this way a deployment orchestrator could easily take the data and parse it, passing it to some kind of discovery server.

#### Dynamic Content

There's not much point in a static HTTP server written in Node. Node's strength lies in it's ability to rapidly serve dynamic content.

Let's add a small filtering feature to our server.

We'll begin by copying the main rest-server folder created in the main recipe to a new folder called rest-server-dynamic-content.

Let's modify the top of rest-server-dynamic-content/index.js to look like so:

const http = require('http')  
const qs = require('querystring')  
const url = require('url')

Next let's add a mocked out user list resource:

const userList = [  
 {'id': 1, 'first\_name': 'Bob', 'second\_name': 'Smith', type: 'red'},  
 {'id': 2, 'first\_name': 'David', 'second\_name': 'Clements', type: 'blue'}  
]

Next we'll rearrange the http.createServer request handler function to the following:

const server = http.createServer((req, res) => {  
 if (req.method !== 'GET') return error(res, 405)  
 if (req.url === '/') return index(res)  
 const {pathname, query} = url.parse(req.url)  
 if (pathname === '/users') return users(query, res)  
   
 error(res, 404)  
})

We've moved the / route logic above the /users route logic, because the conditional check against req.url is very cheap, we want to prioritize low cost routes and rather than penalize them with prior routes that require additional parsing.

We then use url.parse to break up req.url into an object. We're particularly interested in the pathname and query portions of the URL. Once we have a pathname we can do a direct comparison (no regular expressions required), and pass the query to the users route handling function. Notice we've modified the users function signature to additionally accept a query argument.

Let's now modify our users route handling function:

function users (query, res) {  
 const {type} = qs.parse(query)  
 const list = !type ? userList : userList.filter((user) => user.type === type)  
 res.end(`{"data": ${JSON.stringify(list)}}`)   
}

By default the query portion of a URL string that has been parsed with url.parse is a string. So to find the type argument in the quesystring we use qs.parse to convert it to an object, then use ES2015 deconstruction assign the type property to a type reference. Imagine we have a route /users?type=blue, our query parameter will be ?type=blue, and the result of passing that to qs.parse will be an object: {type: 'blue'}. This means our type reference will be the string: 'blue'. If there is no query string, or there's a query string with no type argument then the type value will be undefined (which is coerced to a falsey value in the ternary condition). If we have a type we filter our userList, or other set list to the entire userList. Finally we call JSON.stringify on our list array, as we pass the entire JSON payload to res.end.

We can try this out like so:

$ node index.js

$ curl http://localhost:8080/users?type=blue  
{"data": [{"id":2,"first\_name":"David","second\_name":"Clements","type":"blue"}]}

For bonus points, we can refine our code a little more.

The url.parse module can run the querystring parse function for us.

Let's copy rest-server-dynamic-content/index.js to rest-server-dynamic-content/terser-index.js to begin refining.

First we'll remove the querystring module, our require statements at the top of the file should look like so:

const http = require('http')  
const url = require('url')

Next in our http.createServer request handler function we alter the line where url.parse occurs to the following:

const {pathname, query} = url.parse(req.url, true)

We've added a second argument to url.parse, with a value of true.

Finally we no longer need to manually parse the querystring in our users route handling function, so we'll update users to:

function users ({type}, res) {  
 const list = !type ? userList : userList.filter((user) => user.type === type)  
 res.end(`{"data": ${JSON.stringify(list)}}`)   
}

This server will behave in exactly the same way.

### See also

* *Creating an Express web app* in **Chapter 7 Working with Web Frameworks**
* *Creating a Koa web app* in **Chapter 7 Working with Web Frameworks**
* *Creating a Hapi web app* in **Chapter 7 Working with Web Frameworks**
* *Receiving POST data* in this chapter

## Receiving POST data

If we want to be able to receive POST data, we have to instruct our server on how to accept and handle a POST request.

In a language where I/O blocking is the primary runtime behavior, accessing POST body data would be as straight forward as accessing a property.

For instance, in PHP we could access our POST values with $\_POST['fieldname'], the execution thread would block until an array value was filled.

Contrariwise, Node provides low level interaction with the asynchronous flow of HTTP data allowing us to interface with the incoming message body as a stream, leaving it entirely up to the developer to turn that stream into usable data.

### Streams [../info.png]

For more information on streams see **Chapter 4 Using Streams**

### Getting ready

Let's create a server.js file ready for our code, plus a folder called public with an HTML file inside called form.html.

The form.html file should contain the following:

<form method="POST">  
 <input type="text" name="userinput1"><br>  
 <input type="text" name="userinput2"><br>  
 <input type="submit">  
</form>

### How to do it

We'll provision our server for both GET and POST requests.

Let's start with GET by requiring the core http module and loading form.html into memory, which we'll then serve via http.createServer:

const http = require('http')  
const fs = require('fs')  
const path = require('path')  
const form = fs.readFileSync(path.join(\_\_dirname, 'public', 'form.html'))  
  
http.createServer((req, res) => {  
 if (req.method === 'GET') {  
 get(res)  
 return  
 }  
 reject(405, 'Method Not Allowed', res)  
}).listen(8080)  
  
function get (res) {  
 res.writeHead(200, {'Content-Type': 'text/html'})  
 res.end(form)   
}  
  
function reject (code, msg, res) {  
 res.statusCode = code  
 res.end(msg)  
}

We are synchronously loading form.html at initialization time instead of accessing the disk on each request. When creating servers in Node, initialization is the only time when it's a good idea to perform synchronous I/O.

If we navigate to http://localhost:8080 we'll be presented with a form.

But if we fill out the form and submit, we'll encounter a "Method Not Allowed" response. This is because the method attribute on our HTML form is set to POST. If the method is anything other than GET, our request handler (the function passed to http.createServer) will fall through to calling the reject function which sets the relevant status code and sends the supplied message via the res object.

Our next step is to implement POST request handling.

First we'll add the querystring module to our list of required dependencies at the top of the file. The top section of our server.js file should become:

const http = require('http')  
const fs = require('fs')  
const path = require('path')  
const form = fs.readFileSync(path.join(\_\_dirname, 'public', 'form.html'))  
const qs = require('querystring')

For safety we'll want to define a maximum request size, which we'll use to guard against payload size based DoS attacks:

const maxData = 2 \* 1024 \* 1024 // 2mb

Now we'll add a check for POST methods in the request handler, so our http.createServer calls looks like so:

http.createServer((req, res) => {  
 if (req.method === 'GET') {  
 get(res)  
 return  
 }  
 if (req.method === 'POST') {  
 post(req, res)  
 return  
 }  
 reject(405, 'Method Not Allowed', res)  
}).listen(8080)

Next, let's implement the post function that's called within the request handler:

function post (req, res) {  
 if (req.headers['content-type'] !== 'application/x-www-form-urlencoded') {  
 reject(415, 'Unsupported Media Type', res)  
 return  
 }  
 const size = parseInt(req.headers['content-length'], 10)  
 if (isNaN(size)) {  
 reject(400, 'Bad Request', res)  
 return  
 }  
 if (size > maxData) {  
 reject(413, 'Too Large', res)  
 return  
 }  
  
 const buffer = Buffer.allocUnsafe(size)  
 var pos = 0  
  
 req  
 .on('data', (chunk) => {  
 const offset = pos + chunk.length  
 if (offset > size) {  
 reject(413, 'Too Large', res)  
 return  
 }  
 chunk.copy(buffer, pos)  
 pos = offset  
 })  
 .on('end', () => {  
 if (pos !== size) {  
 reject(400, 'Bad Request', res)  
 return  
 }  
 const data = qs.parse(buffer.toString())  
 console.log('User Posted: ', data)  
 res.end('You Posted: ' + JSON.stringify(data))  
 })  
}

Notice how we check the Content-Type and Content-Size headers sent by the browser. In particular Content-Size is validated at several check-points, this is important for preventing various types of attack from DoS attacks to leaking deallocated memory.

Once the form is completed and submitted, the browser and terminal should present the data provided via the form.

### How it works

The http module sits on top of the net module (Node's TCP library) which in turn interacts with an internal C library called libuv. The libuv C library handles network socket input/output, passing data between the C layer and the JavaScript layer.

When we call http.createServer an object is returned which represents the HTTP server. We immediately call the listen method on the server object which instructs the http module to listen for incoming data on the supplied port (8080).

Every time data is received at the network socket layer, if the data is successfully translated into an HTTP request the http module creates an object representing the request (req) and response (res) then calls our supplied request handler passing it the request and response objects.

Our request handler checks the method property of the request object to determine whether the request is GET or POST, and calls the corresponding function accordingly, falling back to calling our reject helper function if the request is neither GET nor POST.

The get function uses writeHead to indicate a success code (200) and set the Content-Type header to inform the browser of the mime-type of our form content (text/html). The res object is a WriteStream, which have write and end methods. Our get function finishes by calling res.end passing it the cached form content, this simultaneously writes to the response stream and ends the stream, thus closing the HTTP connection.

The reject function sets the statusCode and similarly calls res.end with the supplied message.

Our post function implements the core objective of our server. The post function checks the Content-Type and Content-Size HTTP headers to determine that we can support the supplied values (we'll talk more about size validation shortly) and uses it to preallocate a buffer. The HTTP request object (req) is a Node stream, which inherits from the EventEmitter object. Readable streams constantly emit data events until an end event is emitted. In the data event listener we use the Buffer copy method to duplicate the bytes in each incoming chunk into our preallocated buffer and update the pos to chunk.length so the next data event starts from where we left off in the previous event.

When all the data is received from the client the end event will be triggered. Our end event listener converts the buffer to a string, passing it into qs.parse. This converts the POST data (which is in the format userinput1=firstVal&userinput2=secondVal) into an object. This object is logged our to the console, and serialized with JSON.stringify as it's passed to res.end and thus mirrored back to the user.

We cannot trust the client to reliably represent the size of the content, as this could be manipulated by an attacker so we take several measures to validate the Content-Size HTTP header. HTTP headers will always be in string format, so we use parseInt to convert from string to number. If the Content-Size header sent wasn't a number size would be NaN - in that case we send a 400 Bad Request response.

#### Web Frameworks

Node's core API provides a powerful set of primitives to build functionality as we see fit. Of course, this also means there's a lot of angles to think about. In Chapter 6 we'll be talking about Web Frameworks where the low-level considerations have been taken care of, allowing us to focus primarily on business logic.

If size is a number we pass it to Buffer.allocUnsafe which creates a buffer of the given size. The choice by Node core developers to put "unsafe" in name is deliberately alarming.

Buffer.allocUnsafe will create a buffer from deallocated (i.e. unlinked) memory. That means any kind of data might appear in a buffer created with allocUnsafe, potentially including highly sensitive data like cryptographic private keys. This is fine as long as there isn't some way of leaking previously deallocated memory to the client. By using it we accept the burden of ensuring that a malicious request can't leak the data. This is why in the end event listener we check that pos is equal to size. If it isn't then the request is ending prematurely, and the old memory in our buffer hasn't been fully overwritten by the payload. Without the size check in the end event listener internal memory could leak to the client.

We could use Buffer.alloc instead, which zero-fills the memory (overwrites the memory with 00 bytes) before handing the buffer back but Buffer.allocUnsafe is faster.

The other check against size is in the data event listener, where we make sure the payload size doesn't exceed the provided Content-Size. This scenario could be a malicious attempt to overload the memory of our server, resulting in a Denial Of Service attack.

### There's More

POST data can also be sent as JSON, let's take a look at how to receive POST requests with an application/json mime type.

#### Accepting JSON

REST architectures (among others) typically handle the application/json content type in preference to the application/x-www-form-urlencoded type. Generally this is due to the versatility of JSON as a multi-language interchange data format.

Let's convert our form and server to working with JSON instead of URL-encoded data.

We're going to use a third party module called fast-json-parse for safely and efficiently parsing the JSON.

To do this we'll have to initialize our folder with a package.json file and then install fast-json-parse.

Let's run the following on the command line

$ npm init -y  
$ npm install --save fast-json-parse

In the server.js file we need to add the following to our required dependencies at the top of the file:

const parse = require('fast-json-parse')

The first line of our post function should be changed to checking Content-Type for application/json, like so:

function post (req, res) {  
 if (req.headers['content-type'] !== 'application/json') {  
 reject(415, 'Unsupported Media Type', res)  
 return  
 }  
 /\* ... snip .. \*/  
}

The final step in converting our server.js file is to adjust the end event listener like so:

/\* ... snip .. \*/  
 .on('end', () => {  
 if (pos !== size) {  
 reject(400, 'Bad Request', res)  
 return  
 }  
 const data = buffer.toString()  
 const parsed = parse(data)  
 if (parsed.err) {  
 reject(400, 'Bad Request', res)  
 return   
 }  
 console.log('User Posted: ', parsed.value)  
 res.end('{"data": ' + data + "}")  
 })  
/\* ... snip .. \*/

Unfortunately HTML forms do not natively support POSTing in the JSON format, so we'll need to add a touch of JavaScript to public/form.html.

Let's add the following script tag to form.html, underneath the <form> element:

<script>  
 document.forms[0].addEventListener('submit', function (evt) {  
 evt.preventDefault()  
 var form = this  
 var data = Object.keys(form).reduce(function (o, i) {  
 if (form[i].name) o[form[i].name] = form[i].value  
 return o  
 }, {})  
 form.innerHTML = ''  
 var xhr = new XMLHttpRequest()  
 xhr.open('POST', '/')  
 xhr.setRequestHeader('Content-Type', 'application/json')  
 xhr.send(JSON.stringify(data))  
 xhr.addEventListener('load', function () {  
 var res  
 try { res = JSON.parse(this.response) } catch (e) {  
 res = {error: 'Mangled Response'}  
 }  
 form.innerHTML = res.error   
 ? res.error   
 : 'You Posted: ' + JSON.stringify(res.data)  
 })  
 })  
</script>

Our form and server should now largely behave in the same manner as the main recipe.

Except our frontend is a tiny Single Page App and JSON (the backbone of modern web architecture) is being used for communication between server and client.

### See also

* *Handling file uploads* in this chapter
* *Implementing Authentication* in **Chapter 7 Working with Web Frameworks**
* *Creating an HTTP server* in this chapter
* *Processing big data* in **Chapter 4 Using Streams**

## Handling file uploads

We cannot process an uploaded file in the same way we process other POST data. When a file input is submitted in a form, the browser embeds the file(s) into a multipart message.

Multipart was originally developed as an email format allowing multiple pieces of mixed content to be combined into one payload. If we attempted to receive the upload as a stream and write it to a file, we would have a file filled with multipart data instead of the file or files themselves.

We need a multipart parser, the writing of which is more than a recipe can cover. So we'll be using multipart-read-stream module which sits on top of the well-established busboy module to convert each piece of the multipart data into an independent stream, which we'll then pipe to disk.

### Getting Ready

Let's create a new folder called uploading-a-file and create a server.js file and an uploads directory inside that:

$ mkdir uploading-a-file  
$ cd uploading-a-file  
$ touch server.js  
$ mkdir uploads

We'll also want to initialize a package.json file and install multipart-read-stream and pump.

On the command line, inside the uploading-a-file directory we run the following commands:

$ npm init -y  
$ npm install --save multipart-read-stream pump

#### Streams

For more about streams (and why pump is essential) see the previous chapter, **Chapter 4 Using Streams**

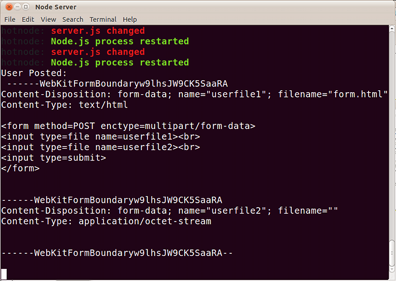
Finally we'll make some changes to our form.html file from the last recipe:

<form method="POST" enctype="multipart/form-data">  
 <input type="file" name="userfile1"><br>  
 <input type="file" name="userfile2"><br>  
 <input type="submit">  
</form>

We've included an enctype attribute of multipart/form-data to signify to the browser that the form will contain upload data and we've replaced the text inputs with file inputs.

To gain some understanding of how multipart requests differ from normal POST requests, let's use our newly modified form.html file with the server from the previous recipe to see how a server without multipart capabilities handles a multipart upload.

If we upload the form.html file itself we should see something like the following:



*Result of uploading multipart form data server from previous recipe*

Our original POST server simply logs the raw HTTP message body to the console, which in this case is multipart data.

We had two file inputs on the form. Though we only uploaded one file, the second input is still included in the multipart request. Each file is separated by a predefined boundary that is set in a secondary attribute of the Content-Type HTTP headers.

### How to do it

Let's set up our initial modules and load the form HTML as in our former recipe:

const http = require('http')  
const fs = require('fs')  
const path = require('path')  
const mrs = require('multipart-read-stream')  
const pump = require('pump')  
const form = fs.readFileSync(path.join(\_\_dirname, 'public', 'form.html'))

Next we'll set up the HTTP server, along with a GET handler function and a reject function for dealing with unsupported methods.

http.createServer((req, res) => {  
 if (req.method === 'GET') {  
 get(res)  
 return  
 }  
 if (req.method === 'POST') {  
 post(req, res)  
 return  
 }  
 reject(405, 'Method Not Allowed', res)  
}).listen(8080)  
  
function get (res) {  
 res.writeHead(200, {'Content-Type': 'text/html'})  
 res.end(form)   
}  
  
function reject (code, msg, res) {  
 res.statusCode = code  
 res.end(msg)  
}

Finally we'll write the post function:

function post (req, res) {  
 if (!/multipart\/form-data/.test(req.headers['content-type'])) {  
 reject(415, 'Unsupported Media Type', res)  
 return  
 }  
 console.log('parsing multipart data')  
 const parser = mrs(req.headers, part)  
 var total = 0  
 pump(req, parser)  
  
 function part (field, file, name) {  
 if (!name) {  
 file.resume()  
 return  
 }  
 total += 1  
 const filename = `${field}-${Date.now()}-${name}`  
 const dest = fs.createWriteStream(path.join(\_\_dirname, 'uploads', filename))  
 pump(file, dest, (err) => {  
 total -= 1  
 res.write(err  
 ? `Error saving ${name}!\n`  
 : `${name} successfully saved!\n`  
 )  
 if (total === 0) res.end('All files processed!')  
 })  
 }  
}

Now we have an upload server.

If we run

$ node server.js

Then open a browser at http://localhost:8080 we can upload some files, and check if our upload folder to see if they were added.

### How it works

Setting the enctype attribute of the HTML <form> element to multipart/form-data causes the browser to set the Content-Type in the request header to multipart/form-data and to embed data in any files supplied via the <input type="file"> elements into a multipart wrapper.

Our post function checks for the appropriate multipart/form-data content type in the req.headers object, rejecting the request with a 415 HTTP code (Unsupported Media Type) if the content type isn't multipart/form-data.

We create our parser by instantiating the multipart-request-stream module (mrs), passing it req.headers, and part (a function we declare shortly after).

We set up a total variable, which we use to track files from their point of discovery to when they've been completely written to disk.

The pump module is used to pipe data from the req object to the parser, this will cause our part function to be called each time the parser stream encounters a multipart boundary that contains file data.

In the part function we check to see that the name has a non-falsey value. This is because the browser will include all file fields even if they're not populated in the multipart data. If a file section has no name, then we can simply skip it. The file argument passed to part is a stream, in the event of an empty file section, we call the resume method (a stream method) to make the stream run to completion, allowing us to process the next file section in the multipart data.

Once we've verified the section has file data, we add one to total.

Then we create a filename based on the HTML elements field name, the current epoch time stamp and the original file name.

We create a write stream called dest that writes the incoming file into the uploads folder. We use pump again to pipe data from the file stream (as passed to the part function by the multipart-read-stream parser) to the dest stream.

This effectively writes a particular section of the multipart data to the uploads folder with the appropriate file. Using streams to do this means no matter how big the file is, memory and CPU usage will remain flat.

In the final parameter to the second call to pump, we provide a fat arrow callback. This will be called either in the event of an error or once all data has been written from the file stream into the dest stream. When the callback supplied to pump is called, we minus one from the total and write a message based on error state to the response stream.

When the total reaches 0 we know we've processed all the files in the multipart data and can end the response with a completion message.

### There's more

Multipart data doesn't just contain files, and file uploading from the browser isn't limited to multipart POST requests. Let's explore.

#### Processing all field types in multipart data

Multipart data can contain both files and field values.

Let's copy the uploading-a-file folder to a folder called processing-all-types.

Let's modify our public/form.html file by changing one of the file inputs to a text input:

<form method="POST" enctype="multipart/form-data">  
 <input type="text" name="userinput1"><br>  
 <input type="file" name="userfile2"><br>  
 <input type="submit">  
</form>

The multipart-read-stream module is a thin wrapper around the [busboy](http://npm.im/busboy) module, it listens to busboy for a file event and calls the user supplied function (which we called part in the main recipe).

Fortunately, multipart-read-stream returns the busboy instance, which also emits a field event. We can listen to this to process any non-file elements contained in the multipart data.

In server.js, let's add a field event listener directly under the assignment of our parser variable, making the top of our post function look like the following:

function post (req, res) {  
 if (!/multipart\/form-data/.test(req.headers['content-type'])) {  
 reject(415, 'Unsupported Media Type', res)  
 return  
 }  
 console.log('parsing multipart data')  
 const parser = mrs(req, res, part, () => {  
 console.log('finished parsing')  
 })  
 parser.on('field', (field, value) => {  
 console.log(`${field}: ${value}`)  
 res.write(`processed "${field}" input.\n`)  
 })  
 var total = 0  
 pump(req, parser)  
 /\* ... snip ... \*/  
}

#### Uploading files via PUT

Browsers are also capable of uploading files via an HTTP PUT request.

While we can only send one file per request, we don't need to do any parsing on the server side since we can simply stream the request contents directly to a file. This means less server-side processing overhead.

It would be magnificent if we could achieve this by changing our form's method attribute from POST to PUT, but alas, no, there is no specification for this.

However thanks to XMLHttpRequest Level 2 (xhr2), we can now transfer binary data via JavaScript in modern browsers (see http://caniuse.com/xhr2 (IE9 and Opera mini are lacking support)).

We can grab a file pointer using a change event listener on the input file element, and then we open a PUT request and send the file upon form submission.

Let's copy the uploading-a-file folder to a folder called uploading-a-file-with-put.

We won't be needing multipart-read-stream, but we will need the through2 module, so let's alter our dependencies accordingly:

$ npm i --save through2  
$ npm uninst --save multipart-read-stream

Next we'll modify our public/form.html file like so:

<form id="upload">  
 <input type="file" name="userfile1"><br>  
 <input type="submit">  
</form>  
<pre id="status"></pre>

We've added an id attribute to the form and the method and enctype attributes have been removed. We're also using just one file element because we can only send one file per request.

We've also added a <pre> tag which we'll be using to display status updates from the server.

In the same public/form.html file let's add an inline script at the end of the file:

<script>  
(function () {  
 var fieldName = 'userfile1'  
 var field = document.querySelector('[name=' + fieldName + ']')  
 var uploadForm = document.getElementById('upload')  
 var status = document.getElementById('status')  
 var file  
 field.addEventListener('change', function () {  
 file = this.files[0]  
 })  
 uploadForm.addEventListener('submit', function (e) {  
 e.preventDefault()  
 if (!file) return  
 var xhr = new XMLHttpRequest()  
 xhr.file = file  
 xhr.open('put', window.location, true)  
 xhr.setRequestHeader("x-field", fieldName)  
 xhr.setRequestHeader("x-filename", file.fileName || file.name)  
 xhr.onload = updateStatus  
 xhr.send(file)  
 file = ''  
 uploadForm.reset()  
 })  
 function updateStatus() {  
 status.innerHTML += this.status === 200   
 ? this.response  
 : this.status + ': ' + this.response  
 }  
}())  
</script>

Our script attaches a change listener to the file input element.

When the user selects a file, we the context (this) of the handler function for the change listener to take the first file in the files array (this.files).

Once a user submits the form our submit listener prevents default behavior (stops the browser from automatically submitting), checks whether a file is selected (doing nothing if no file has been selected), initializes an xhr object and opens a PUT request to our server. Then we set two custom headers, x-field and x-filename, we'll use these in our server.js file to determine the name of the input field and the original file name on the clients file system.

We set the onload method to our updateStatus function which will append responses from the server to our <pre> tag.

Finally we use the send method to initiate the PUT request and clean up by clearing the file variable and resetting our form.

Let's modify the top of our server.js file as follows:

const http = require('http')  
const fs = require('fs')  
const path = require('path')  
const pump = require('pump')  
const through = require('through2')  
const form = fs.readFileSync('public/form.html')  
const maxFileSize = 51200

We've removed the multipart-read-stream dependency, added through2 and created a maxFileSize constant.

Let's modify http.createServer response handler like so:

http.createServer((req, res) => {  
 if (req.method === 'GET') {  
 get(res)  
 return  
 }  
 if (req.method === 'PUT') {  
 put(req, res)  
 return  
 }  
 reject(405, 'Method Not Allowed', res)  
}).listen(8080)

We've simply changed the check for the POST method to a check for PUT method and call a put function instead of a post function.

Finally we need to remove the old post function, and replace it with the following put function:

function put (req, res) {  
 const size = parseInt(req.headers['content-length'], 10)  
 if (isNaN(size)) {  
 reject(400, 'Bad Request', res)  
 return  
 }  
 if (size > maxFileSize) {  
 reject(413, 'Too Large', res)  
 return  
 }  
  
 const name = req.headers['x-filename']  
 const field = req.headers['x-field']  
 const filename = `${field}-${Date.now()}-${name}`  
 const dest = fs.createWriteStream(path.join(\_\_dirname, 'uploads', filename))  
 const counter = through(function (chunk, enc, cb) {  
 this.bytes += chunk.length  
 if (this.bytes > maxFileSize) {  
 cb(Error('size'))  
 return  
 }  
 cb(null, chunk)  
 })  
 counter.bytes = 0  
 counter.on('error', (err) => {  
 if (err.message === 'size') reject(413, 'Too Large', res)  
 })  
 pump(req, counter, dest, (err) => {  
 if (err) return reject(500, `Error saving ${name}!\n`, res)  
 res.end(`${name} successfully saved!\n`)  
 })  
}

After some Content-Length checks we grab the original filename and field as supplied through the HTTP headers and construct a filename in similar fashion to our output filenames in the main recipe. Also similar to the main recipe, we create a dest stream.

For safety we also use the through2 module to create a byte counting stream. This could be important, since a malicious client could lie about Content-Length and send a much larger payload. We keep a running total of bytes passing through our stream, if they exceed the maximum we send an error down the stream.

As outlined in *Chapter 3 Using Streams* the pump module will propagate errors to all streams in the pipeline. We need to catch an error on the counter stream before pump does that, since it will close the req stream which will implicitly end the response. So we listen directly for an error event on the counter stream and send a 413: Too Large response within the error event handler.

Then we set up a pipeline from req to counter to our output dest, when the pipeline ends we send a success message, or failure if some other error has occurred.

### See also

* *Receiving POST data* in this chapter
* *Implementing authentication* in **Chapter 7 Working with Web Frameworks**
* *Creating an HTTP server* in this chapter
* *Piping streams in production* in **Chapter 4 Using Streams**

## Making an HTTP POST request

Making a GET request with Node is trivial, in fact HTTP GET requests have been covered in **Chapter 4 Using Streams** in the context of stream processing.

HTTP GET requests are so simple we can fit a request that prints to stdout into a single shell command (the -e flag passed to the node binary instructs node to evaluate the string that follows):

$ node -e "require('http').get('http://example.com', (res) => res.pipe(process.stdout))"

In this recipe we'll look into constructing POST requests.

### Getting Ready

Let's create a folder called post-request, then create an index.js inside the folder and open it in our favorite text editor.

### How to do it

We only need one dependency, the core http module. Let's require it:

const http = require('http')

Now let's define a payload we wish to POST to an endpoint:

const payload = `{  
 "name": "Cian Ó Maidín",  
 "company": "nearForm"  
}`

Now we'll define the configuration object for the request we're about to make:

const opts = {  
 method: 'POST',  
 hostname: 'reqres.in',  
 port: 80,  
 path: '/api/users',  
 headers: {  
 'Content-Type': 'application/json',  
 'Content-Length': Buffer.byteLength(payload)  
 }  
}

Notice how we use Buffer.byteLength to determine the Content-Length header.

#### [reqres.in](http://reqres.in)

We're using [reqres.in](http://reqres.in), a dummy REST API provided as a public service. The endpoint we're posting to simply mirrors the payload back in the response.

Next we'll make the request, supplying a callback handler which will be called once the request has completed:

const req = http.request(opts, (res) => {  
 console.log('\n Status: ' + res.statusCode)  
 process.stdout.write(' Body: ')  
 res.pipe(process.stdout)  
 res.on('end', () => console.log('\n'))  
})

Let's not forget to handle errors:

req.on('error', (err) => console.error('Error: ', err))

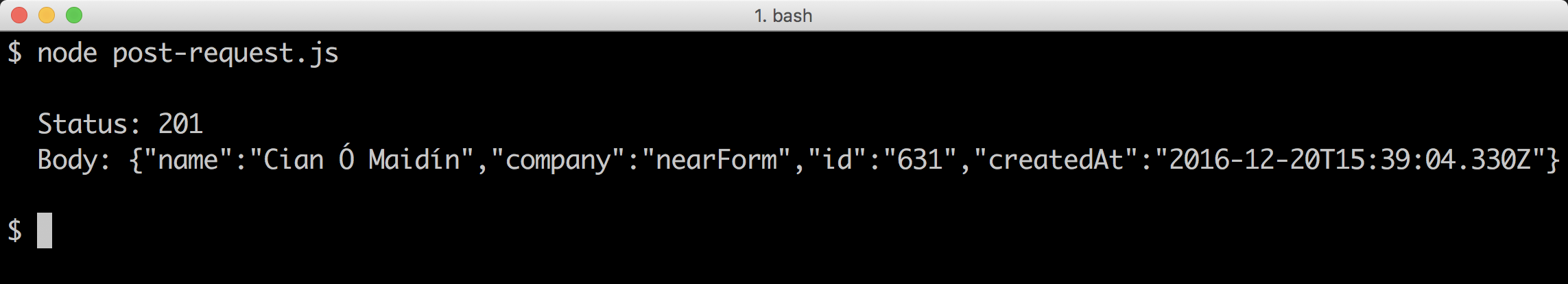
Finally the most important part, sending the payload.

req.end(payload)

Now if we execute our script:

$ node index.js

Providing the website [reqres.in](http://reqres.in) is functioning correctly, we should see something like the following:

 [**reqres.in**](http://reqres.in) **will simply mirror the posted payload**

### How it works

The http module provides both server and client capabilities.

We use the http.request method to create a request stream, which takes an options object (opts) describing the request and a callback.

#### HTTPS Requests

If the endpoint is encrypted (e.g. a standard HTTPS endpoint) We simply swap out the http module for the https module, the rest of the code remains the same.

In the headers of the opts object we set Content-Type and Content-Length headers. While the request will still be successful without providing Content-Length it is good practice and allows the server to make informed assumptions about the payload. However, setting Content-Length to a number lower than the payload size will result in an error. This is why it's important to use Buffer.byteLength because this gives the exact size of the string in bytes, which can differ from the string length when unicode beyond the ascii range are in the string (since unicode characters can be from 1 to 4 bytes, but are treated as a single character where String.prototype.length is concerned).

The http.request method opens a socket that's connected to the endpoint described in opts and returns a stream (which we assign to req).

When we write to this stream (using req.end) data is posted to the endpoint and the underlying socket is closed (because we ended the stream),

Since req is a stream it also has an error event, which we listen to. This would be fired in the event of network or socket errors, whereas a server error response would be reflected in the response status code.

The fat arrow callback passed to the http.request method is passed a response object, which we call res. We output the status using res.statusCode. Then we write the "Body:" label to the process.stdout stream (console.log would add a newline, we don't want that in this case), followed by piping the res object to process.stdout. Finally we listen to the end event on the res object to add two final newlines to the output (console.log adds an additional newline).

### There's more

How would we parse an entire data set at once? Can we stream a payload to a POST endpoint? How would we go about making a multipart POST upload in Node?

#### Buffering a GET Request

Sometimes it may be necessary to receive response data in it's entirety before it can be parsed.

Let's create a folder called buffering-a-request with a fresh index.js file.

In the index.js file we write the following code:

const http = require('http')  
const assert = require('assert')  
const url = 'http://www.davidmarkclements.com/ncb3/some.json'  
  
http.get(url, (res) => {  
 const size = parseInt(res.headers['content-length'], 10)  
 const buffer = Buffer.allocUnsafe(size)  
 var index = 0  
 res.on('data', (chunk) => {  
 chunk.copy(buffer, index)  
 index += chunk.length  
 })  
 res.on('end', () => {  
 assert.equal(size, buffer.length)  
 console.log('GUID:', JSON.parse(buffer).guid)  
 })  
})

When we run our script:

$ node index.js

We should see a log message containing the guid property found in the JSON data set.

We use Buffer.allocUnsafe to preallocate a buffer, "unsafe" is in the name because it uses garbage memory instead of zero-filling it - this is more performant but the risk of leaking internal data should be understood (for instance if the content length was greater than the actual amount of data received). In our case everything happens locally, so it's not a problem.

Then we listen to the data event of the response object (res), each chunk that comes through the data event is a Buffer instance, we use the copy method to copy the contents of the buffer into our pre-allocated buffer, keeping a running total of bytes copied in the index variable, which is passed to the copy method as the offset argument.

When the response ends, (in the end event handler), we check that the servers provided Content-Length header matches the amount of data received by checking size with buffer.length. Finally we parse the whole payload with JSON.parse and grab the guid property from the subsequent object returned from JSON.parse.

#### Parsing remote data

In a production setting, we should never call JSON.parse without wrapping a try/catch block around it. This is because any invalid JSON will cause JSON.parse to throw, which will cause the process to crash. The [fast-json-parse](http://npm.im/fast-json-parse) supplies, safe, high performance and clean JSON parsing.

#### Streaming payloads

Since the instance returned from http.request is a writable stream, we can take an input stream and pipe it to the POST request as data.

In this case we want to notify the server that we'll be incrementally writing the request body to the server in chunks.

Let's copy the main recipe folder to a new folder, call it streaming-payloads.

Now we'll tweak the index.js file slightly.

Let's make the top of the file look accordingly:

const http = require('http')  
const opts = {  
 method: 'POST',  
 hostname: 'reqres.in',  
 port: 80,  
 path: '/api/users',  
 headers: {  
 'Content-Type': 'application/json',  
 'Transfer-Encoding': 'chunked'  
 }  
}

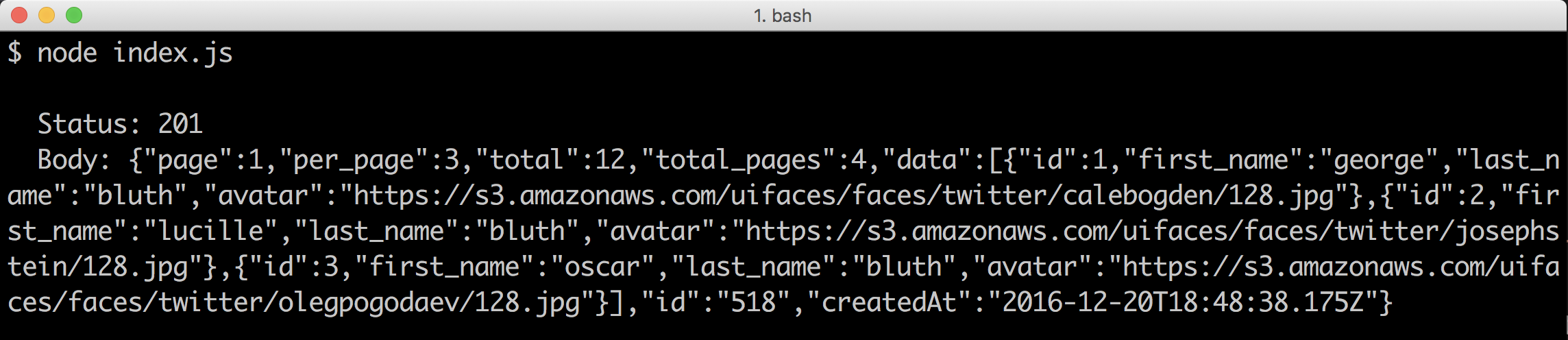
The payload assignment has been completely removed and we've replaced the Content-Length header with a Transfer-Encoding headed, set to chunked.

At the bottom of the file, we can replace the line req.end(payload) with:

http.get('http://reqres.in/api/users', (res) => {  
 res.pipe(req)  
})

We've initialized a stream of JSON from [reqres.in](http://reqres.in) (res) and piped it directly to request object (req).

When we run our script, we should see something like the following.



#### Multipart POST uploads

For fun and profit, let's build our own multipart request which we can post to the multipart upload server we created in the [Receiving POST data](#receiving-post-data) recipe (if we haven't completed that recipe now might be a good time to read up).

Let's create a new folder called multipart-post-uploads, with an index.js. We'll also initialize a package.json file and install steed (a low-overhead asynchronous control flow library).

$ mkdir multipart-post-uploads  
$ touch index.js  
$ npm init -y  
$ npm install --save steed

Now in the index.js file let's begin by setting up our dependencies:

const http = require('http')  
const fs = require('fs')  
const path = require('path')  
const steed = require('steed')()

Now for some configuration:

const files = process.argv.slice(2)  
const boundary = Date.now()  
const opts = {  
 method: 'POST',  
 hostname: 'localhost',  
 port: 8080,  
 path: '/',  
 headers: {  
 'Content-Type': 'multipart/form-data; boundary="' + boundary + '"',  
 'Transfer-Encoding': 'chunked'  
 }  
}

We're using process.argv (command line arguments), to determine which files to send in the multipart data. Our Content-Type header tells the server the multipart boundary that will designate each section in the data, and signify the end of the multipart data.

Next we'll set up the request:

const req = http.request(opts, (res) => {  
 console.log('\n Status: ' + res.statusCode)  
 process.stdout.write(' Body: ')  
 res.pipe(process.stdout)  
 res.on('end', () => console.log('\n'))  
})  
  
req.on('error', (err) => console.error('Error: ', err))

Finally we'll write the multipart data to the req stream, based on the files provided via the command line:

const parts = files.map((file, i) => (cb) => {  
 const stream = fs.createReadStream(file)  
 stream.once('open', () => {  
 req.write(  
 `\r\n--${boundary}\r\n` +  
 'Content-Disposition: ' +  
 `form-data; name="userfile${i}";` +  
 `filename="${path.basename(file)}"\r\n` +  
 'Content-Type: application/octet-stream\r\n' +  
 'Content-Transfer-Encoding: binary\r\n' +  
 '\r\n'  
 )  
 })  
 stream.pipe(req, {end: false})  
 stream.on('end', cb)  
})  
  
steed.series(parts, () => req.end(`\r\n--${boundary}--\r\n`))

The parts assignment is essentially a list of operations which we subsequently run in series using steed.series. To create this we map over each of the file paths, and return a function that takes a callback (cb).

Within that function, we create a read stream from the supplied file path. Read streams have an open event, which is triggered the moment the file is opened. We listen for this event and write the multipart header for the file. The header contains the field name (just called name in the header) - we set this simply to userfile plus the index of the file as it appeared in the command line arguments. A more sophisticated approach would involce mime-type detection and we'd be setting Content-Type in the multipart header appropriately. Since all data can be binary, we simplify by sending everything with the Content-Type of application/octet-stream and Content-Transfer-Type of binary.

Then we pipe from stream to req, passing an additional options argument to pipe, with an end property set to false. This will prevent req from being closed when stream finishes. Since we may have additional files, the req stream needs to stay open.

When stream ends we call cb. At this point, our flow control library steed, will process the next operation. When every function in the parts array has been processed by steed.series we end the request stream (req) with the multipart end boundary.

We can test this out by sending the source index.js twice to our multipart server from the [Receiving POST data](#receiving-post-data) recipe.

First let's start the upload server.

In the uploading-a-file folder, which was created in the [Receiving POST data](#receiving-post-data) recipe, we run:

$ node server.js

Then in a new terminal, in our multipart-post-uploads folder we run

$ node index.js package.json index.js

This will send the package.json file and the source index.js file to our server.

We can check the uploads folder in the uploading-a-file directory, to determine whether the index.js and package.json file were correctly uploaded.

### See also

* *Receiving POST data* in this chapter
* *Using the pipe method* in **Chapter 4 Using Streams**
* *Processing big data* in **Chapter 4 Using Streams**

## Communicating with WebSockets

HTTP was not made for the kind of real-time web applications that many developers are creating today. As a result, all sorts of workarounds have been discovered to mimic the idea of bidirectional, uninterrupted communication between servers and clients.

WebSockets don't mimic this behavior; they provide it.

In this recipe we will use the third-party ws module to create a pure WebSocket server that will receive and respond to WebSocket requests from the browser.

### Getting Ready

Let's create a new folder called websocket-app with a server.js file, plus a folder called public containing a index.html file.

$ mkdir websocket-app  
$ cd websocket-app  
$ touch server.js  
$ mkdir public  
$ touch public/index.html

We also need to initialize our app as a Node package, and install the ws module.

$ npm init -y   
$ npm install --save ws

### How to do it

In server.js let's begin by requiring our dependencies:

const http = require('http')  
const fs = require('fs')  
const ws = require('ws')

Next we want to load public/index.html into memory (we'll write index.html shortly), create an HTTP server, and then enhance it with a WebSocket server:

const app = fs.readFileSync('public/index.html')  
const server = http.createServer((req, res) => {  
 res.setHeader('Content-Type', 'text/html')  
 res.end(app)  
})  
const wss = new ws.Server({server})

Now that we have our WebSocket server instance (wss), we can listen to its connection event:

wss.on('connection', (socket) => {  
 socket.on('message', (msg) => {  
 console.log(`Received: ${msg}`)  
 console.log(`From IP: ${socket.upgradeReq.connection.remoteAddress}`)  
 if (msg === 'Hello') socket.send('Websockets!')  
 })  
})

Finally let's tell our HTTP server to listen on port 8080:

server.listen(8080)

Now for our frontend code.

Let's place the following in public/index.html

<input id="msg"><button id="send">Send</button>  
<div id="output"></div>

Now in the same public/index.html, we'll place our client side JavaScript.

At the bottom of public/index.html let's add the following:

<script>  
(function () {  
 var ws = new WebSocket('ws://localhost:8080')  
 var output = document.getElementById('output')  
 var send = document.getElementById('send')  
  
 function log (event, msg) {  
 return '<div>' + event + ': ' + msg + '</div>';  
 }  
  
 send.addEventListener('click', function () {  
 var msg = document.getElementById('msg').value  
 ws.send(msg)  
 output.innerHTML += log('Sent', msg)  
 })  
  
 ws.onmessage = function (e) {  
 output.innerHTML += log('Received', e.data)  
 }  
  
 ws.onclose = function (e) {  
 output.innerHTML += log('Disconnected', e.code + '-' + e.type)  
 }  
  
 ws.onerror = function (e) {  
 output.innerHTML += log('Error', e.data);  
 }  
}())  
</script>

Let's try it out by starting our server:

$ node server.js

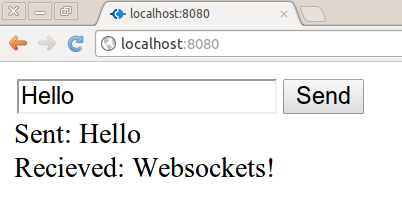
Then going to http://localhost:8080 in our browser, typing "Hello" in the textbox and clicking on the "Send" button.

The terminal should then output something like:

Received: Hello  
From IP: ::1

On other systems (where IPv4 is the default) the IP might be 127.0.0.1 instead.

Our browser should display something like the following:



### How it works

WebSocket servers start out as HTTP servers, then the browser connects to the HTTP server and asks to upgrade; at this point, the WebSocket logic takes over.

We supply an HTTP server instance to the WebSocket server at initialization time by passing it in the options object (we use ES6 shorthand to set a property of server with the value being the server instance).

When we navigate the browser to http://localhost:8080, an HTTP request is made and we send our in-memory public/index.html file as the response.

As soon as the HTML is loaded in the browser, the inline script is executed and the WebSocket upgrade request is made to our server.

When the server receives this WebSocket upgrade request, our WebSocket server instance (wss) emits a connection event that supplies socket as the first parameter of the connection event handler function.

The socket parameter is an instance of EventEmitter; we use its message event to respond to incoming messages on the established WebSocket connection.

In the message event handler function we log the received data and the client IP address to the terminal and check whether the incoming message is "Hello". If it is, we use the socket.send method to respond to the client with "WebSockets!".

### There's more

WebSockets have so much potential for efficient low latency real-time web apps. Let's take a look at a WebSocket client outside of the browser and then further see how browser APIs can be wrapped in one of Node's fundamental paradigms: streams.

#### Creating a Node.js WebSocket client

The ws module also allows us to create a WebSocket client outside of the browser environment.

Let's see if we recreate equivalent (and enhanced) functionality to our browser app in the terminal. Essentially we're going to create a generic interactive WebSocket testing command-line tool!

We'll start by creating a new folder called websocket-client with an index.js file. We'll also need to create a package.json and install the ws module

$ mkdir websocket-client  
$ cd websocket-client  
$ npm init -y  
$ npm install --save ws

Now let's open index.js in our favorite editor, and begin by requiring and initializing the ws module and the core readline module:

const WebSocket = require('ws')  
const readline = require('readline')  
const ws = new WebSocket(process.argv[2] || 'ws://localhost:8080')  
const rl = readline.createInterface({  
 input: process.stdin,  
 output: process.stdout,  
 prompt: '-> '  
})

Notice that we allow a command line argument to determine the WebSocket address, although we default to the address of the server in the main recipe.

Next let's set up some convenience references to ANSI terminal escape codes (we'll be using these to set colors in the terminal):

const gray = '\u001b[90m'  
const red = '\u001b[31m'  
const reset = '\u001b[39m'

Next we'll listen for an open event on our WebSocket client instance (ws), and output an appropriate status message when it fires:

ws.on('open', () => {  
 rl.output.write(`${gray}-- Connected --${reset}\n\n`)  
 rl.prompt()  
})

Notably, we use rl.output instead of process.stdout, this makes our implementation agnostic to the I/O streams, which theoretically allows for extensible and plug-able code.

The readline module allows us to take line by line input, so let's listen for the line event and send user input as a WebSocket message:

rl.on('line', (msg) => {  
 ws.send(msg, () => {  
 rl.output.write(`${gray}<= ${msg}${reset}\n\n`)  
 rl.prompt()  
 })  
})

If the message is successfully sent (using ws.send), the supplied callback (second argument passed to ws.send) is called. Here we output a status message confirming the user input was sent, and set up the prompt for further input.

WebSocket communication is of course bidirectional, so let's listen for any messages coming from the server:

ws.on('message', function (msg) {  
 readline.clearLine(rl.output)  
 readline.moveCursor(rl.output, -3 - rl.line.length, -1)  
 rl.output.write(`${gray}=> ${msg}${reset}\n\n`)  
 rl.prompt(true)  
})

Finally we'll finish by listening the close and error events, and outputting relevant status' to the terminal:

ws.on('close', () => {  
 readline.cursorTo(rl.output, 0)  
 rl.output.write(`${gray}-- Disconnected --${reset}\n\n`)  
 process.exit()  
})  
ws.on('error', (err) => {  
 readline.cursorTo(rl.output, 0)  
 rl.output.write(`${red}-- Error --${reset}\n`)  
 rl.output.write(`${red}${err.stack}${reset}\n`)  
})

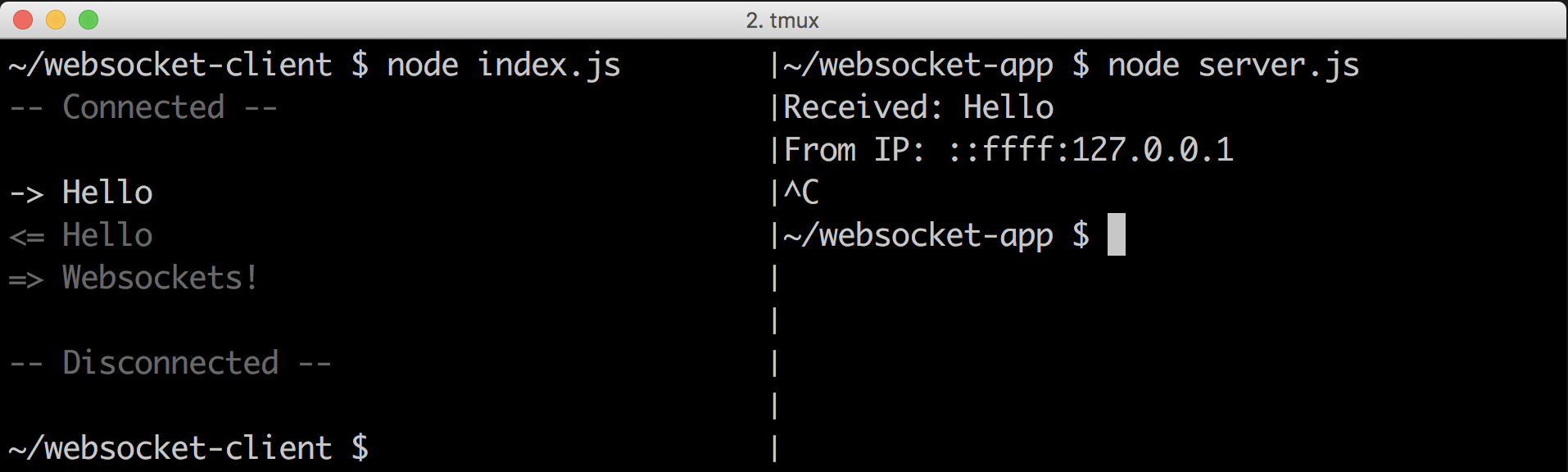
Lets start out server from the main recipe by running (from the websocket-app folder):

$ node server.js

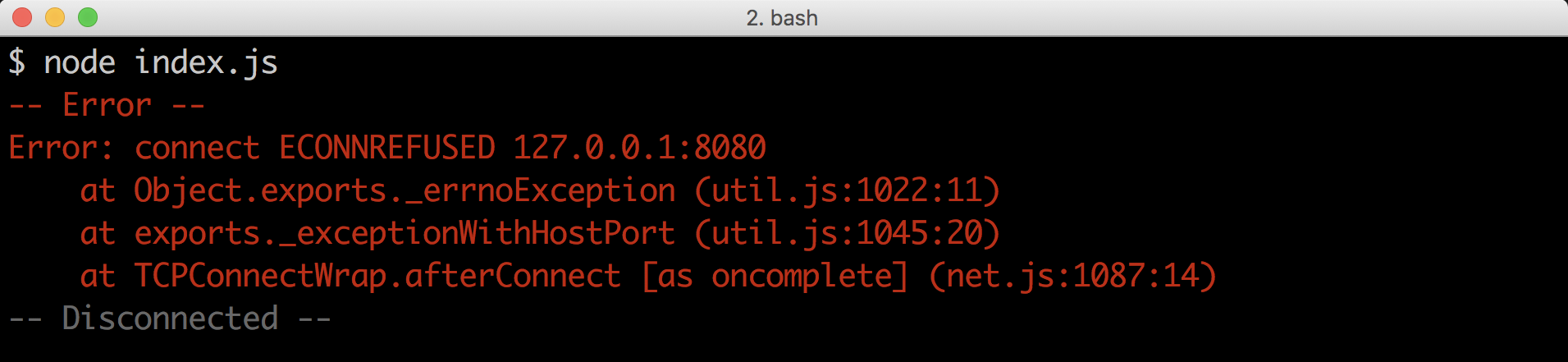
Then in a separate terminal, from the websocket-client folder, we can execute

$ node index.js

We should see an interaction illustrated in the following figure:



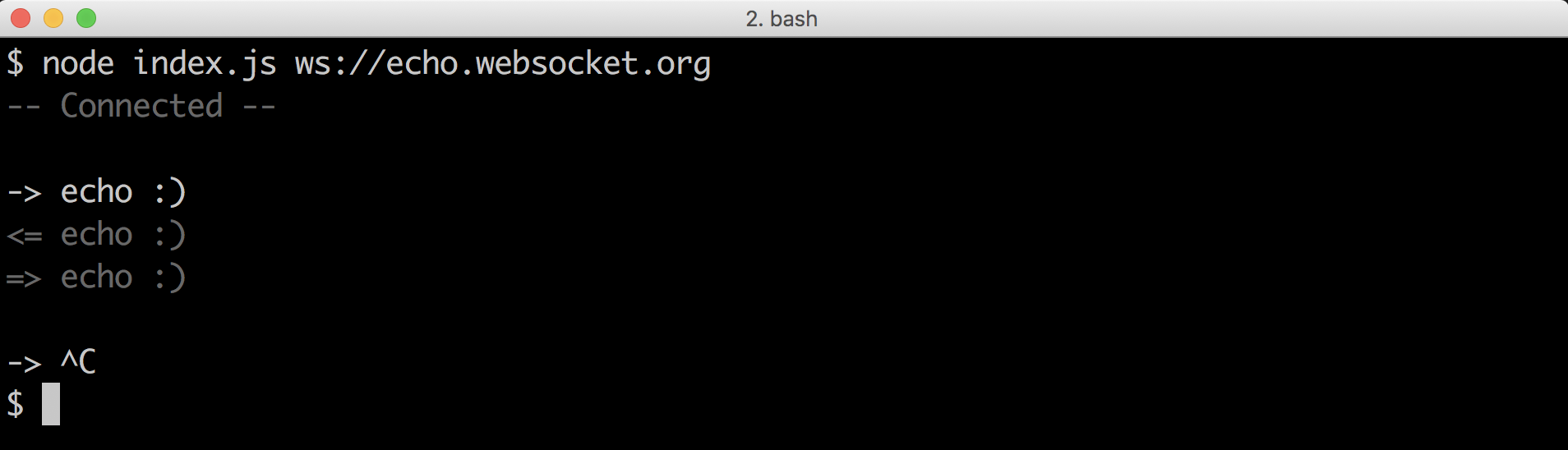
If we try to run our Node.js websocket client without the server started we'll see the error handler in effect:



Since we also allow command line input, we can easily connect to other WebSocket servers. For instance we could connect to ws://echo.websocket.org like so:

$ node index.js ws://echo.websocket.org

We can see the results of doing so, and typing "Echo :)" in the following figure:



The WebSocket protocol is not purely request-response oriented, so a message may come from the server without direct client interaction. We cater to this possibility in several ways.

Every non-terminal status (everything other than close or error status') is followed by two newlines. This gives us space to seamlessly inject a received message.  
When a message comes from the server we use readLine.clearline to wipe the current user prompt. Then we use readline.moveCursor to jump to the above line, and set the X position to the first column. We write the message out to rl.output (the terminal), and then call rl.prompt with true (this preserves current prompt content and cursor position).

### See also

* *Creating an HTTP server* in this chapter
* *Interfacing with standard I/O* in **Chapter 3 Coordinating I/O**
* *Processing big data* in **Chapter 4 Using Streams**

## Creating an SMTP server

In this recipe, we'll create our own internal SMTP server (just like the first SMTP servers) using the smtp-protocol module,

For information on converting an internal SMTP server to an externally exposed MX record server, see the *There's more...* section at the end of this recipe.

### Getting Ready

Let's create a folder called smtp-server, with an index.js file.

Then on the command-line from within the smtp-server directory, we can initialize our folder and install the smtp-protocol module:

$ npm init -y   
$ npm install --save smtp-protocol

### How to do it

Let's start by requiring relevant dependencies:

const fs = require('fs')  
const path = require('path')  
const smtp = require('smtp-protocol')

We're only going to accept emails to certain host domains, and only for certain users. We also need somewhere to store emails we have accepted.

So let's create two whitelist sets, one for hosts and the other users, along with a target path for mail boxes:

const hosts = new Set(['localhost', 'example.com'])  
const users = new Set(['you', 'another'])  
const mailDir = path.join(\_\_dirname, 'mail')

Before we create the server, we have some set up to perform. We need to make sure that the mail directory and user mail boxes exist on the file system, if they don't we need to create them.

Let's write the following code to do that:

function ensureDir (dir, cb) {  
 try { fs.mkdirSync(dir) } catch (e) {  
 if (e.code !== 'EEXIST') throw e  
 }  
}  
  
ensureDir(mailDir)  
for (let user of users) ensureDir(path.join(mailDir, user))

Next we'll create out SMTP server:

const server = smtp.createServer((req) => {  
 req.on('to', filter)  
 req.on('message', (stream, ack) => save(req, stream, ack))  
 req.on('error', (err) => console.error(err))  
})  
  
server.listen(2525)

This gives us an outline of what we'll do for each incoming SMTP request.

We still need to implement the filter and save functions shortly.

Our filter function will deconstruct the intended recipient email address and checks the hosts and users whitelist, calling accept or reject accordingly:

function filter (to, {accept, reject}) {  
 const [user, host] = to.split('@')  
 if (hosts.has(host) && users.has(user)) {  
 accept()  
 return  
 }  
 reject(550, 'mailbox not available')  
}

Finally our save function will take the incoming message and save it to any relevant mailboxes:

function save (req, stream, {accept}) {  
 const {from, to} = req  
 accept()  
 to.forEach((rcpt) => {  
 const [user] = rcpt.split('@')  
 const dest = path.join(mailDir, user, `${from}-${Date.now()}`)  
 const mail = fs.createWriteStream(dest)  
 mail.write(`From: ${from} \n`)  
 mail.write(`To: ${rcpt} \n\n`)  
 stream.pipe(mail, {end: false})  
 })  
}

Now if we run our index.js file our server should start (and create the relevant directory structures):

$ node index.js

We can manually test our mail server by opening a new terminal and running:

$ node -e "process.stdin.pipe(require('net').connect(2525)).pipe(process.stdout)"

This will allow us to interact with our SMTP server in real time, which means we can manually create an email message at the protocol level.

If we follow the below script we should be able to construct a message to our SMTP server:

helo  
mail from: me@me.com  
rcpt to: you@example.com  
data  
hello there!  
.  
quit

After each line we should receive a response from our server, the whole interaction should look similar to the following figure.



### How it works

SMTP is based upon a series of plain text communications between an SMTP client and server over a TCP connection. The smtp-protocol module carries out these communications for us.

When we call the createServer method, we pass an event handler that is passed a request object (req). This object is an event emitter (inherits from the core events module EventEmitter constructor). We listen for to, message, and error events.

The to event supplies two arguments to the handler callback (our filter function). The first argument is the full recipient address, the second (named ack) allows us to accept or reject the incoming message.

Our filter function uses parameter object deconstruction to pull out the accept and reject functions from the ack object and then uses assignment array deconstruction to define the user and host references. When we call split on the to string, splitting by the "at" (@) sign, we should have an array with two elements, using deconstruction, user points to index 0 of the resulting array while host points at index 1.

Our hosts and users whitelist are native Set objects, we use the has method to determine whether each side of the email address matches our criteria. If it does we call accept, if it doesn't we reject the recipient. At a protocol level a message prefixed with a 250 code (successful action) will be sent when we call accept. By default reject will respond with a code of 500 (command unrecognized), but we specify 550 (mailbox unavailable) which is more appropriate to the case.

The SMTP protocol allow for multiple recipients, until a message "data" message is sent. Each accepted recipient is added to an array stored on req.to (smtp-protocol builds this array internally). When a client sends a "data" message, the message event handler is fired.

The message event handler is passed two arguments, stream and ack. As with the to event handler, ack allows us to accept or reject. The stream object is a stream of the message body.

We call our save function in the message event handler, with the req, stream and ack objects.

The save function (like the filter function) deconstructs the ack object at the parameter level (we only need the accept function in this case). The to and from properties are pulled from the req object, also via deconstruction. Since we aren't performing any validating steps here, we can just immediately call the accept function.

The to reference is always an array (since there may be multiple recipients), we look through it with forEach and create a write stream (mail) pointing to the users mailbox, to a file name constructed from the from email address and a timestamp (dest). We write a simple header to the file, with "From" and "To" fields, and then pipe from our incoming message body stream (stream) to our file write stream (mail). When we call the pipe method we supply a second options argument, with end set to false. This is important when piping from a source stream to multiple destinations. Without passing this option, the first destination stream to finish would call end on the message body stream. Any other reading from the stream to other destination streams would cease, so data would be lost.

### There's more

Let's make an SMTP client to go with our SMTP server!

#### Making an SMTP client

The smtp-protocol module can be used for creating clients too, so let's make a client and use it alongside our SMTP server from the main recipe.

Let's copy the smtp-server folder, naming the duplicate smtp-client.

We don't need to install any new dependencies, so let's simply clear the contents of index.js and open it in our favorite editor.

Let's start with by requiring dependencies:

const os = require('os')  
const readline = require('readline')  
const smtp = require('smtp-protocol')

We're going to use the core readline module as a user interface, creating an interactive shell for sending mail. Let's create a readline interface instance:

const rl = readline.createInterface({  
 input: process.stdin,  
 output: process.stdout,  
 prompt: ''  
})

Now a config object to hold mail settings:

const cfg = {  
 host: 'localhost',  
 port: 2525,  
 email: 'me@me.com',  
 hostname: os.hostname()  
}

One more piece of housekeeping before we move on to the main task of connecting to our SMTP server. We're going to listen for the Ctrl + C key combination to cancel a writing an email, and quit the shell, and Ctrl + D to tell the interface we're done writing and ready to send.

The way to do this with the readline module is somewhat unintuitive. We listen for a close event to capture Ctrl + D and listen for SIGINT event to Ctrl + C (if we don't listen for SIGINT Ctrl + C would also trigger the close event).

So let's listen for SIGINT and respond accordingly:

rl.on('SIGINT', () => {  
 console.log('... cancelled ...')  
 process.exit()  
})

Now let's write the SMTP connection section:

smtp.connect(cfg.host, cfg.port, (mail) => {   
 mail.helo(cfg.hostname)  
 mail.from(cfg.email)  
 rl.question('To: ', (to) => {  
 to.split(/;|,/gm).forEach((rcpt) => mail.to(rcpt))  
 rl.write('===== Message (^D to send) =====\n')  
 mail.data(exitOnFail)  
 const body = []  
 rl.on('line', (line) => body.push(`${line}\r\n`))  
 rl.on('close', () => send(mail, body))  
 })  
})

We have yet to write the exitOnFail and send functions.

Let's start with send:

function send (mail, body) {  
 console.log('... sending ...')  
 const message = mail.message()  
 body.forEach(message.write, message)  
 message.end()  
 mail.quit()  
}

And finally exitOnFail:

function exitOnFail (err, code, lines, info) {  
 if (code === 550) {  
 err = Error(`No Mailbox for Recipient "${info.rcpt}"`)  
 }  
 if (!err && code !== 354 && code !== 250 && code !== 220 && code !== 200) {  
 err = Error(`Protocol Error: ${code} ${lines.join('')}`)  
 }  
 if (!err) return  
 console.error(err.message)  
 process.exit(1)  
}

Now if we have two terminals open, both with the current working directory set as the parent folder of both the smtp-server and smtp-client folders we can start our SMTP server like so

$ node smtp-server

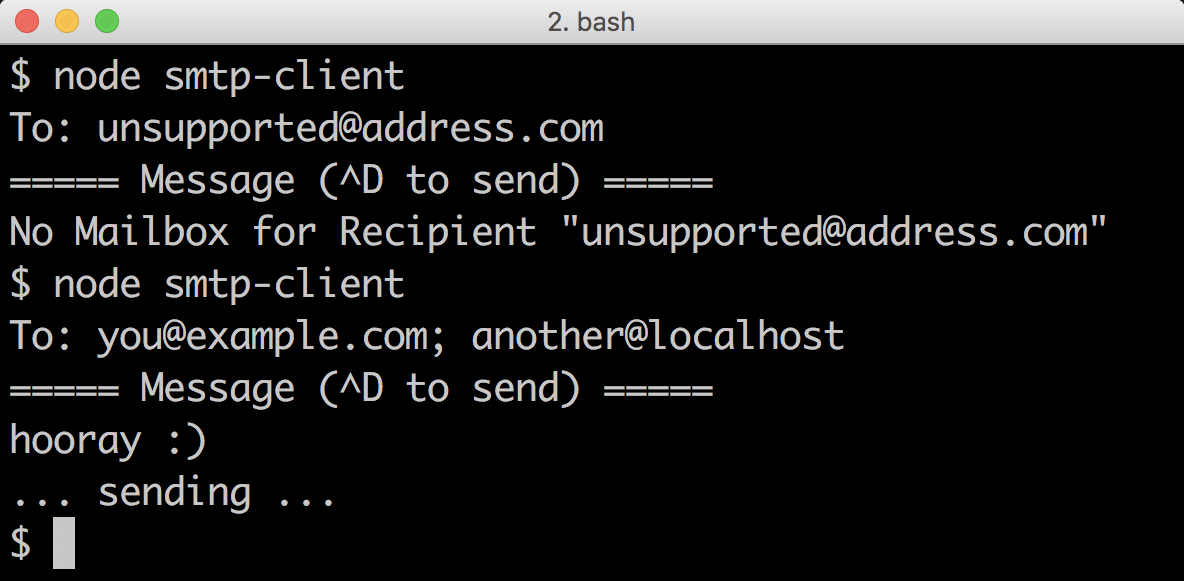
Then in the other terminal window, we can start our client thusly:

$ node smtp-client

If we supply a supported recipient address (such as "you@example.com") we should be able to successfully send mail to our SMTP server. We can also send to multiple addresses, delimiting either by comma (,) or semicolon (;).

If we try an unsupported address, we should see a message telling us that the server does not have an associated mailbox for the address specified.

The following figure shows interactions with the SMTP client:

 *Interacting with the SMTP client*

The smtp.connect method is analogous to the net connect method, it creates a client connection to our server.

Instead of supplying a socket instance (an object representing a TCP connection), it gives us a mail instance (an object representing an SMTP connection).

We interact with the protocol in a fairly direct way, calling the respective mail.helo and mail.from methods to send relevant protocol data across the wire.

Then we use our readline interface instance (rl) to prompt the user for recipients (reading a line of user input).

We split the result using a regular expression, allowing for both comma (,) and semicolon (;) delimited recipient lists.

We call mail.to for every recipient which will again send addresses to the SMTP server. The callback passed to mail.to in turn calls exitOnFail with an object containing the recipient (for detailed errors later on).

Then we instruct the user to write a message, and press Ctrl + D when finished. We use the line event on the rl instance to compose an array of lines (the body), then the close event (triggered on Ctrl + D) to pass the mail and body object to our send function.

The mail.message method returns a writable message stream for the body of the email (much like the message event on our server supplies a readable message stream).

We loop through the lines of our body, calling message.write for each line, then end the message stream and call mail.quit to end our SMTP session.

### See also

* *Using the pipe method* in **Chapter 4 Using Streams**
* *Interfacing with standard I/O* in **Chapter 3 Coordinating I/O**
* *Processing big data* in **Chapter 4 Using Streams**