Simple Instructions for a Working MEAN Starter App

The Basic API

(This will just be M-E-N for now. Angular or React to follow. But this will get you to the point of a working ReST API.)

> The basic API portion of this guide is a refinement, expansion, specialization, and in some cases an errata document for things found in a book called RESTful Web API Design with Node.JS 10. It's a good resource, but I did not find it to always be both concise and clear, and it wasn't always complete and correct. But it did explain concepts well and the process of debugging what appeared to be small errors or omissions caused me to learn more about what I was doing.
>
> The JWT authentication I begin to layer on to the API in the Authentication (second) section is a variation and step-by-step that borrows some from very nice work done by Jason Watmore and documented on his blog. You'll find some strong similarities in much of the starter code, which deviates as I modify those concepts to attach it to our existing API.
>
> The Application (third) section will start with an auto-generated vanilla React single-page app, which we'll then modify by borrowing some key concepts from Jason's working React front end so make some big, quick leaps to an authenticated application that uses our API to authenticate and retrieve some data.
>
> Everything we'll do will still start with a blinking cursor so we can build our understanding brick-by-brick. Our work will result in a guide that describes pretty fast way to get a mongo-backed browser application ready for prototyping.

Let's use Ubuntu. I'm going to use 20.04 LTS. I had an ISO already downloaded, so I used it to create a virtual machine on Vmware Workstation 15. I started at 9:30 AM with the creation of the VM and installed immediately after.

At about 9:40, I have a working VM. It's not really necessary for this exercise, but it's nice to give root a password as it makes some things easier later. So open a terminal:

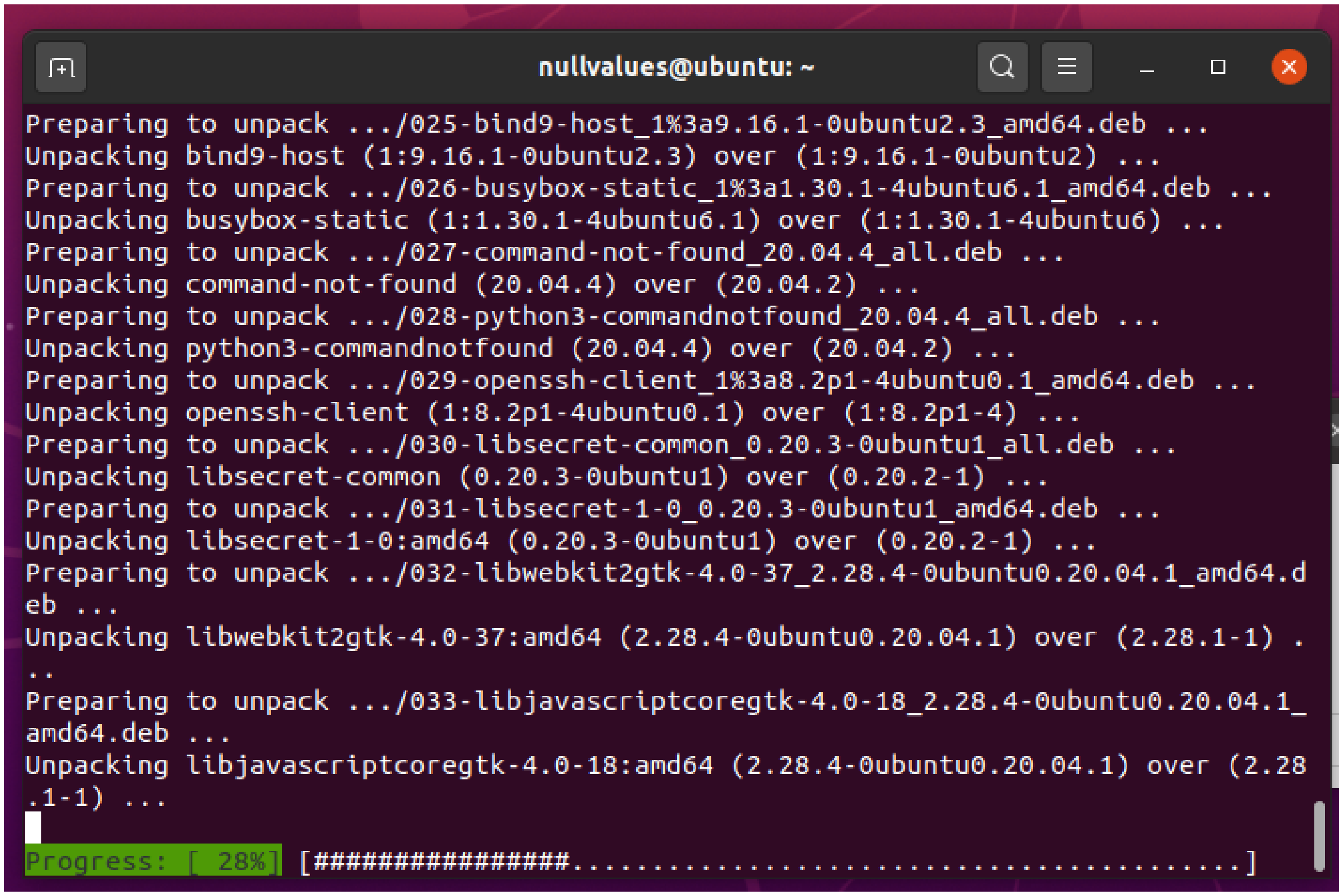
```
nullvalues@ubuntu:~$ sudo passwd root
[sudo] password for nullvalues:
New password:
Retype new password:
passwd: password updated successfully
nullvalues@ubuntu:~$
```

And let's patch this system right away.

```
nullvalues@ubuntu:~$ sudo apt update

does some updates, usually...
```

Then upgrade:

```
nullvalues@ubuntu:~$ sudo apt upgrade
```



```
Preparing to unpack .../025-bind9-host_1%3a9.16.1-0ubuntu2.3_amd64.deb ...
Unpacking bind9-host (1:9.16.1-0ubuntu2.3) over (1:9.16.1-0ubuntu2) ...
Preparing to unpack .../026-busybox-static_1%3a1.30.1-4ubuntu6.1_amd64.deb ...
Unpacking busybox-static (1:1.30.1-4ubuntu6.1) over (1:1.30.1-4ubuntu6) ...
Preparing to unpack .../027-command-not-found_20.04.4_all.deb ...
Unpacking command-not-found (20.04.4) over (20.04.2) ...
Preparing to unpack .../028-python3-commandnotfound_20.04.4_all.deb ...
Unpacking python3-commandnotfound (20.04.4) over (20.04.2) ...
Preparing to unpack .../029-openssh-client_1%3a8.2p1-4ubuntu0.1_amd64.deb ...
Unpacking openssh-client (1:8.2p1-4ubuntu0.1) over (1:8.2p1-4) ...
Preparing to unpack .../030-libsecret-common_0.20.3-0ubuntu1_all.deb ...
Unpacking libsecret-common (0.20.3-0ubuntu1) over (0.20.2-1) ...
Preparing to unpack .../031-libsecret-1-0_0.20.3-0ubuntu1_amd64.deb ...
Unpacking libsecret-1-0:amd64 (0.20.3-0ubuntu1) over (0.20.2-1) ...
Preparing to unpack .../032-libwebkit2gtk-4.0-37_2.28.4-0ubuntu0.20.04.1_amd64.d
eb ...
Unpacking libwebkit2gtk-4.0-37:amd64 (2.28.4-0ubuntu0.20.04.1) over (2.28.1-1) .
..
Preparing to unpack .../033-libjavascriptcoregtk-4.0-18_2.28.4-0ubuntu0.20.04.1_
amd64.deb ...
Unpacking libjavascriptcoregtk-4.0-18:amd64 (2.28.4-0ubuntu0.20.04.1) over (2.28
.1-1) ...

Progress: [ 28%] [################.......................................]
```

I usually install Chrome via Snap. It's easy to load from the terminal you have open.

```
nullvalues@ubuntu:~$ snap-store &
```

Search for Chromium and install. (You can also install non-open source Chrome from google.com. Both work fine.) Snap is also a good way to add your favorite IDE, such at Atom or Webstorm. I'll be using Webstorm. It's about 10:10 and I stopped to help troubleshoot a BST issue for about 10 minutes. But I'm back and we're ready to install node.

I took a break to do other work, starting at 11:30. I used this to install nodejs: https://github.com/nodesource/distributions/blob/master/README.md#debinstall

I didn't have curl yet, so I'll install it:

```
nullvalues@ubuntu:~$ sudo apt install curl
[sudo] password for nullvalues:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer
required:
  libfprint-2-tod1 libllvm9
Use 'sudo apt autoremove' to remove them.
The following NEW packages will be installed:
  curl
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 161 kB of archives.
After this operation, 411 kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu focal-updates/main amd64 curl
amd64 7.68.0-1ubuntu2.2 [161 kB]
Fetched 161 kB in 0s (333 kB/s)
Selecting previously unselected package curl.
(Reading database ... 179984 files and directories currently installed.)
Preparing to unpack .../curl_7.68.0-1ubuntu2.2_amd64.deb ...
Unpacking curl (7.68.0-1ubuntu2.2) ...
Setting up curl (7.68.0-1ubuntu2.2) ...
Processing triggers for man-db (2.9.1-1) ...
nullvalues@ubuntu:~$
```

Then install per the instructions for Debian / Ubuntu in the link:

```
curl -sL https://deb.nodesource.com/setup_14.x | sudo -E bash -
sudo apt install -y nodejs
```

The second line installs both nodejs and npm as indicated in comments at the end of the node source unpacking:

```
## Run `sudo apt-get install -y nodejs` to install Node.js 14.x and npm
## You may also need development tools to build native addons:
     sudo apt-get install gcc g++ make
```

You can check it:

```
nullvalues@ubuntu:~$ npm

Usage: npm <command>

where <command> is one of:
    access, adduser, audit, bin, bugs, c, cache, ci, cit,
    clean-install, clean-install-test, completion, config,
    create, ddp, dedupe, deprecate, dist-tag, docs, doctor,
    edit, explore, fund, get, help, help-search, hook, i, init,
    install, install-ci-test, install-test, it, link, list, ln,
    login, logout, ls, org, outdated, owner, pack, ping, prefix,
    profile, prune, publish, rb, rebuild, repo, restart, root,
    run, run-script, s, se, search, set, shrinkwrap, star,
    stars, start, stop, t, team, test, token, tst, un,
    uninstall, unpublish, unstar, up, update, v, version, view,
    whoami

npm <command> -h  quick help on <command>
npm -l            display full usage info
npm help <term>   search for help on <term>
npm help npm      involved overview

Specify configs in the ini-formatted file:
    /home/nullvalues/.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config

npm@6.14.8 /usr/lib/node_modules/npm
nullvalues@ubuntu:~$
```

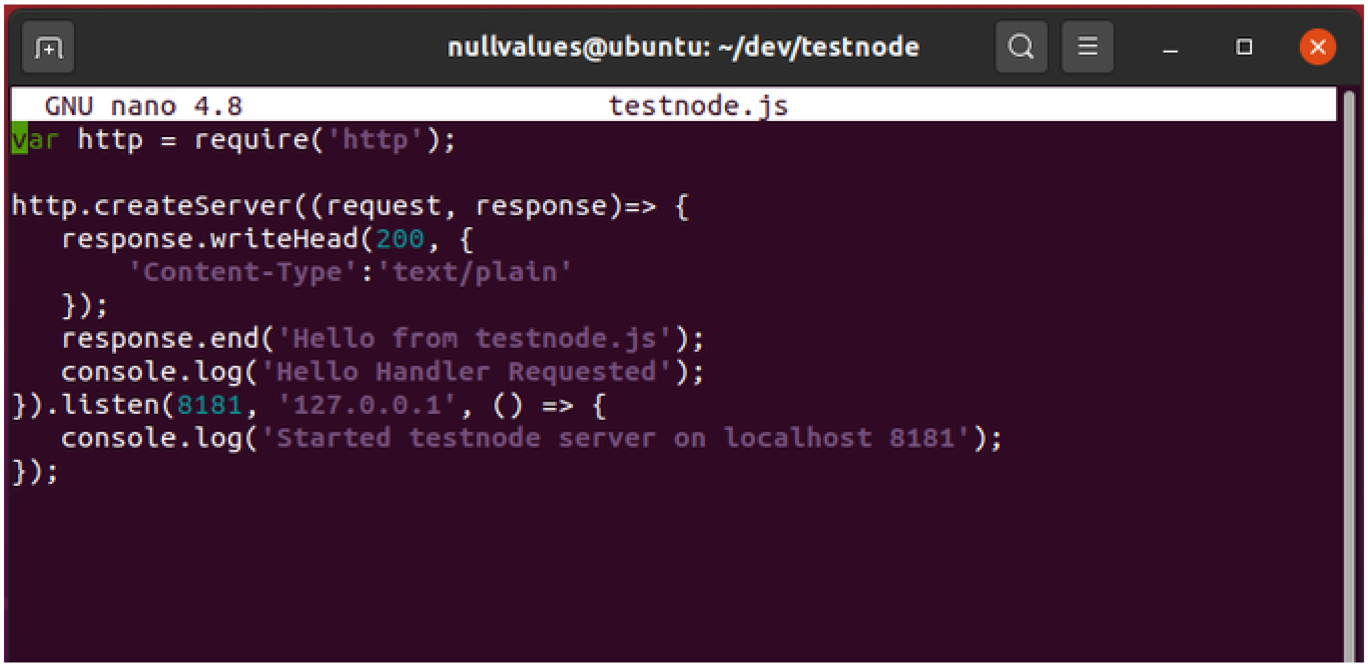
Let’s install the “E” in MEAN, and a common (though deprecated) unit test module:

```
nullvalues@ubuntu:~$ sudo npm install -g express nodeunit
npm WARN deprecated nodeunit@0.11.3: you are strongly encouraged to use
other testing options
npm WARN deprecated request@2.88.2: request has been deprecated, see
https://github.com/request/request/issues/3142
npm WARN deprecated har-validator@5.1.5: this library is no longer
supported
/usr/bin/nodeunit -> /usr/lib/node_modules/nodeunit/bin/nodeunit

> ejs@2.7.4 postinstall /usr/lib/node_modules/nodeunit/node_modules/ejs
> node ./postinstall.js

Thank you for installing EJS: built with the Jake JavaScript build tool
(https://jakejs.com/)

+ express@4.17.1
+ nodeunit@0.11.3
added 284 packages from 218 contributors in 10.941s
nullvalues@ubuntu:~$
```

Okay, let’s make a simple test app to see if things are working thus far. Make a directory for the app, I’m making mine /nullvalues/dev/testnode.

```
nullvalues@ubuntu:~/dev/testnode$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (testnode)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /home/nullvalues/dev/testnode/package.json:

{
  "name": "testnode",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes)
nullvalues@ubuntu:~/dev/testnode$ sudo npm install -g http --save
+ http@0.0.1-security
added 1 package in 0.235s
nullvalues@ubuntu:~/dev/testnode$
```

Add a simple text file to this directory called testnode.js so we can test the install. You can do it in nano without much fuss.



```
GNU nano 4.8                                testnode.js
var http = require('http');

http.createServer((request, response)=> {
   response.writeHead(200, {
       'Content-Type':'text/plain'
   });
   response.end('Hello from testnode.js');
   console.log('Hello Handler Requested');
}).listen(8181, '127.0.0.1', () => {
   console.log('Started testnode server on localhost 8181');
});
```

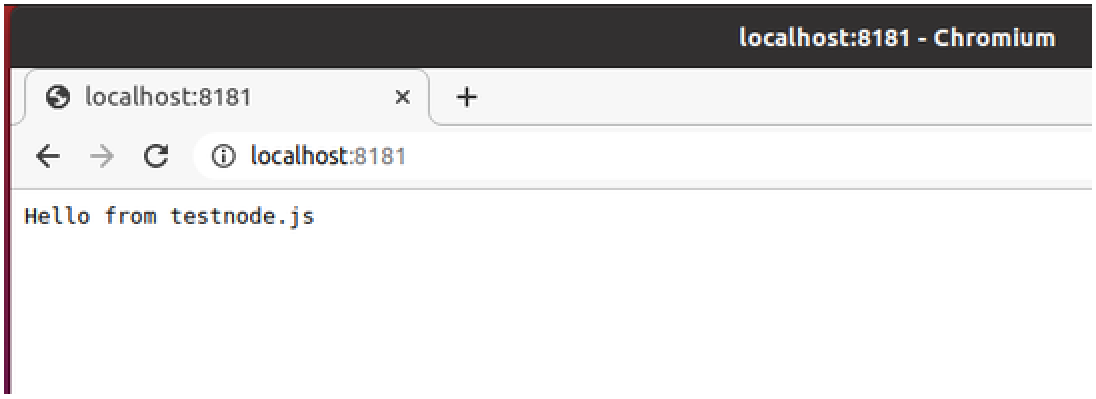
I used some boilerplate from the book I referenced earlier for this:

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

http.createServer((request, response)=> {
   response.writeHead(200, {
       'Content-Type':'text/plain'
   });
   response.end('Hello from testnode.js');
   console.log('Hello Handler Requested');
}).listen(8181, '127.0.0.1', () => {
   console.log('Started testnode server on localhost 8181');
});
```

Save and test your server:

```
nullvalues@ubuntu:~/dev/testnode$ node testnode.js
Started testnode server on localhost 8181
```

Here we are:



Hello from testnode.js

```
nullvalues@ubuntu:~/dev/testnode$ node testnode.js
Started testnode server on localhost 8181
Hello Handler Requested
```

Notice the request was logged to the console. That will go elsewhere later, but it's nice to see here for now and could easily be piped to a log file, like this:

```
nullvalues@ubuntu:~/dev/egt$ node testnode.js > log.txt
```

Then you can tail the log file elsewhere like this:

```
nullvalues@ubuntu:~/dev/egt$ tail -f -n 40 log.txt
Returning everything.
Connection to mongo has been called 1 times.
Creating new connection handle.
{"exes":[{"_id":"5f593ae99b22f4f49d94c6ad","x":1},{"_id":"
5f59645df36161c9b729d96c","x":202}]}
GET /api 304 36.514 ms - -
```

But more on that later. At this point we know that Node and NPM are working. It's 12:04, time for lunch anyway. Hit ctrl-c in your terminal to stop your server. You can delete that directory, if you like. Or keep it around to test basic functionality later. This is roughly an hour of elapsed work to this point.

It's 12:30. Lunch was light, but tasty. Thanks for asking.

Let's test the Express Generator. We're going to call it express-generator-test. So let's make a directory /nullvalues/dev/egt and run it there. (This site is useful for Express instructions.)

```
nullvalues@ubuntu:~/dev$ mkdir egt
nullvalues@ubuntu:~/dev$ cd egt
nullvalues@ubuntu:~/dev/egt$ npx express-generator
npx: installed 10 in 1.005s
  warning: the default view engine will not be jade in future releases
  warning: use `--view=jade' or `--help' for additional options

   create : public/
   create : public/javascripts/
   create : public/images/
   create : public/stylesheets/
   create : public/stylesheets/style.css
   create : routes/
   create : routes/index.js
   create : routes/users.js
   create : views/
   create : views/error.jade
   create : views/index.jade
   create : views/layout.jade
   create : app.js
   create : package.json
   create : bin/
   create : bin/www

   install dependencies:
     $ npm install

   run the app:
     $ DEBUG=egt:* npm start

nullvalues@ubuntu:~/dev/egt$ npm install
npm WARN deprecated jade@1.11.0: Jade has been renamed to pug, please
install the latest version of pug instead of jade
npm WARN deprecated constantinople@3.0.2: Please update to at least
constantinople 3.1.1
npm WARN deprecated transformers@2.1.0: Deprecated, use jstransformer
npm notice created a lockfile as package-lock.json. You should commit
this file.
added 100 packages from 139 contributors and audited 101 packages in
2.432s
found 4 vulnerabilities (3 low, 1 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
nullvalues@ubuntu:~/dev/egt$
```

There are two interesting things there:

- Jade has been replaced with Pug. Not sure what that means just yet, but we’ll learn.
- There’s an error we should sort out.

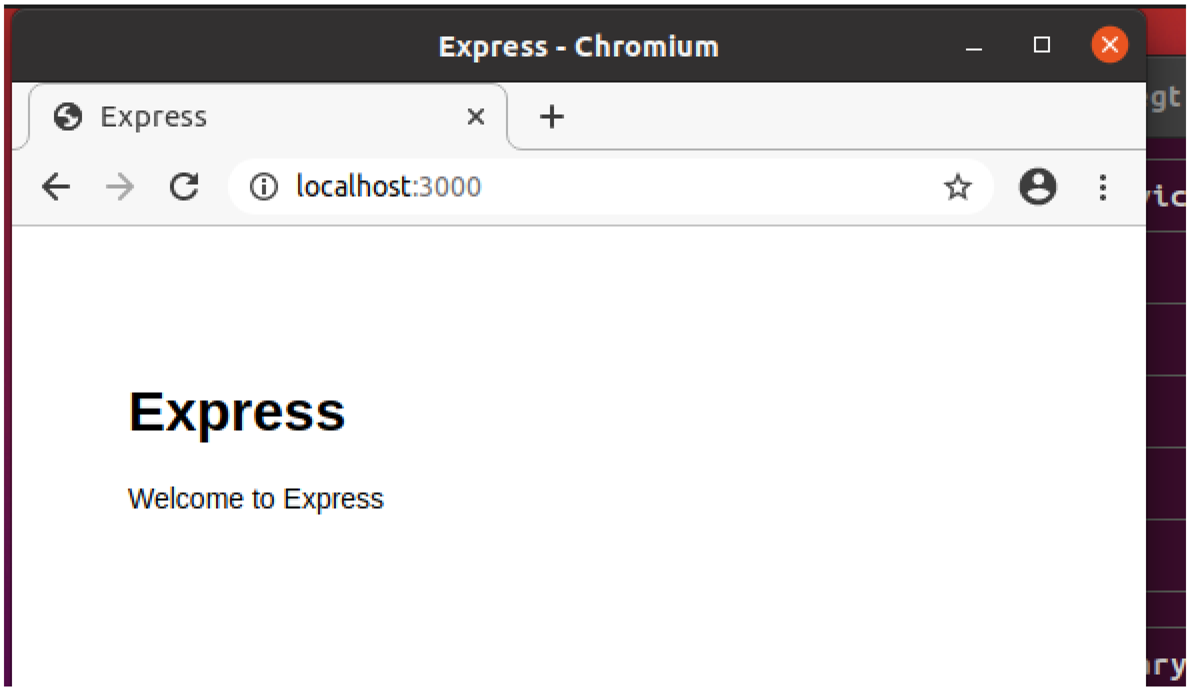
```
nullvalues@ubuntu:~/dev/egt$ npm install pug
+ pug@3.0.0
added 38 packages from 14 contributors and audited 139 packages in 2.464
s

3 packages are looking for funding
  run `npm fund` for details

found 4 vulnerabilities (3 low, 1 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
nullvalues@ubuntu:~/dev/egt$ npm audit fix
up to date in 0.518s

3 packages are looking for funding
  run `npm fund` for details

fixed 0 of 4 vulnerabilities in 139 scanned packages
  4 vulnerabilities required manual review and could not be updated
nullvalues@ubuntu:~/dev/egt$
```

That was half-promising. Let’s try to re-run the generator and see if the Jade error goes away.

```
nullvalues@ubuntu:~/dev/egt$ npx express-generator
npx: installed 10 in 0.925s
  warning: the default view engine will not be jade in future releases
  warning: use `--view=jade' or `--help' for additional options

destination is not empty, continue? [y/N] y
   create : public/
   create : public/javascripts/
   create : public/images/
   create : public/stylesheets/
   create : public/stylesheets/style.css
   create : routes/
   create : routes/index.js
   create : routes/users.js
   create : views/
   create : views/error.jade
   create : views/index.jade
   create : views/layout.jade
   create : app.js
   create : package.json
   create : bin/
   create : bin/www

   install dependencies:
     $ npm install

   run the app:
     $ DEBUG=egt:* npm start

nullvalues@ubuntu:~/dev/egt$
```

Looks better. The particular critical error seems to involve Sandbox Bypass Arbitrary Code Execution, upon inspection with npm audit, so we're going to leave this alone until we're close to production. It doesn't impact our simple setup. Let's see what we get now with an Express generated app:

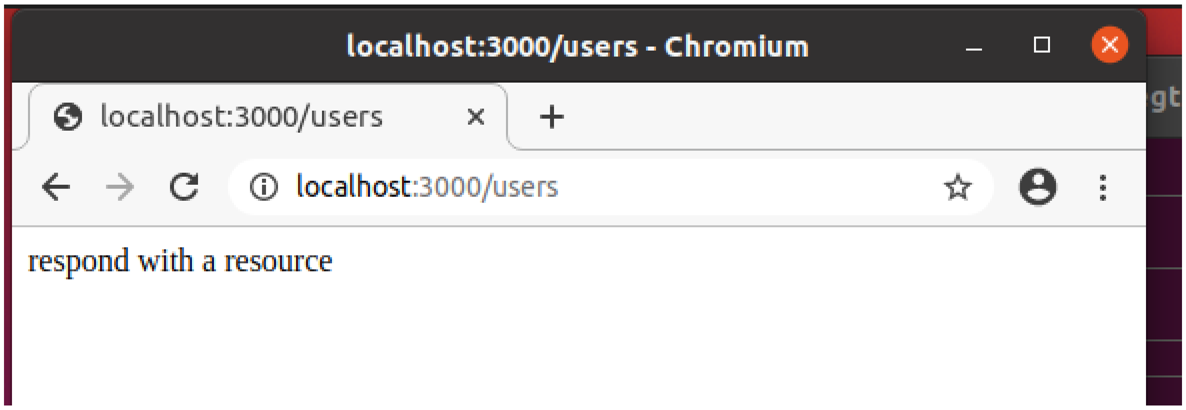
```
nullvalues@ubuntu:~/dev/egt$ node bin/www
```



Express

Welcome to Express

```
nullvalues@ubuntu:~/dev/egt$ node bin/www
GET / 200 118.571 ms - 170
GET /stylesheets/style.css 200 2.223 ms - 111
GET /favicon.ico 404 10.859 ms - 1062
```



respond with a resource

```
nullvalues@ubuntu:~/dev/egt$ node bin/www
GET /users 200 4.985 ms - 23
```

Staring to look useful: we have a web server and routes. It's 12:50. About 1:20 elapsed time.

Picked this up at 1:15 after responding to other things. Let's install MongoDB. (This is useful.)

```
nullvalues@ubuntu:~/dev/egt$ wget -qO - https://www.mongodb.org/static
/pgp/server-4.4.asc | sudo apt-key add -
nullvalues@ubuntu:~/dev/egt$ echo "deb [ arch=amd64,arm64 ]
https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/4.4 multiverse" |
sudo tee /etc/apt/sources.list.d/mongodb-org-4.4.list
nullvalues@ubuntu:~/dev/egt$ sudo apt update
...
Fetched 805 kB in 1s (890 kB
/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
All packages are up to date.
nullvalues@ubuntu:~/dev/egt$ sudo apt-get install -y mongodb-org
...
Setting up mongodb-org-server (4.4.1) ...
Adding system user `mongodb' (UID 126) ...
Adding new user `mongodb' (UID 126) with group `nogroup' ...
Not creating home directory `/home/mongodb'.
Adding group `mongodb' (GID 133) ...
Done.
Adding user `mongodb' to group `mongodb' ...
Adding user mongodb to group mongodb
Done.
Setting up mongodb-org-shell (4.4.1) ...
Setting up mongodb-database-tools (100.1.1) ...
Setting up mongodb-org-mongos (4.4.1) ...
Setting up mongodb-org-database-tools-extra (4.4.1) ...
Setting up mongodb-org-tools (4.4.1) ...
Setting up mongodb-org (4.4.1) ...
Processing triggers for man-db (2.9.1-1) ...
nullvalues@ubuntu:~/dev/egt$
```

And let’s make sure to enable it when the system starts and start it so we can test it:

```
nullvalues@ubuntu:~/dev/egt$ sudo systemctl unmask mongod
nullvalues@ubuntu:~/dev/egt$ sudo systemctl enable mongod
Created symlink /etc/systemd/system/multi-user.target.wants/mongod.
service  /lib/systemd/system/mongod.service.
nullvalues@ubuntu:~/dev/egt$ sudo systemctl start mongod
nullvalues@ubuntu:~/dev/egt$ mongo
MongoDB shell version v4.4.1
connecting to: mongodb://127.0.0.1:27017/?
compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("1ed54ecb-b73a-4ad6-9cd1-
b21709199d29") }
MongoDB server version: 4.4.1
---
The server generated these startup warnings when booting:
        2020-09-09T13:26:31.588-07:00: Using the XFS filesystem is
strongly recommended with the WiredTiger storage engine. See
http://dochub.mongodb.org/core/prodnotes-filesystem
        2020-09-09T13:26:32.044-07:00: Access control is not enabled
for the database. Read and write access to data and configuration is
unrestricted
---
> use testnode
switched to db testnode
> db.myCollection.insertOne( { x: 1 } );
> db.myCollection.find().pretty();
{ "_id" : ObjectId("5f593ae99b22f4f49d94c6ad"), "x" : 1 }
>
bye
```

Ctrl-D to quit. Looks bueno. It’s 1:30, time for coffee and a call. Elapsed time to this point is 1 hr 35 minutes.

It’s 4 PM and all is well. Let’s set up a little database connection to Mongo. I’m going to switch from Nano for editing to Webstorm because it’s easier to switch between modules and see what’s going on. Besides, a little code intelligence can be helpful. Let’s create a directory called modules at /nullvalues/dev/egt/modules and create a file in that directory called datatools.js. Then, before we get too far, let’s install the NodeJS mongodb driver globally.

```
nullvalues@ubuntu:~/dev/egt$ sudo npm install -g mongodb --save
[sudo] password for nullvalues:
+ mongodb@3.6.1
added 20 packages from 11 contributors in 1.041s
nullvalues@ubuntu:~/dev/egt$
```

Now open your datatools.js and write a singleton of your choosing to handle the Mongo connection. It might look like this:

```
let MongoClient = require('mongodb').MongoClient;

let MongoHandle = function () {
    let mCnxn = null;
    let mInstance = null;

    async function MongoConnect() {
        try {
            let dbCnxn = "mongodb://localhost:27017";
            return await MongoClient(dbCnxn, {useUnifiedTopology:
true}).connect();
        } catch (err) {
            return err;
        }
    }

    async function Get() {
        try {
            mInstance++;
            console.log("Connection to mongo has been called " +
mInstance + " times.");

            if (mCnxn != null) {
                console.log('Connection handle is already created.');
                return mCnxn;
            } else {
                console.log('Creating new connection handle.');
                mCnxn = await MongoConnect();
                return mCnxn;
            }
        } catch (err) {
            return err;
        }
    }

    return {Get: Get}
}
module.exports = MongoHandle();
```

Let’s create another module that we’ll actually interface with. We’ll call it api.js. Mine looks like this to start so we can test the Mongo connection and read just a couple of records:

```
let mongoAdapter = require('./datatools.js');

async function databaseHandle() {
    try {
        const mCnxn = await mongoAdapter.Get();
        try {
            let dbName = "testnode";
            let dbHandle = await mCnxn.db(dbName);
            return dbHandle;
        } catch (err) {
            console.log("database may not exist");
        }
    } catch (err) {
        console.log("connection failed.");
    }
}

async function simpleRead() {
    let collection = [];
    try {
        let dbHandle = await databaseHandle();
        collection  = await dbHandle.collection('myCollection').find
({}).toArray();
    } catch (err) {
        console.log(err.stack);
    } finally {
        // handle will persist, no need for this.
        //await mClient.close();
    }
    let thisSet = JSON.stringify({"exes": collection});
    console.log(thisSet);
    return JSON.parse(thisSet);
}
```

We're going to create a matching routes module for api by changing directory to ~/egt/routes and copying the auto-generated users.js to api.js. We make a few changes, including referencing our API module.

```
var express = require('express');
var documentSet = require('../modules/api.js')
var router = express.Router();

/* GET a simple listing of some predefined document type. */
router.get('/', async function(req, res, next) {
  let exes = await documentSet.wholeSet();
  if (exes == undefined) {
    res.writeHead(404, {'Content-Type': 'text/plain'});
    res.end();
  } else {
    res.json(exes);
  }
});

module.exports = router;
```

Add these lines to your app.js so we have a route for API:

```
var api = require('./routes/api');

app.use('/api', api);
```

In the real world we probably won't use "API" for this name, but the idea is right. Who knows, maybe we will?

So we run an npm bin/www and see what happens.

```
nullvalues@ubuntu:~/dev/egt$ npm bin/www
```

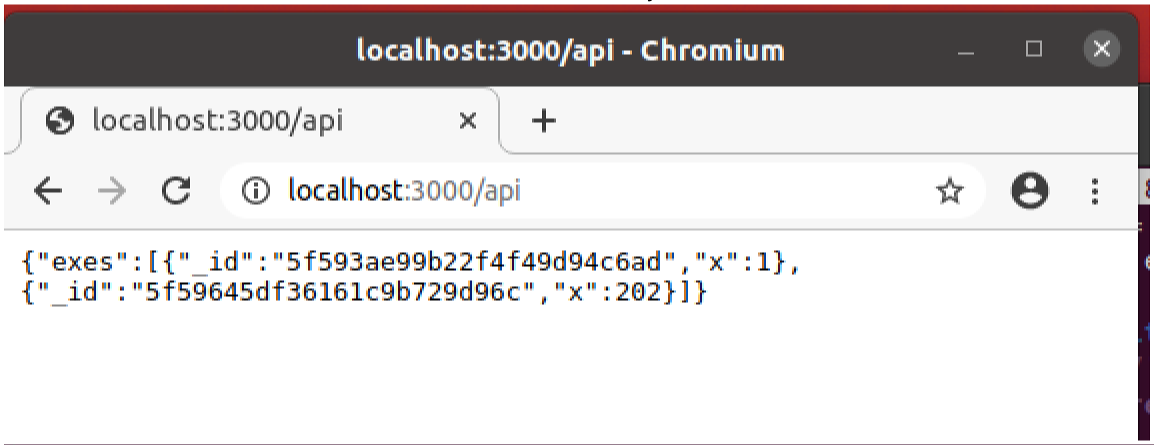
It failed. Just on a hunch I installed npm locally with a:

```
nullvalues@ubuntu:~/dev/egt$ npm install mongodb
+ mongodb@3.6.1
added 16 packages from 10 contributors, removed 38 packages and audited
117 packages in 1.818s
found 4 vulnerabilities (3 low, 1 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
nullvalues@ubuntu:~/dev/egt$
```

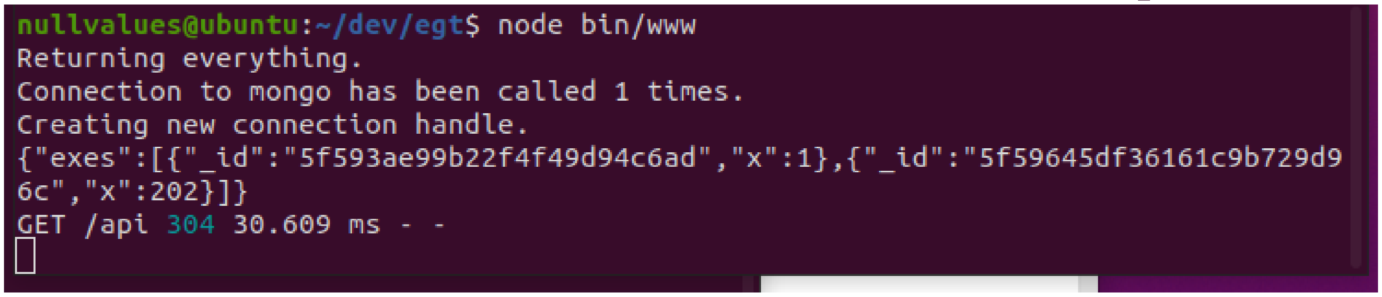
It seems like our global install of the NodeJS Mongo driver might have gone wonky. Try again with the local install:

```
nullvalues@ubuntu:~/dev/egt$ node bin/www
```

We'll need to do some work on that later. But it seems to run. Let's try it:

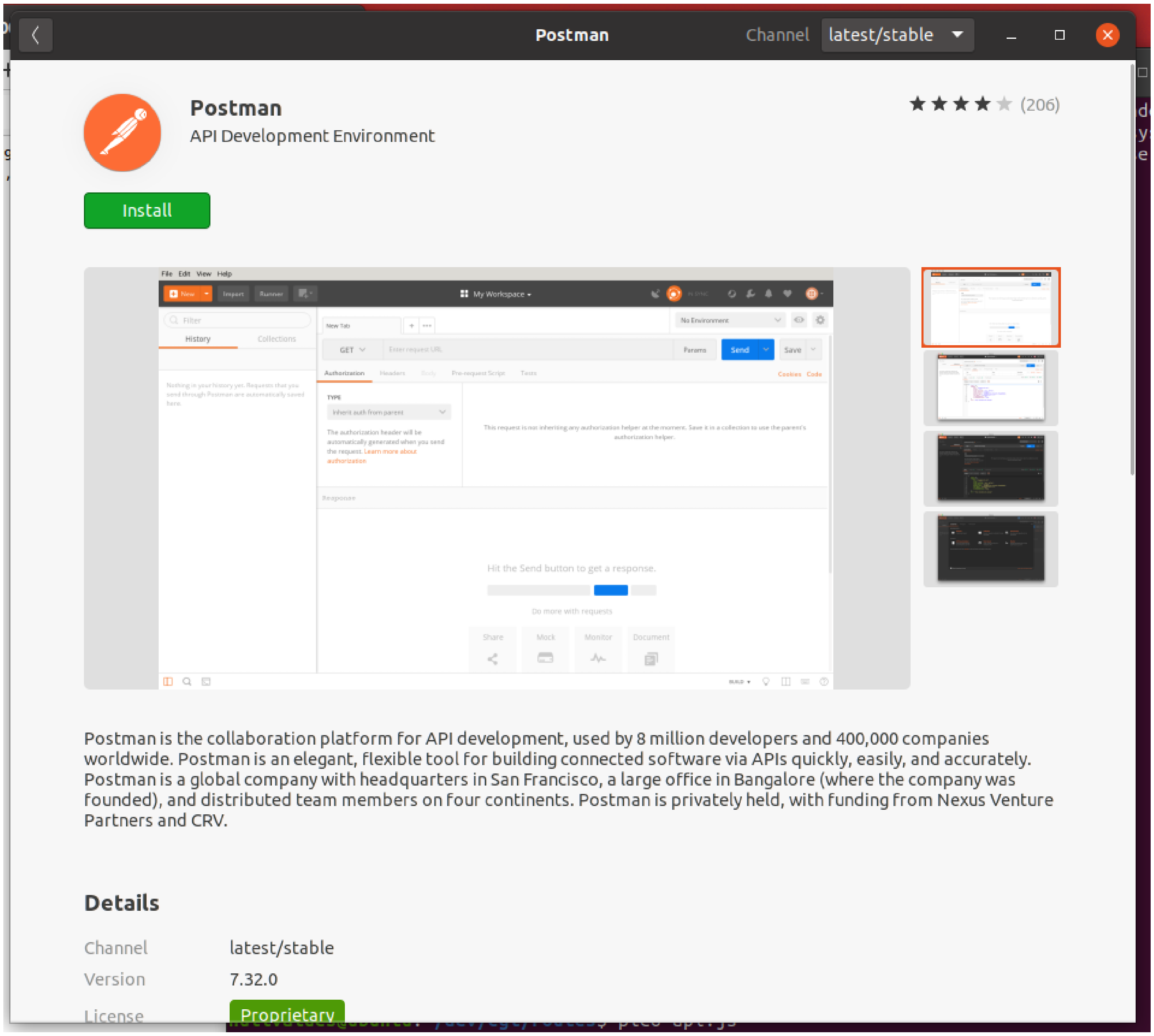


```
{"exes":[{"_id":"5f593ae99b22f4f49d94c6ad","x":1},
{"_id":"5f59645df36161c9b729d96c","x":202}]}
```



```
nullvalues@ubuntu:~/dev/egt$ node bin/www
Returning everything.
Connection to mongo has been called 1 times.
Creating new connection handle.
{"exes":[{"_id":"5f593ae99b22f4f49d94c6ad","x":1},{"_id":"5f59645df36161c9b729d9
6c","x":202}]}
GET /api 304 30.609 ms - -
```

Seems right! That's encouraging. After this point we'll need better testing tools, so we may as well install Postman through Snap:



Postman

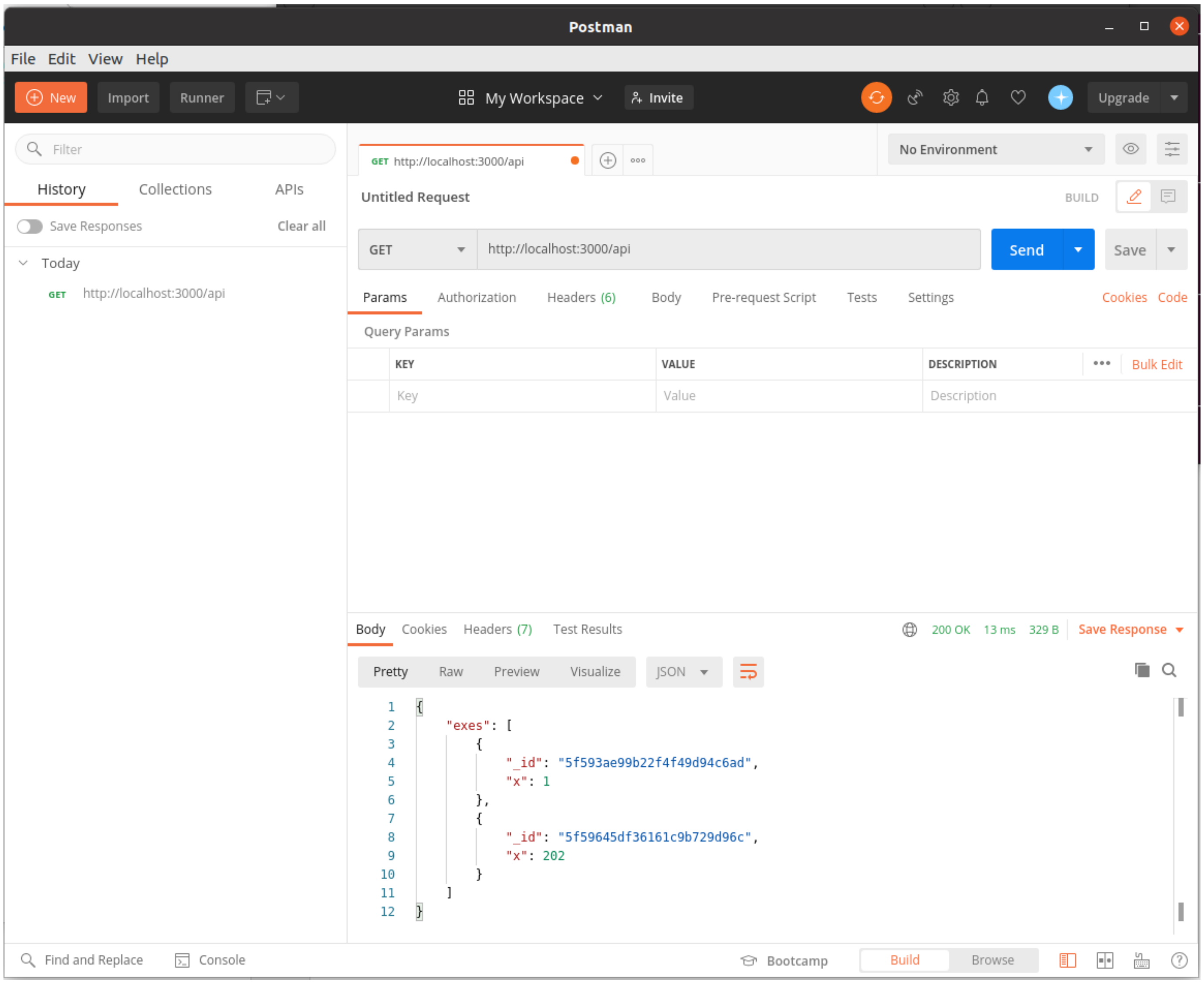
API Development Environment

Postman is the collaboration platform for API development, used by 8 million developers and 400,000 companies worldwide. Postman is an elegant, flexible tool for building connected software via APIs quickly, easily, and accurately. Postman is a global company with headquarters in San Francisco, a large office in Bangalore (where the company was founded), and distributed team members on four continents. Postman is privately held, with funding from Nexus Venture Partners and CRV.

Details

Channel	latest/stable
Version	7.32.0
License	Proprietary

We run the same request against our test app and see the same result in a nicer format:

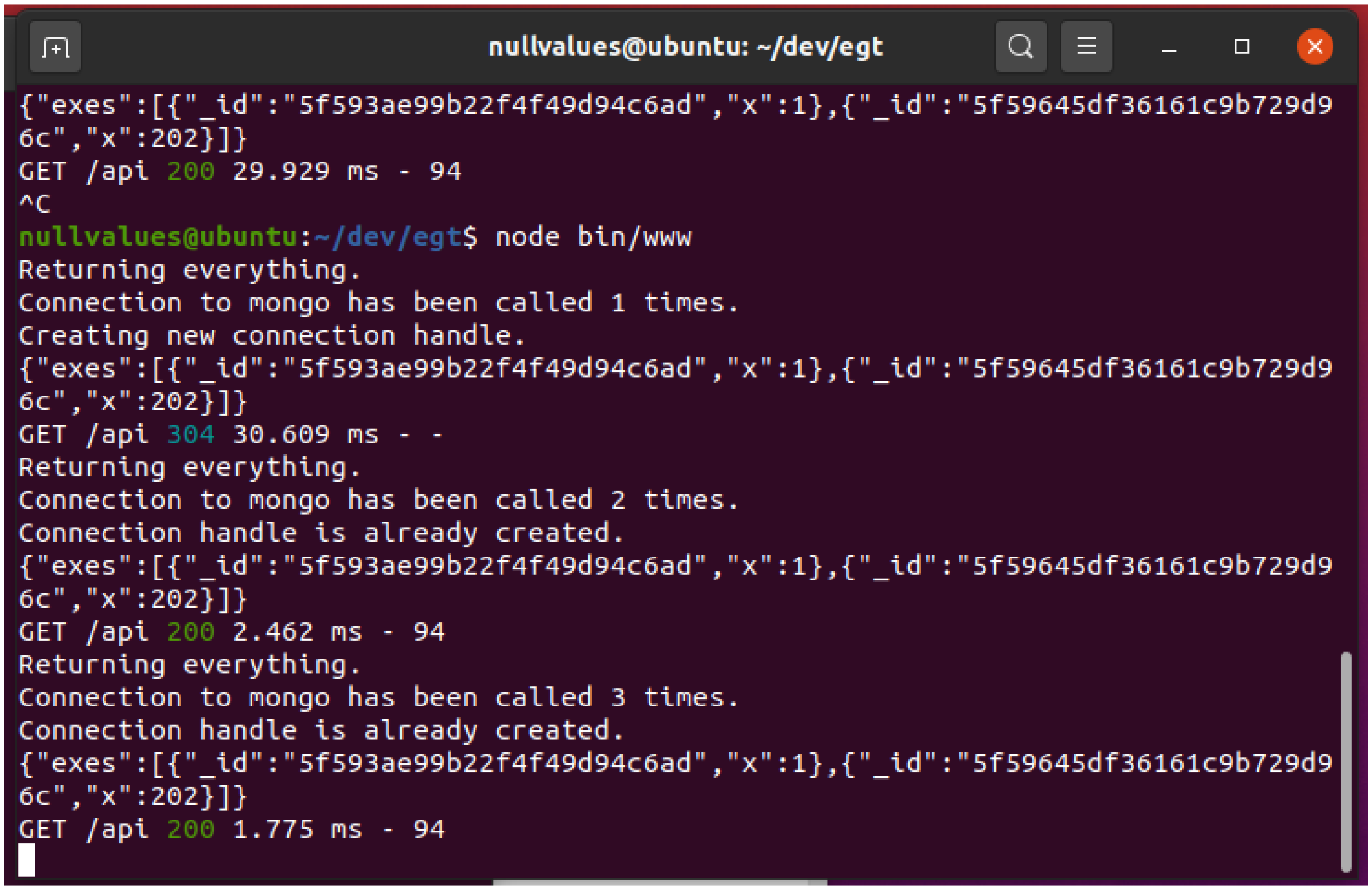

Postman
File Edit View Help
New Import Runner
My Workspace
Invite
Upgrade
Filter
History Collections APIs
Save Responses Clear all
Today
GET http://localhost:3000/api
No Environment
Untitled Request
BUILD
Send
Save
Params Authorization Headers (6) Body Pre-request Script Tests Settings
Cookies Code
Query Params
Bulk Edit
Body Cookies Headers (7) Test Results
Save Response
Pretty Raw Preview Visualize JSON
Find and Replace Console Bootcamp Build Browse

GET http://localhost:3000/api

	KEY	VALUE	DESCRIPTION
	Key	Value	Description

200 OK 13 ms 329 B

```
{
    "exes": [
        {
            "_id": "5f593ae99b22f4f49d94c6ad",
            "x": 1
        },
        {
            "_id": "5f59645df36161c9b729d96c",
            "x": 202
        }
    ]
}
```



```
{"exes":[{"_id":"5f593ae99b22f4f49d94c6ad","x":1},{"_id":"5f59645df36161c9b729d9
6c","x":202}]}
GET /api 200 29.929 ms - 94
^C
nullvalues@ubuntu:~/dev/egt$ node bin/www
Returning everything.
Connection to mongo has been called 1 times.
Creating new connection handle.
{"exes":[{"_id":"5f593ae99b22f4f49d94c6ad","x":1},{"_id":"5f59645df36161c9b729d9
6c","x":202}]}
GET /api 304 30.609 ms - -
Returning everything.
Connection to mongo has been called 2 times.
Connection handle is already created.
{"exes":[{"_id":"5f593ae99b22f4f49d94c6ad","x":1},{"_id":"5f59645df36161c9b729d9
6c","x":202}]}
GET /api 200 2.462 ms - 94
Returning everything.
Connection to mongo has been called 3 times.
Connection handle is already created.
{"exes":[{"_id":"5f593ae99b22f4f49d94c6ad","x":1},{"_id":"5f59645df36161c9b729d9
6c","x":202}]}
GET /api 200 1.775 ms - 94
```

And our server doesn't really see any difference. That's cool. It's starting to look a little like a real service we might deploy. The Singleton even seem to be doing its job in reducing connection overhead: transaction times are trending down. That seems promising, especially for a high-volume API.

It's about 5:00. Not quite another hour elapsed, so let's call it 2 hours total. Here's the caveat emptor: I taught myself this once before and am doing it a second time after retraining myself previously on how to do things like write a Singleton in JS. I'm definitely not that good. Or fast. Or competent. So the elapsed time is just for you to gauge how things should go if you do what I do and are largely copying what I have here to get your environment working, rather than creating new.

Let's keep trucking. Like the good kids we are, we're going to start by writing tests first. We're going to cheat this very first time for the sake of clarity in this example, but we're going to write a simple unit test now as an example we can expand later. In our ~/dev directory, create a test directory if you don't already have it. Then create a test-api.js file in that directory.

This is what mine looks like. You'll notice that I cheated and copied the JSON output from Postman as the literal to test against.

```
let api = require('../modules/api');

exports.wholeSetTest = async function (test) {
    test.equal(await api.wholeSet(),
        '{"exes":[{"_id":"5f593ae99b22f4f49d94c6ad","x":1},
                  {"_id":"5f59645df36161c9b729d96c","x":202}]}'
    );
    test.done();
}
```

Let's run it through NodeUnit that we installed earlier.

```
nullvalues@ubuntu:~/dev/egt$ nodeunit test/test-api.js

test-api.js
Returning everything.
Connection to mongo has been called 1 times.
Creating new connection handle.
{"exes":[{"_id":"5f593ae99b22f4f49d94c6ad","x":1},{"_id":"
5f59645df36161c9b729d96c","x":202}]}
 wholeSetTest

AssertionError:
{
  exes: [
    { _id: '5f593ae99b22f4f49d94c6ad', x: 1 },
    { _id: '5f59645df36161c9b729d96c', x: 202 }
  ]
}
==
'{"exes":[{"_id":"5f593ae99b22f4f49d94c6ad","x":1},{"_id":"
5f59645df36161c9b729d96c","x":202}]}'
    at Object.equal (/usr/lib/node_modules/nodeunit/lib/types.js:83:39)
    at Object.exports.wholeSetTest (/home/nullvalues/dev/egt/test/test-
api.js:4:10)
    at processTicksAndRejections (internal/process/task_queues.js:93:5)


FAILURES: 1/1 assertions failed (115ms)
nullvalues@ubuntu:~/dev/egt$
```

Dang. I'm into new territory, so I wasn't sure if that was going to work or not. There must be a literal comparison problem, but at least we got a result from our test. What do you bet it has to match quote for quote? I'm also hitting the API directly, not actually testing the route. So it looks like what we're getting is a JSON object rather than a string literal that we'd get from the API via the route (with header, etc). That's actually a valid test, but likely only one of two we'd want.

Since I copied the literal from Postman, I bet I got it wrong. Let's modify the comparison object to look more like JSON and less like a string literal:

```
    test.equal(await api.wholeSet(),
    "{exes:[
    {'_id':5f593ae99b22f4f49d94c6ad,'x':1},
    {'_id':5f59645df36161c9b729d96c,'x':202}]}"
    );
```

That failed still. This seems futile, doesn't it, to worry about space for space matching? So let's try to evaluate a little differently.

```
let api = require('../modules/api');

exports.wholeSetTest = async function (test) {
    let result = await api.wholeSet();
    let expectation = {exes: [{ _id: '5f593ae99b22f4f49d94c6ad', 'x':
1},
                              { _id: '5f59645df36161c9b729d96c', 'x':
202}]}
    test.deepEqual(result, expectation);
    test.done();
}
```

```
nullvalues@ubuntu:~/dev/egt$ nodeunit test/test-api.js

test-api.js
Returning everything.
Connection to mongo has been called 1 times.
Creating new connection handle.
{"exes":[{"_id":"5f593ae99b22f4f49d94c6ad","x":1},{"_id":"
5f59645df36161c9b729d96c","x":202}]}
 wholeSetTest

OK: 1 assertions (112ms)
nullvalues@ubuntu:~/dev/egt$
```

It seems that the deepEqual() method that NodeJS offers is a little more thoughtful about how it evaluates for equivalence. That's something to dig into more later, but at least we have a working test (though fragile one that only works for this data set at the moment).

You might recognize right there that we're using promises on the test, and throughout the API. If you pull that async and await here, this test will fail. The object returns before the database object. You can try it if you want. I forgot it the first time and the test failed. So writing the test actually reinforced the underlying nature of the code. This Agile thing is growing on me.

It's 5:40 and we have a working M-E-N stack with a test. So in a little less than three hours we:

- set up an Ubuntu VM as our server and client
- loaded our development and test tools to the VM
- installed our MEN portion of the stack on the VM
  - Mongo
  - Express
  - NodeJS
- wrote a sample application with a test that gets data from a database to a web interface
- learned something new about writing tests
- wrote this documentation as we went

More to come later when we expand the API to do CRUD operations, and we'll figure out how to add in middleware into the routes for authentication. But this is technically a working (though pretty useless) REST API, as promised.

Adding Authentication to the API

We're going to start with simple, local JWT authentication, and then we'll extend JWT use later to an SSO service. Retrieval of the JWT probably isn't a major issue, so let's learn to protect our routes by using some Express JWT boilerplate and then work outward from there.

In our modules folder, I have added a file called, appropriately jwt.js.

```
const expressJwt = require('express-jwt');
module.exports = jwt;
function jwt() {
    // make this easy for now, probably has a better home elsewhere
    const { secret } = {"secret": "1F209B6E-zqKTZ19 && 1F209B6E-
11UrAee2h && 1F209B6E-1Rp3Vns44A"};
    // include a slice of public routes that don't require auth,
otherwise authenticate
    return expressJwt({ secret, algorithms: ['HS256'] }).unless({
        path: [
            '/auth',
            '/about'
        ]
    });
}
```

This should make it possible to protect all routes (served by this application) by default, unless specified here as being excluded.

Add a config.json file to your root folder, and include a secret in it that we can use to encrypt the JWT tokens. Mine looks like this:

```
{
  "secret":"1F209B6E-zqKTZ19 && 1F209B6E-11UrAee2h && 1F209B6E-
1Rp3Vns44A"
}
```

It's not really important what that secret is, so just make it yours and keep it a secret.

In our app.js in our root project folder, let's add directives to use our jwt module and protect our routes.

```
const jwt = require('./modules/jwt');

var authRouter = require('./routes/auth');

// secure the api
app.use(jwt());

// api routes
app.use('/auth', authRouter);
```

Mine looks like this now:

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');
const jwt = require('./modules/jwt');

var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
var api = require('./routes/api');
var authRouter = require('./routes/auth')

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

// use JWT auth to secure the api
app.use(jwt());
// api routes
app.use('/auth', authRouter);
app.use('/', indexRouter);
app.use('/users', usersRouter);
app.use('/api', api);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page
  res.status(err.status || 500);
  res.render('error');
});

module.exports = app;
```

Because we started with boilerplate from a book example, there's some junk in there we'll clean out later. But it won't really get in the way at the moment.

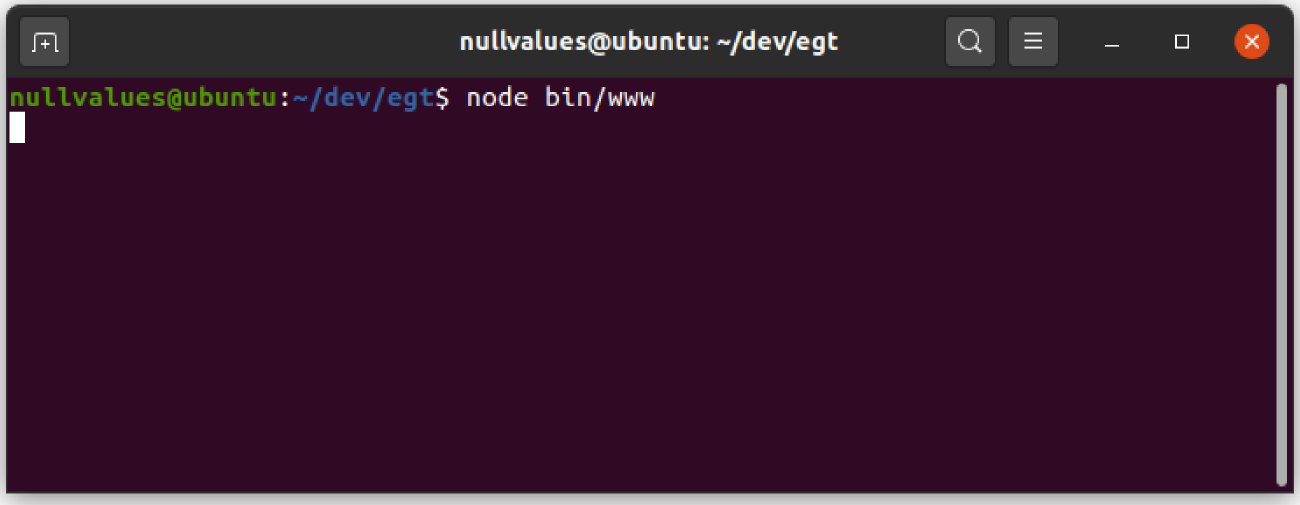
At this point, we're missing the guts of the authentication. We need a route and a matching module. So let's create a routes/auth.js file and add a simple auth function.

```
const express = require('express');
const router = express.Router();
const authService = require('../modules/auth');

router.post('/auth', auth);

module.exports = router;
function auth(req, res, next) {
    authService.auth(req.body)
        .then(user => res.json(user))
        .catch(next);
}
```

And now let's create a modules/auth.js file (as shown in our require statement above).

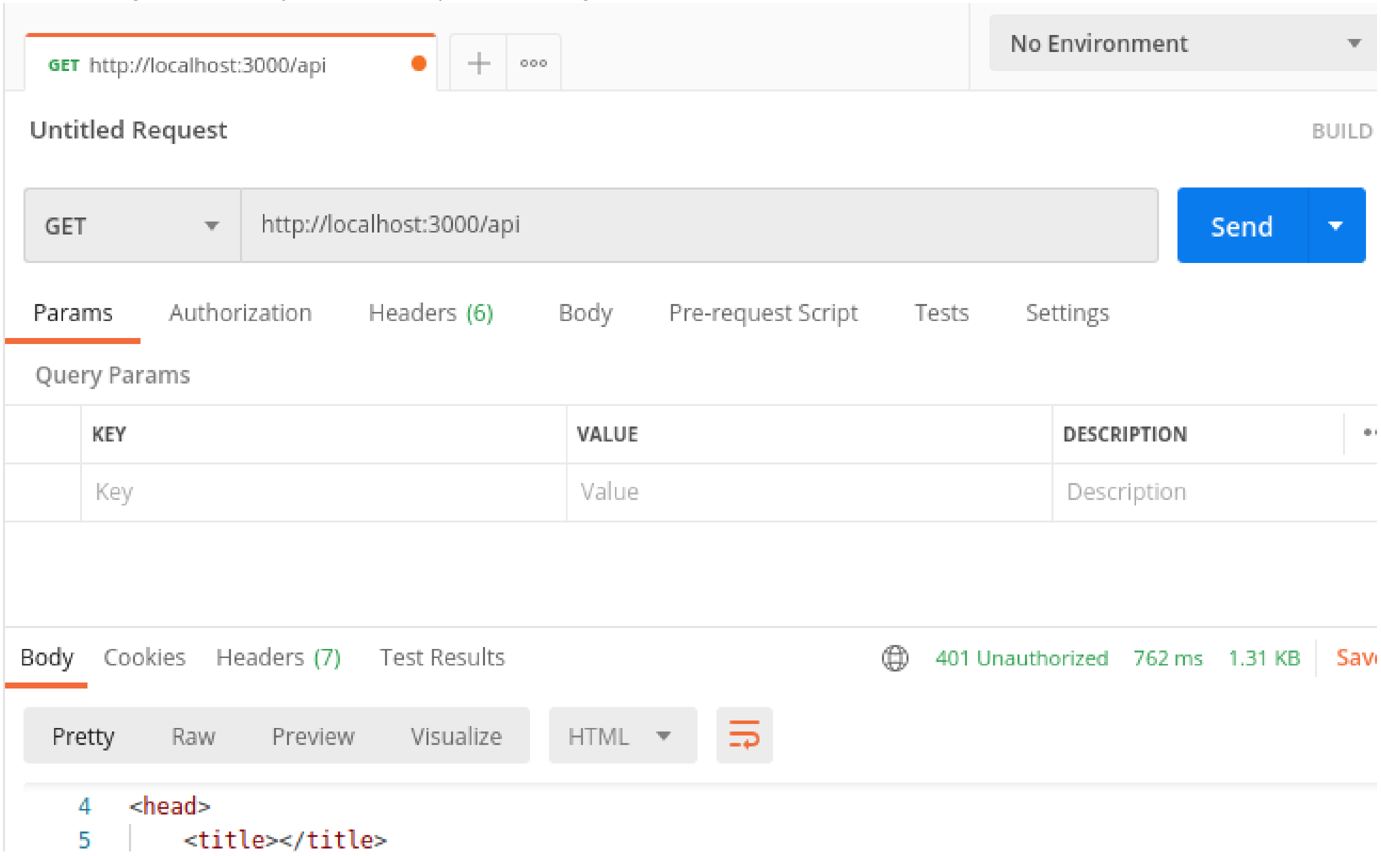
```
// require our secret
const config = require('../config.json');
// and make sure we can use tokesn
const jwt = require('jsonwebtoken');

// we'll move this to mongo in just a minute or two
const users = [{
  id: 10014,
  username: 'Davis',
  userpass: 'ShawtyPants',
  firstName: 'David',
  lastName: 'Jacunsen'
}];

module.exports = auth;

async function auth({ username, userpass }) {
    const user = users.find(u => u.username === username && u.userpass
=== password);

    if (!user) throw 'Authentication Error';

    // 1 day jwt token
    const token = jwt.sign({ sub: user.id }, config.secret, {
expiresIn: '1d' });
    return {
        ...omitPassword(user),
        token
    };
}

function omitPassword(user) {
    const { userpass, ...userWithoutPassword } = user;
    return userWithoutPassword;
}
```

The first time I started it, I got some error that I fixed by correcting one or two typos in paths. It then found all the references and the application started normally:



```
nullvalues@ubuntu:~/dev/egt$ node bin/www
```

We do a little get action on any route that isn't public, and we get:



GET http://localhost:3000/api

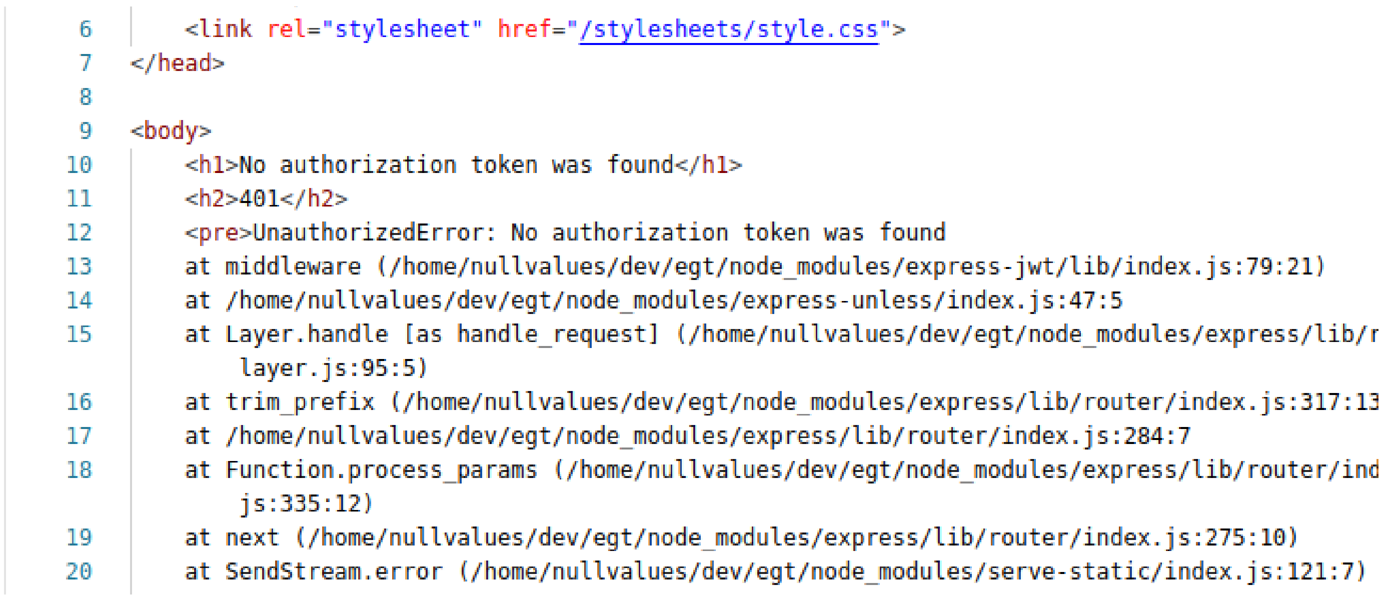
Untitled Request

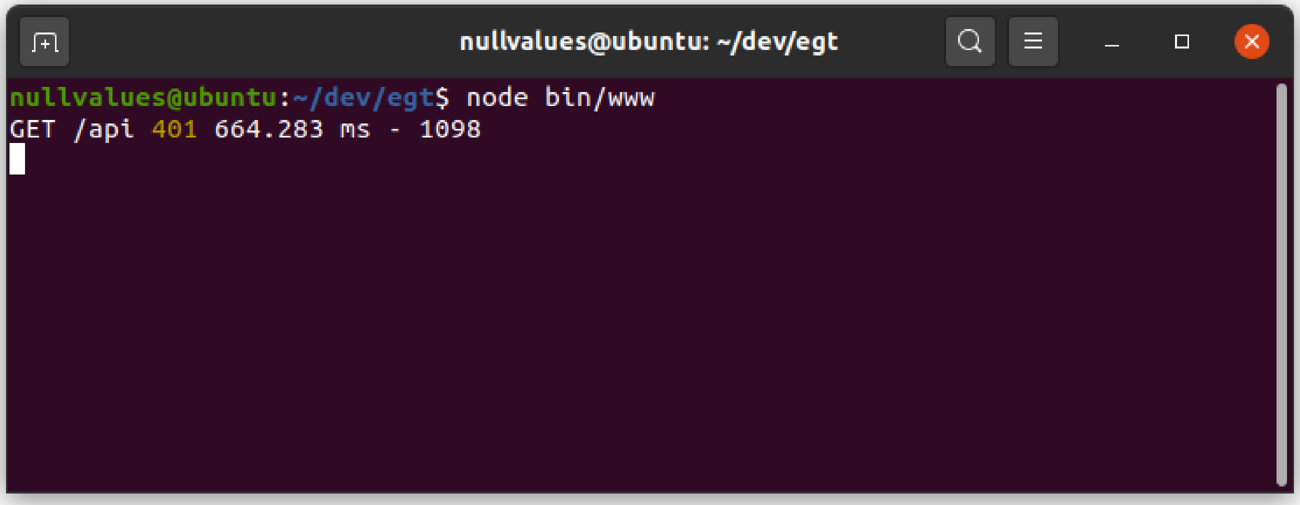
GET http://localhost:3000/api

KEY	VALUE	DESCRIPTION
Key	Value	Description

401 Unauthorized 762 ms 1.31 KB

```
<head>
    <title></title>
```

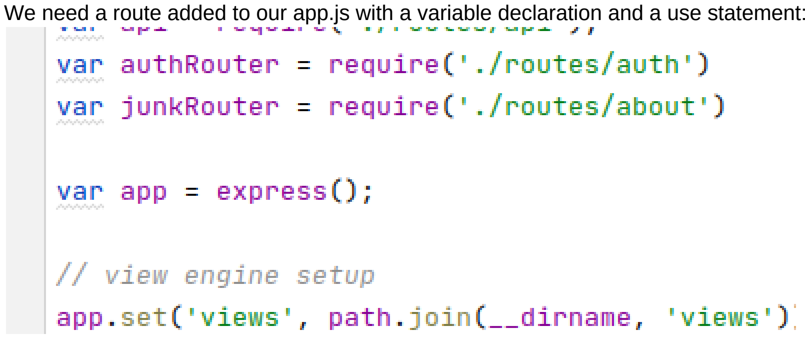

```
    <link rel="stylesheet" href="/stylesheets/style.css">
</head>

<body>
    <h1>No authorization token was found</h1>
    <h2>401</h2>
    <pre>UnauthorizedError: No authorization token was found
    at middleware (/home/nullvalues/dev/egt/node_modules/express-jwt/lib/index.js:79:21)
    at /home/nullvalues/dev/egt/node_modules/express-unless/index.js:47:5
    at Layer.handle [as handle_request] (/home/nullvalues/dev/egt/node_modules/express/lib/r
        layer.js:95:5)
    at trim_prefix (/home/nullvalues/dev/egt/node_modules/express/lib/router/index.js:317:13
    at /home/nullvalues/dev/egt/node_modules/express/lib/router/index.js:284:7
    at Function.process_params (/home/nullvalues/dev/egt/node_modules/express/lib/router/ind
        js:335:12)
    at next (/home/nullvalues/dev/egt/node_modules/express/lib/router/index.js:275:10)
    at SendStream.error (/home/nullvalues/dev/egt/node_modules/serve-static/index.js:121:7)
```

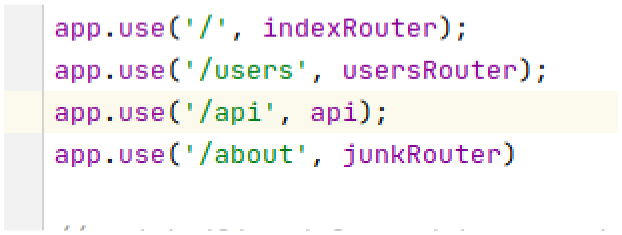


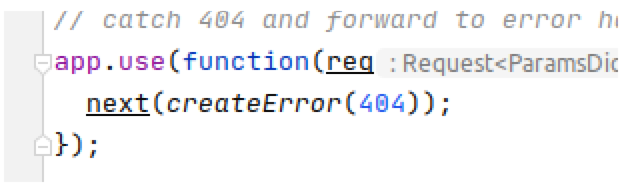
```
nullvalues@ubuntu:~/dev/egt$ node bin/www
GET /api 401 664.283 ms - 1098
```

Cool, our API paths are protected now. Let's do a little more work to make sure public paths are accessible before we move on to checking our token usage.

We need a route added to our app.js with a variable declaration and a use statement:


```
var authRouter = require('./routes/auth')
var junkRouter = require('./routes/about')

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views')
```


```
app.use('/', indexRouter);
app.use('/users', usersRouter);
app.use('/api', api);
app.use('/about', junkRouter)
```

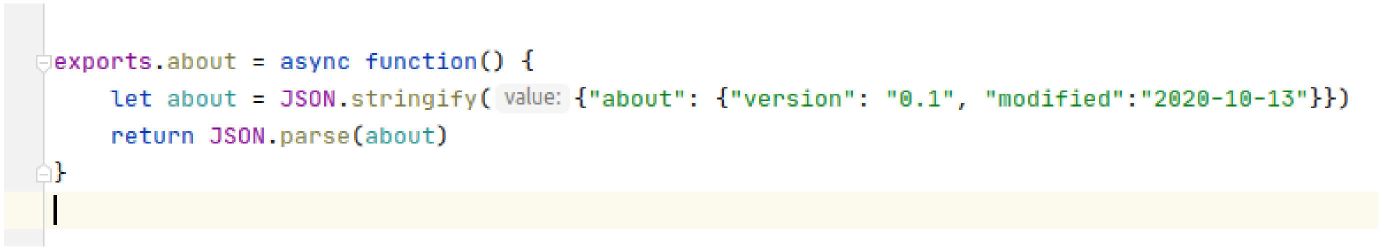


```
// catch 404 and forward to error h
app.use(function(req : Request<ParamsDic
  next(createError(404));
});
```

Now copy routes/api.js to routes/about.js for an easy starting point to a “junk drawer” set of routes. Might call this “other” or “misc” or “helper”, but in any case, it’s going to use an API function via an unprotected route to get the “about” information for this application. Mine looks like this when it’s been edited:

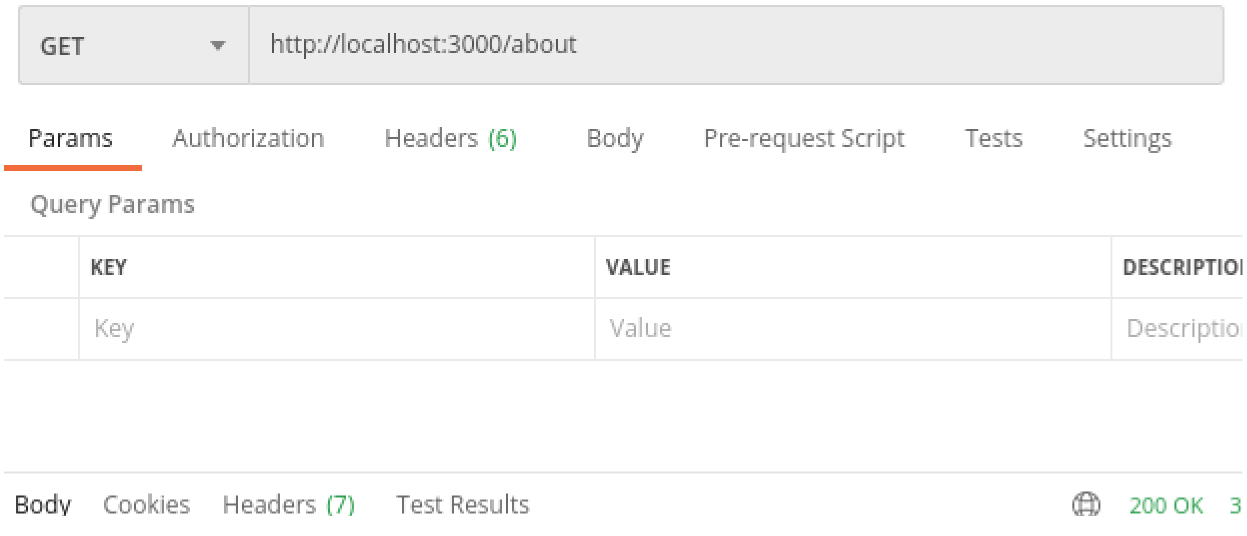
```
var express = require('express');
var api = require('../modules/api.js')
var router = express.Router();

/* GET a simple listing of some predefined document type. */
router.get('/', async function(req, res, next) {
  let about = await api.about();
  if (about == undefined) {
    res.writeHead(404, {'Content-Type': 'text/plain'});
    res.end();
  } else {
    res.json(about);
  }
});

module.exports = router;
```

Add a function to our API:



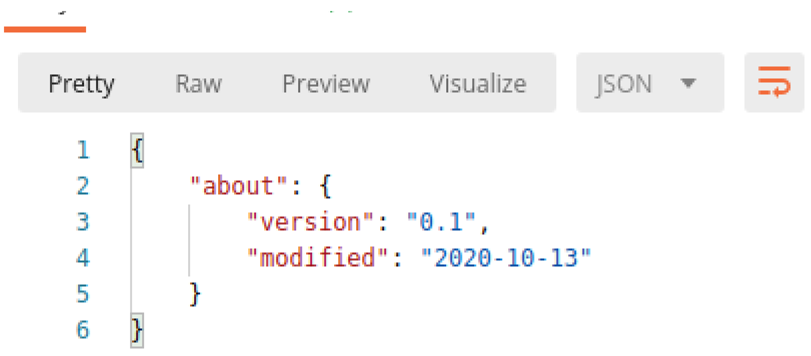
```
exports.about = async function() {
    let about = JSON.stringify( value: {"about": {"version": "0.1", "modified":"2020-10-13"}})
    return JSON.parse(about)
}
```

And now we should be able to run the application again and check this route with postman:



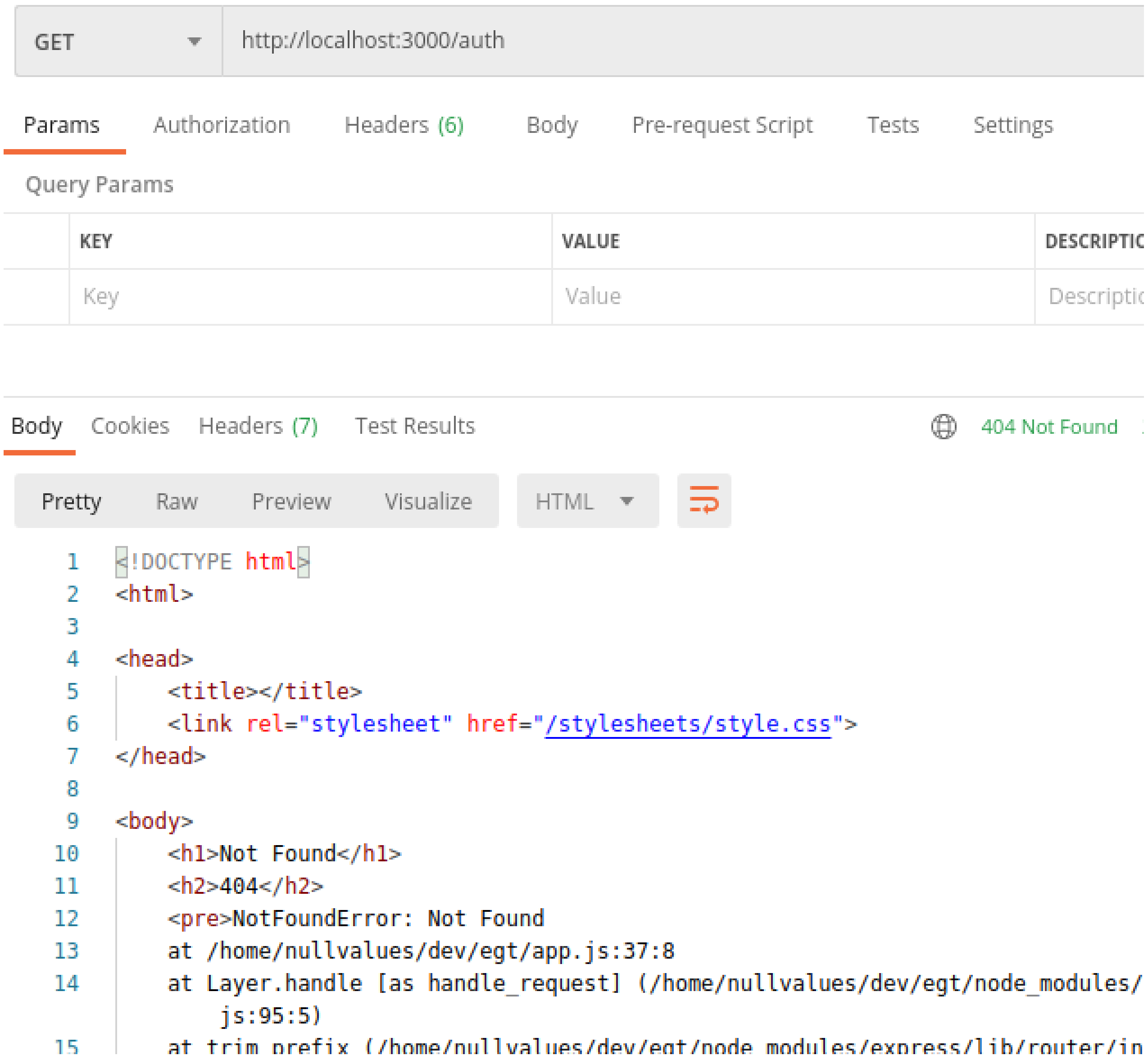
GET http://localhost:3000/about

KEY	VALUE	DESCRIPTIO
Key	Value	Descriptio



```json
{
    "about": {
        "version": "0.1",
        "modified": "2020-10-13"
    }
}
```

That looks like it's working. Simple, but proves the point. Now let's try our auth feature:



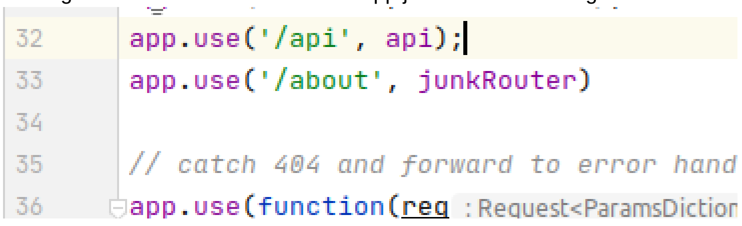
```
GET http://localhost:3000/auth
```

KEY	VALUE	DESCRIPTIC
Key	Value	Descriptic

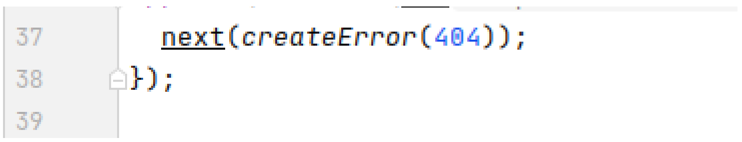
404 Not Found

```html
<!DOCTYPE html>
<html>

<head>
    <title></title>
    <link rel="stylesheet" href="/stylesheets/style.css">
</head>

<body>
    <h1>Not Found</h1>
    <h2>404</h2>
    <pre>NotFoundError: Not Found
    at /home/nullvalues/dev/egt/app.js:37:8
    at Layer.handle [as handle_request] (/home/nullvalues/dev/egt/node_modules/
        js:95:5)
    at trim_prefix (/home/nullvalues/dev/egt/node_modules/express/lib/router/in
```

That's no good. Let's check line 37 of our app.js and see who caught the error.



```
32  app.use('/api', api);
33  app.use('/about', junkRouter)
34
35  // catch 404 and forward to error hand
36  app.use(function(req :Request<ParamsDiction
```

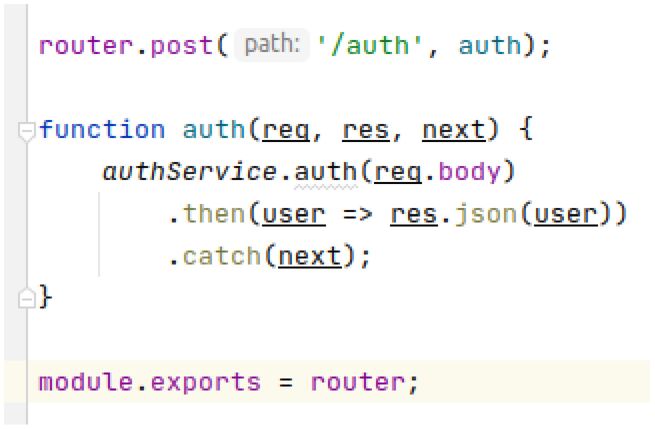

```
    next(createError(404));
});
```

I don't know much about this, but it looks like our route didn't get caught prior to falling through to the default error. This is probably a good time to do some cleanup and simplify to see if anything occurs to us as being out of place.

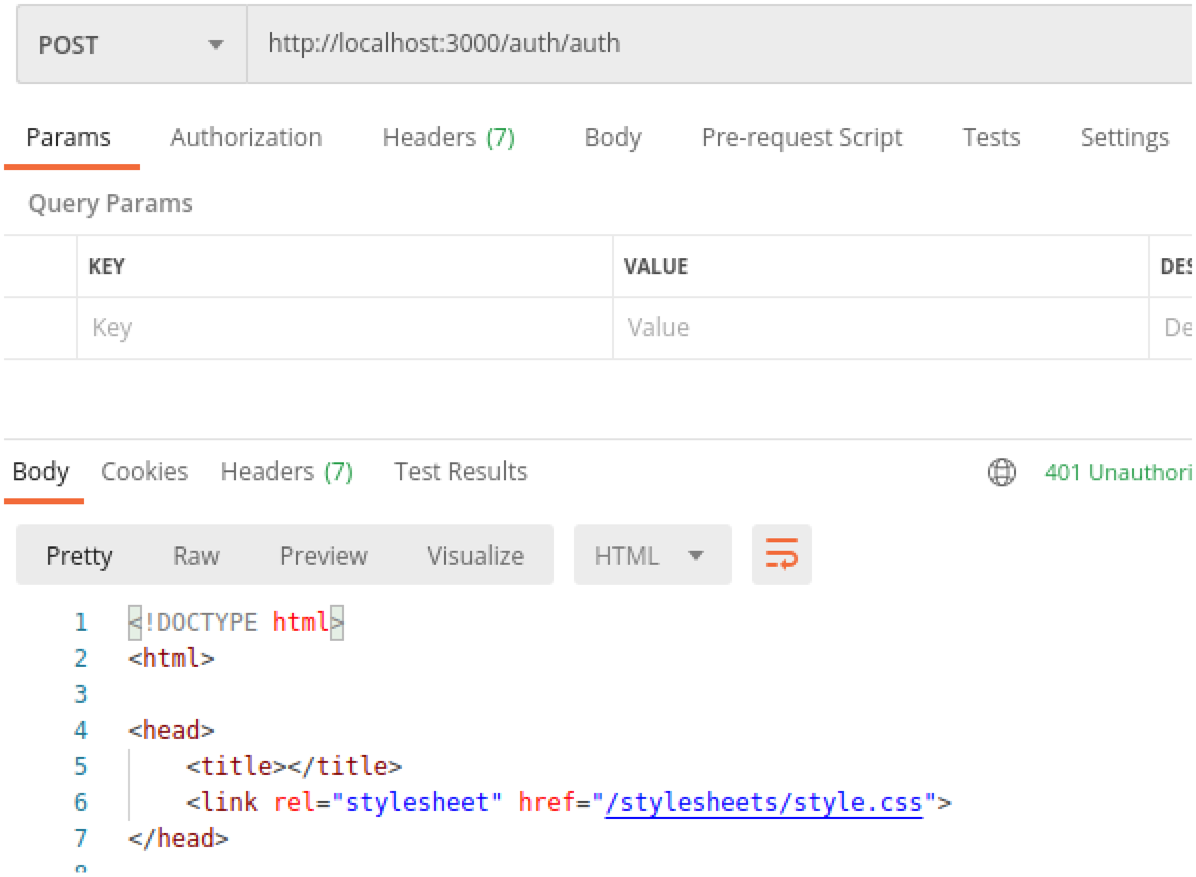
Let's remove unused routes from app.js.

- The User route is unnecessary, it was just boilerplate from the example we started working from.
- API features will be accessed through named routes, but not directly at /api, so we can remove it. It was an easy thing to set up to test, but now isn't necessary. Our "about" route shows that API functions are protected and usable. For example, if you add a trailing slash on the about request url, it will be blocked as unauthenticated.

None of that was in the way, but as I was working, it occurred to me that the paths expressed in the routes files are *relative*, and I think we have a mistake there. We have:


```
router.post( path: '/auth', auth);

function auth(req, res, next) {
    authService.auth(req.body)
        .then(user => res.json(user))
        .catch(next);
}

module.exports = router;
```

This means that we don't have a route for http://localhost:3000/auth, we have a route for http://localhost:3000/auth/auth. No wonder it wasn't caught before falling through. I wonder if that redundant route works.


POST http://localhost:3000/auth/auth

Params | Authorization | Headers (7) | Body | Pre-request Script | Tests | Settings

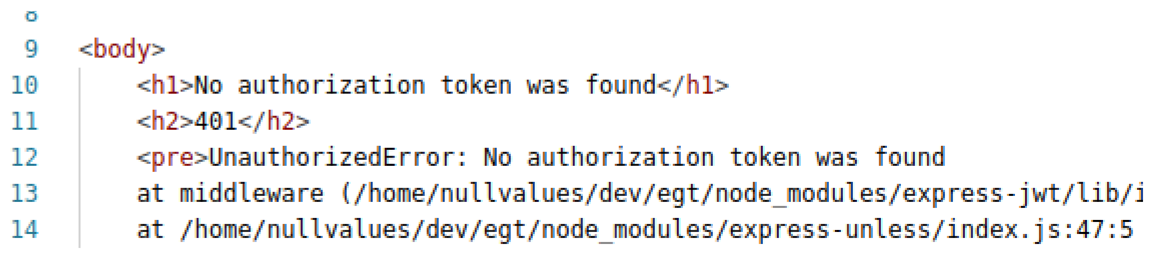
Query Params

KEY	VALUE	DES
Key	Value	De

Body | Cookies | Headers (7) | Test Results — 401 Unauthori

Pretty | Raw | Preview | Visualize | HTML

```
<!DOCTYPE html>
<html>

<head>
    <title></title>
    <link rel="stylesheet" href="/stylesheets/style.css">
</head>
```



```
<body>
    <h1>No authorization token was found</h1>
    <h2>401</h2>
    <pre>UnauthorizedError: No authorization token was found
    at middleware (/home/nullvalues/dev/egt/node_modules/express-jwt/lib/i
    at /home/nullvalues/dev/egt/node_modules/express-unless/index.js:47:5
```

That guess was correct, because now we’re on to a 401 instead of 404. That’s probably progress. Let’s fix the route first, so /auth is not redundant. We might need more auth functions later, so let’s refactor a bit for clarity. I changed my auth route file to look like this:

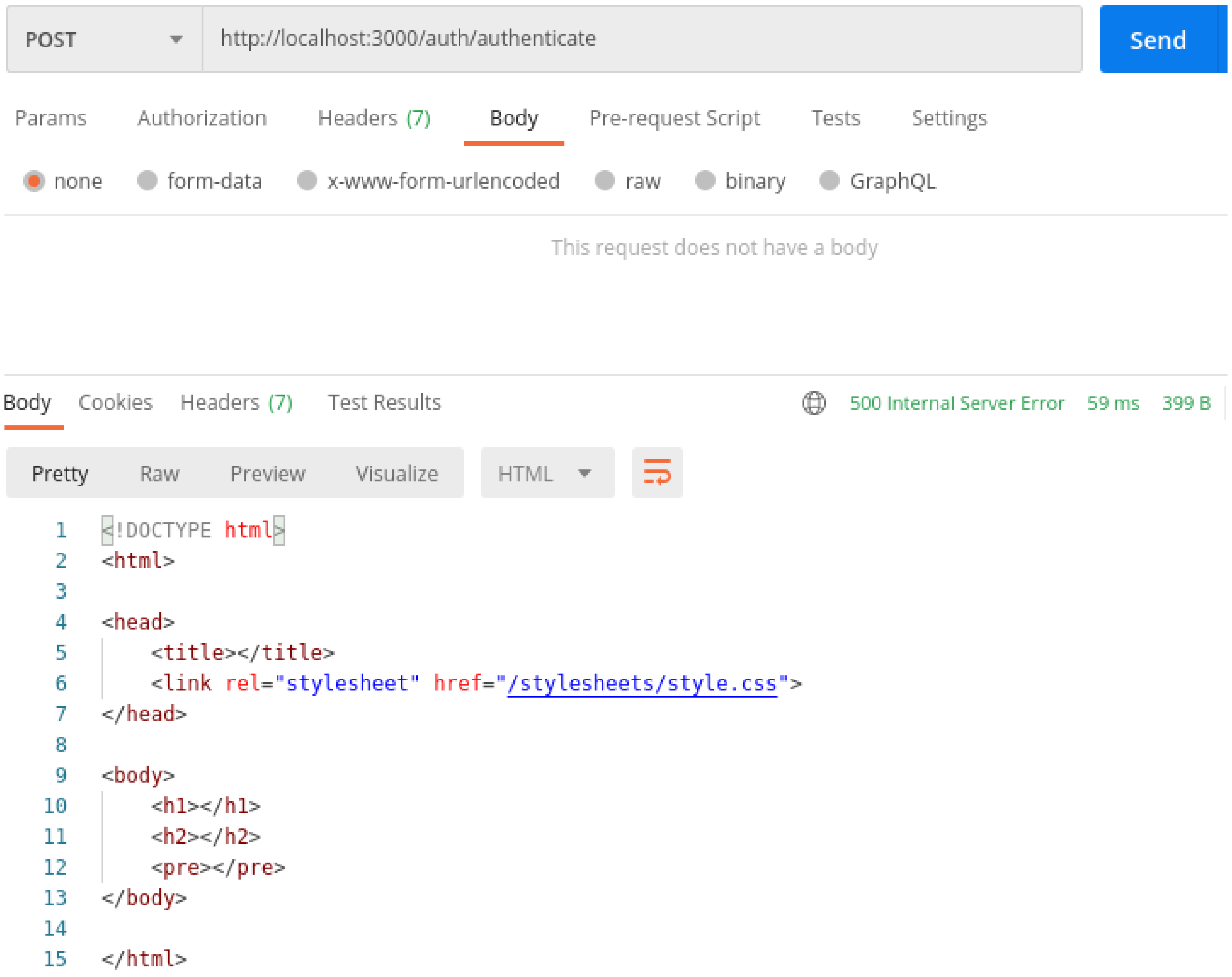
```
const express = require('express');
const router = express.Router();
const authService = require('../modules/auth');

router.post('/authenticate', authenticate);

function authenticate(req, res, next) {
    authService.authenticate(req.body)
        .then(user => res.json(user))
        .catch(next);
}

module.exports = router;
```

And I changed my auth module to match:

```
// require our secret
const config = require('../config.json');
// and make sure we can use tokens
const jwt = require('jsonwebtoken');

// we'll move this to mongo in just a minute or two
const users = [{
    id: 10014,
    username: 'Davis',
    userpass: 'ShawtyPants',
    firstName: 'Davis',
    lastName: 'Jacunsen'
}];

exports.authenticate = async function ({ username, userpass }) {
    const user = users.find(u => u.username === username && u.userpass
=== password);

    if (!user) throw 'Authentication Error';

    // 1 day jwt token
    const token = jwt.sign({ sub: user.id }, config.secret, {
```

```
expiresIn: '1d' });
    return {
        ...omitPassword(user),
        token
    };
}

function omitPassword(user) {
    const { userpass, ...userWithoutPassword } = user;
    return userWithoutPassword;
}
```

Looks a little better:


POST http://localhost:3000/auth/authenticate Send

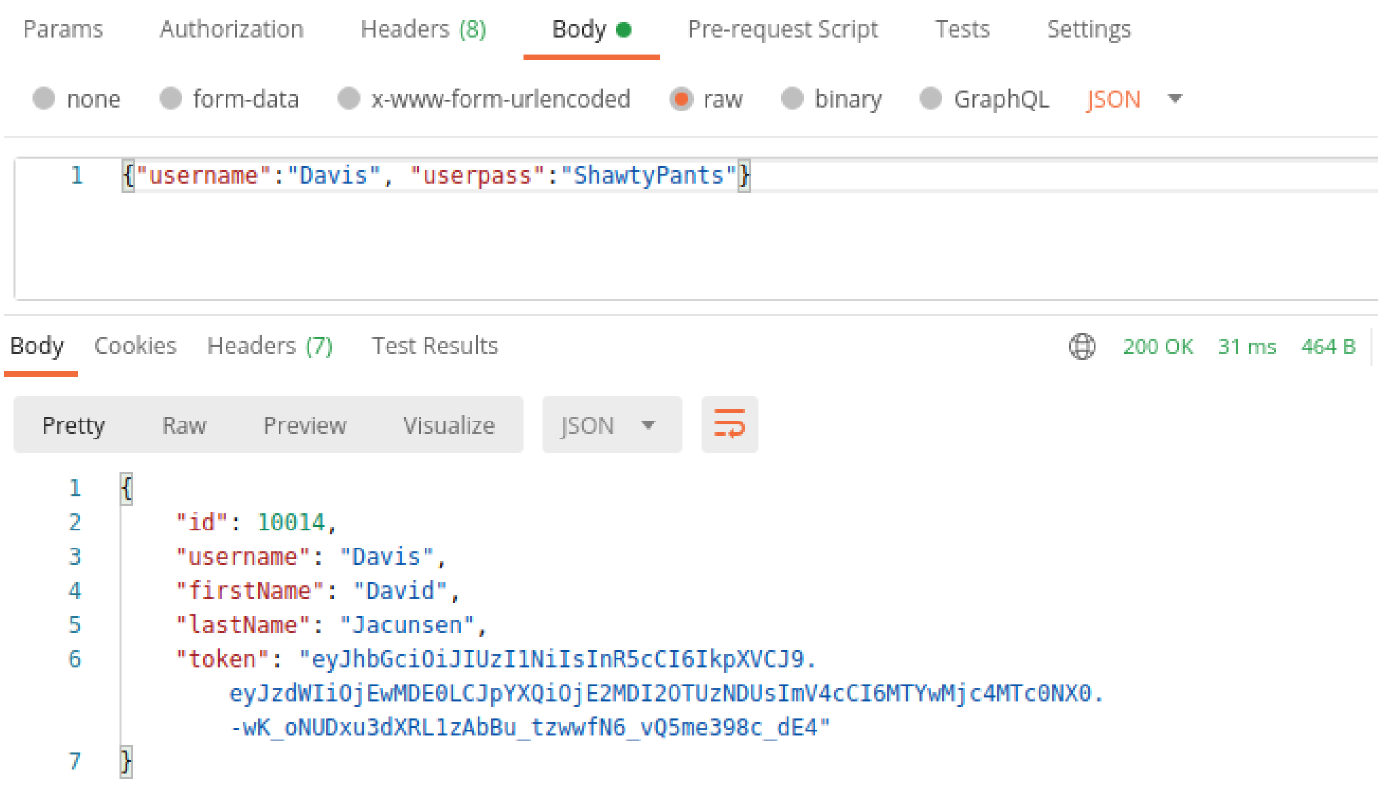
This request does not have a body

500 Internal Server Error 59 ms 399 B

```
<!DOCTYPE html>
<html>

<head>
    <title></title>
    <link rel="stylesheet" href="/stylesheets/style.css">
</head>

<body>
    <h1></h1>
    <h2></h2>
    <pre></pre>
</body>

</html>
```

Note that this is a 500 error, which probably means we didn't handle the failed authentication attempt properly. Let's see if adding an authentication key helps. (Note that I entered raw JSON in Postman.)

POST http://localhost:3000/auth/authenticate Send



```
{"username":"Davis", "userpass":"ShawtyPants"}
```

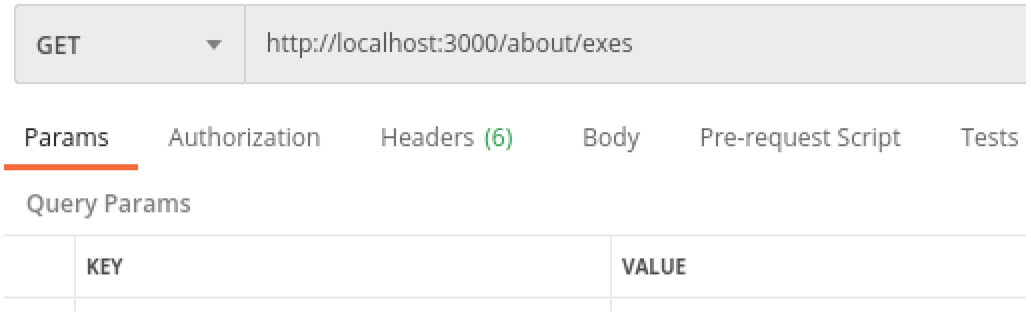
```
{
    "id": 10014,
    "username": "Davis",
    "firstName": "David",
    "lastName": "Jacunsen",
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJzdWIiOjEwMDE0LCJpYXQiOjE2MDI2OTUzNDUsImV4cCI6MTYwMjc4MTc0NX0.
        -wK_oNUDxu3dXRL1zAbBu_tzwwfN6_vQ5me398c_dE4"
}
```

Cool, 200 is better than OK. We've got simple authentication working and we know our 500 error was supposed to happen even if it doesn't look very nice! Let's test it by adding a protected route to "about". (In practice, you might want to keep protected and public routes entirely separate, but this should illustrate the concept of parsing authenticated routing by location.)

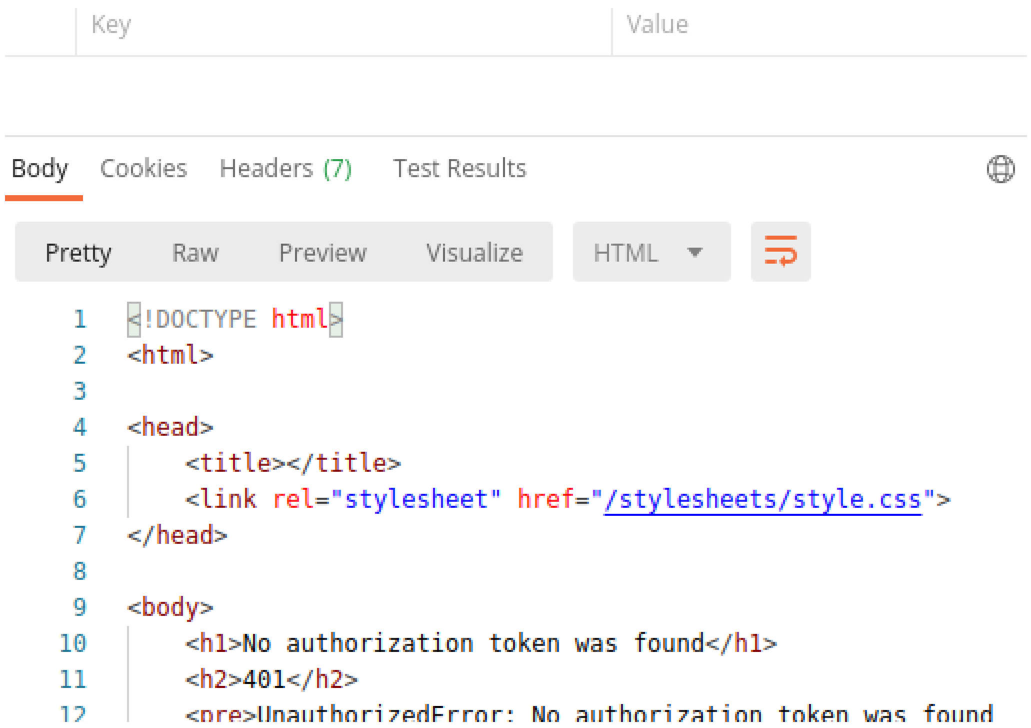
In about about.js router file, let's use essentially the same document grab we had in our api.js router. This should give us a route at /about/exes that is protected.

```
/* GET a simple listing of some predefined document type. */
router.get('/exes', async function(req, res, next) {
  let exes = await api.wholeSet();
  if (exes == undefined) {
    res.writeHead(404, {'Content-Type': 'text/plain'});
    res.end();
  } else {
    res.json(exes);
  }
});
```

Let's grab a new tab in Postman and try it. Make sure to keep your successful token request tab open. We'll need it in a minute, with any luck.



```
http://localhost:3000/about/exes
```

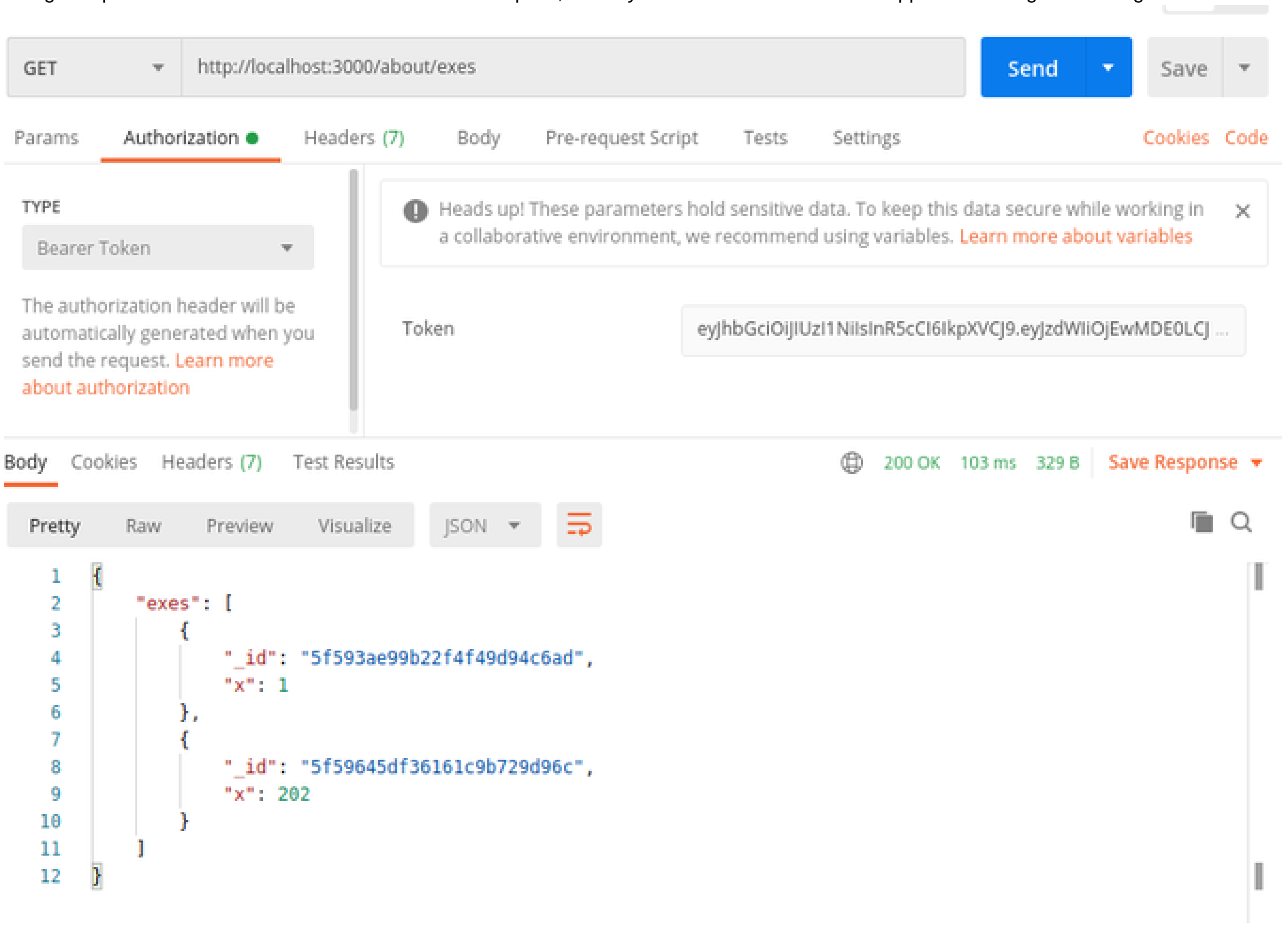

Key Value

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize HTML

```
<!DOCTYPE html>
<html>

<head>
    <title></title>
    <link rel="stylesheet" href="/stylesheets/style.css">
</head>

<body>
    <h1>No authorization token was found</h1>
    <h2>401</h2>
    <pre>UnauthorizedError: No authorization token was found
```

Pretty predictable response, as we have an authentication token we were provided, but don't have a way to tell our routers that we have it in a plain get request. Let's add the token information to the request, the way we would if we had a client application doing some lifting for us:


GET http://localhost:3000/about/exes Send Save

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies Code

TYPE: Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjEwMDE0LCJ ...

Body Cookies Headers (7) Test Results 200 OK 103 ms 329 B Save Response

Pretty Raw Preview Visualize JSON

```
{
    "exes": [
        {
            "_id": "5f593ae99b22f4f49d94c6ad",
            "x": 1
        },
        {
            "_id": "5f59645df36161c9b729d96c",
            "x": 202
        }
    ]
}
```

There we have it: our previous open API request is now a protected request. Sure, we don't have any RBAC working yet, but we do have a user we can now compare against a role matrix for authorization. We also don't have a front end yet, but we have enough infrastructure to which we can hook up just about any browser application. We’ll do that next. But let’s try to trap our 500 error, first. It should be simple to catch a few common error types.

Let’s create a new module for errors, probably called errors.js in our modules folder. Mine looks like this:

```
moodule.exports = errorHandler;

function errorHandler (err, req, res, next) {
    if (typeof (err) === 'string') {
        // custom application error
        return res.status(400).json({ message: err });
    }

    if (err.name === 'AuthenticationError') {
        return res.status(401).json({ name: err.name, message: err.
message });
    }

    if (err.name === 'UnauthorizedError' && err.message === 'invalid
token') {
        return res.status(401).json({ name: "AuthenticationError",
message: "Authentication Failed: Your token is corrupt or has expired."
});
    }

    // default to 500 server error
    return res.status(500).json({ message: err.message });
}
```

We need to add a directive in our app.js to use this module:

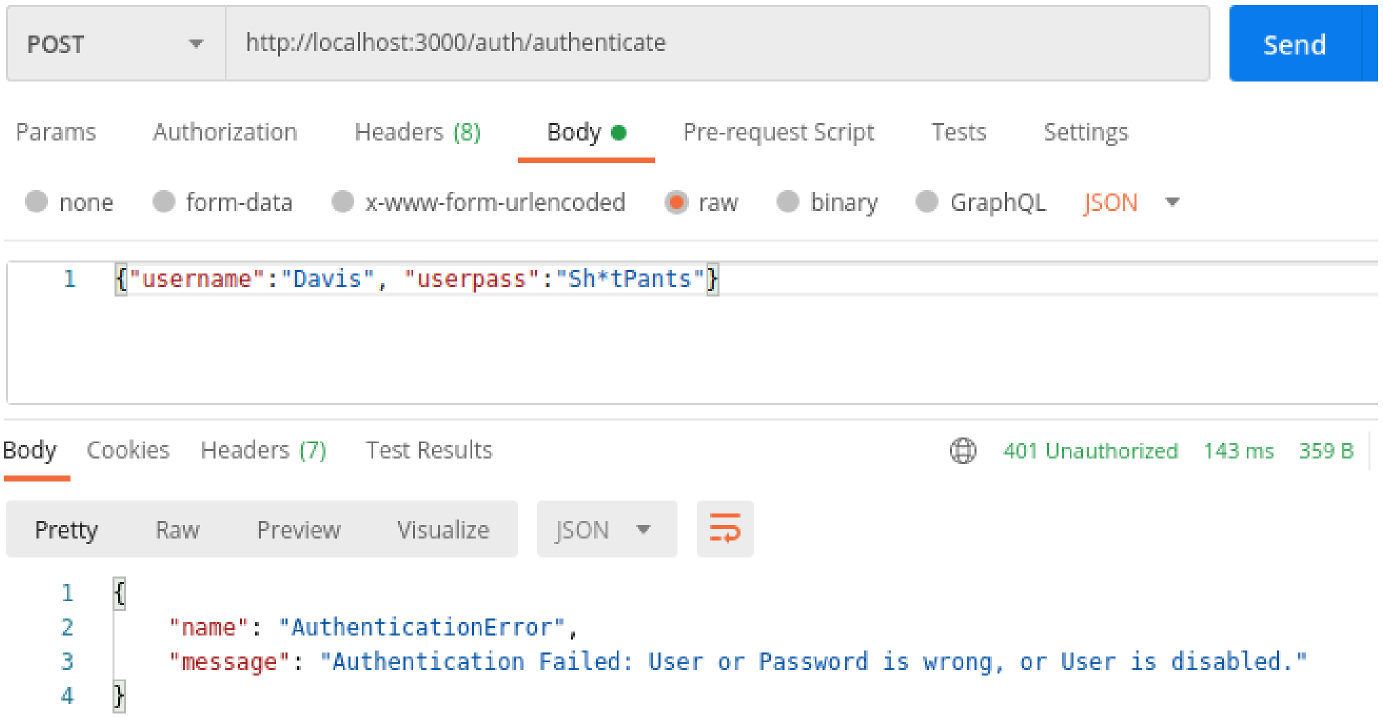
```
const errorHandler = require('./modules/errors');

// we're going to move this elsewhere to simplify here, and expand there
app.use(errorHandler)

/* this is old, deprecated in favor of the single line above for module
reference
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  next(createError(404));
});
// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page
```

```
    res.status(err.status || 500);
    res.render('error');
  });
  */
```

And let’s change our auth module to use the new Error handling constructs.

```
      let authError = new Error()
      if (!user) {
          authError.name ="AuthenticationError";
          authError.message = "Authentication Failed: User or Password is
wrong, or User is disabled.";
          throw authError;
      }
```

Now try with a request we know will make it fail:


POST http://localhost:3000/auth/authenticate

Send

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings

none form-data x-www-form-urlencoded ● raw binary GraphQL JSON

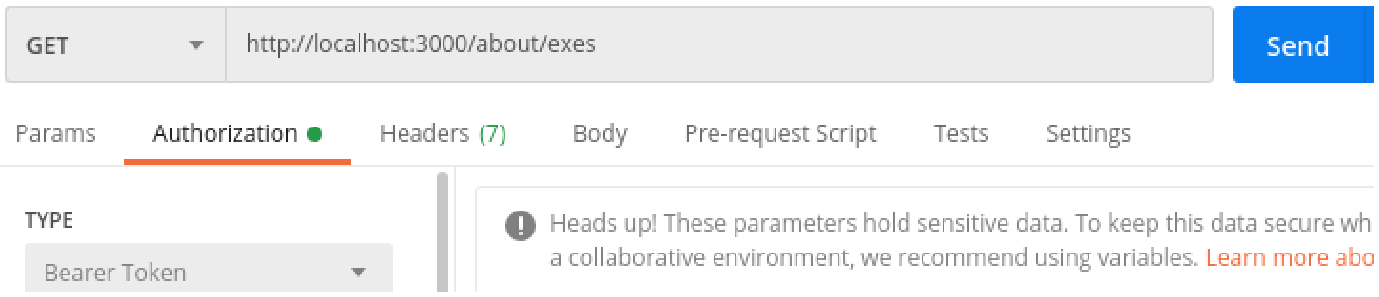
```
{"username":"Davis", "userpass":"Sh*tPants"}
```

Body Cookies Headers (7) Test Results 401 Unauthorized 143 ms 359 B

Pretty Raw Preview Visualize JSON

```
{
    "name": "AuthenticationError",
    "message": "Authentication Failed: User or Password is wrong, or User is disabled."
}
```

Now let’s intentionally break our token and see what happens:


GET http://localhost:3000/about/exes

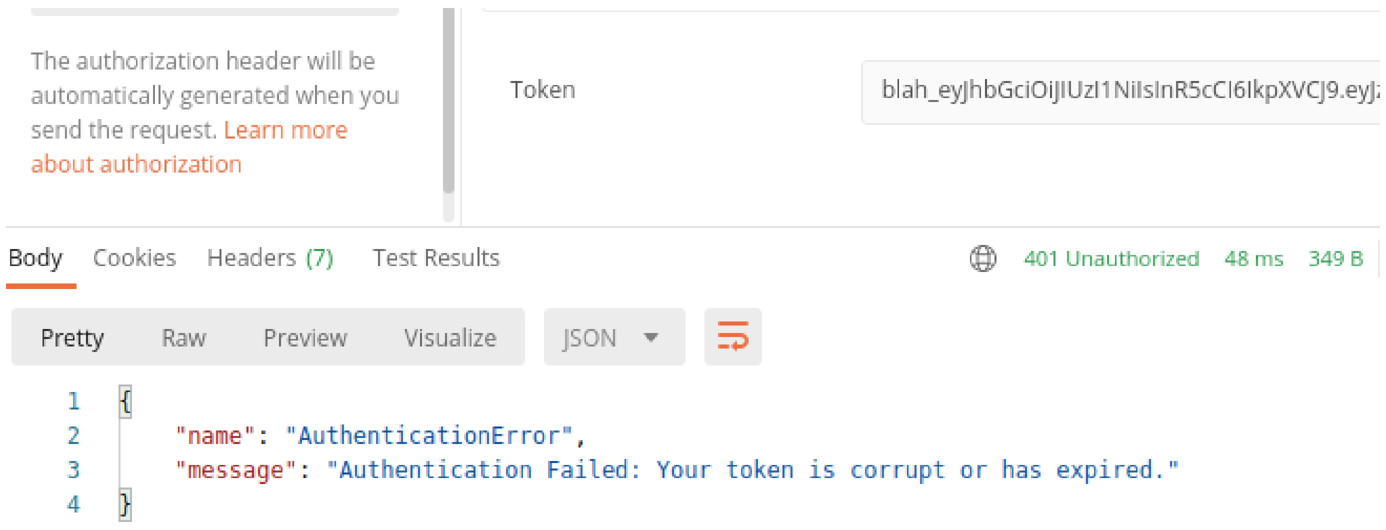
Send

Params Authorization ● Headers (7) Body Pre-request Script Tests Settings

TYPE

Bearer Token

Heads up! These parameters hold sensitive data. To keep this data secure wh
a collaborative environment, we recommend using variables. Learn more abo


The authorization header will be automatically generated when you send the request. Learn more about authorization

Token blah_eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ

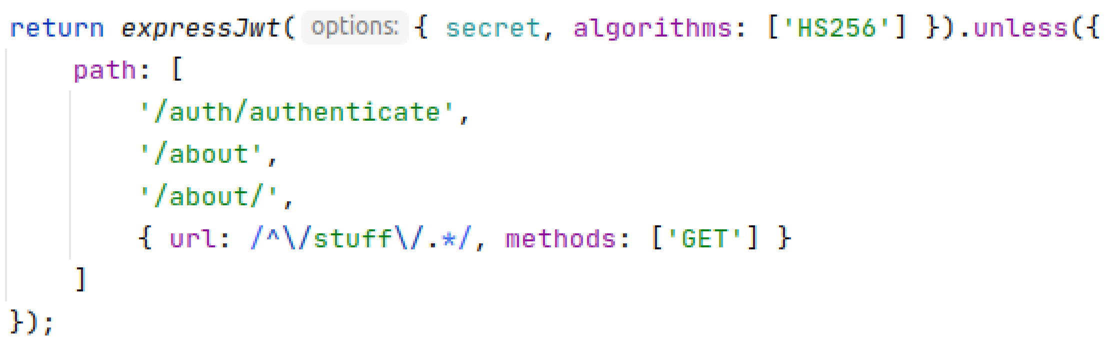
```
{
    "name": "AuthenticationError",
    "message": "Authentication Failed: Your token is corrupt or has expired."
}
```

It seems like we've got our error handling moving in the right direction. Maybe this is better, maybe not. The http-errors module is pretty robust, it seems. Let's move some of that error handling into our errors module as a fallback for our handler to use when we don't specify an error condition. Mine now looks like this, after adding the module requirement and modifying the fallback function to suit.

```
let createError = require('http-errors');

module.exports = errorHandler;

function errorHandler (err, req, res, next) {
    if (typeof (err) === 'string') {
        // custom application error
        return res.status(400).json({ message: err });
    }

    if (err.name === 'AuthenticationError') {
        return res.status(401).json({ name: err.name, message: err.
message });
    }

    if (err.name === 'UnauthorizedError' && err.message === 'invalid
token') {
        return res.status(401).json({ name: "AuthenticationError",
message: "Authentication Failed: Your token is corrupt or has expired."
});
    }

    handleAll(err, req, res, null)
    // default to 500 server error
    //return res.status(500).json({ message: err.message });
}

function handleAll (err, req, res, next) {
    // set locals, only providing error in development
    res.locals.message = err.message;
```

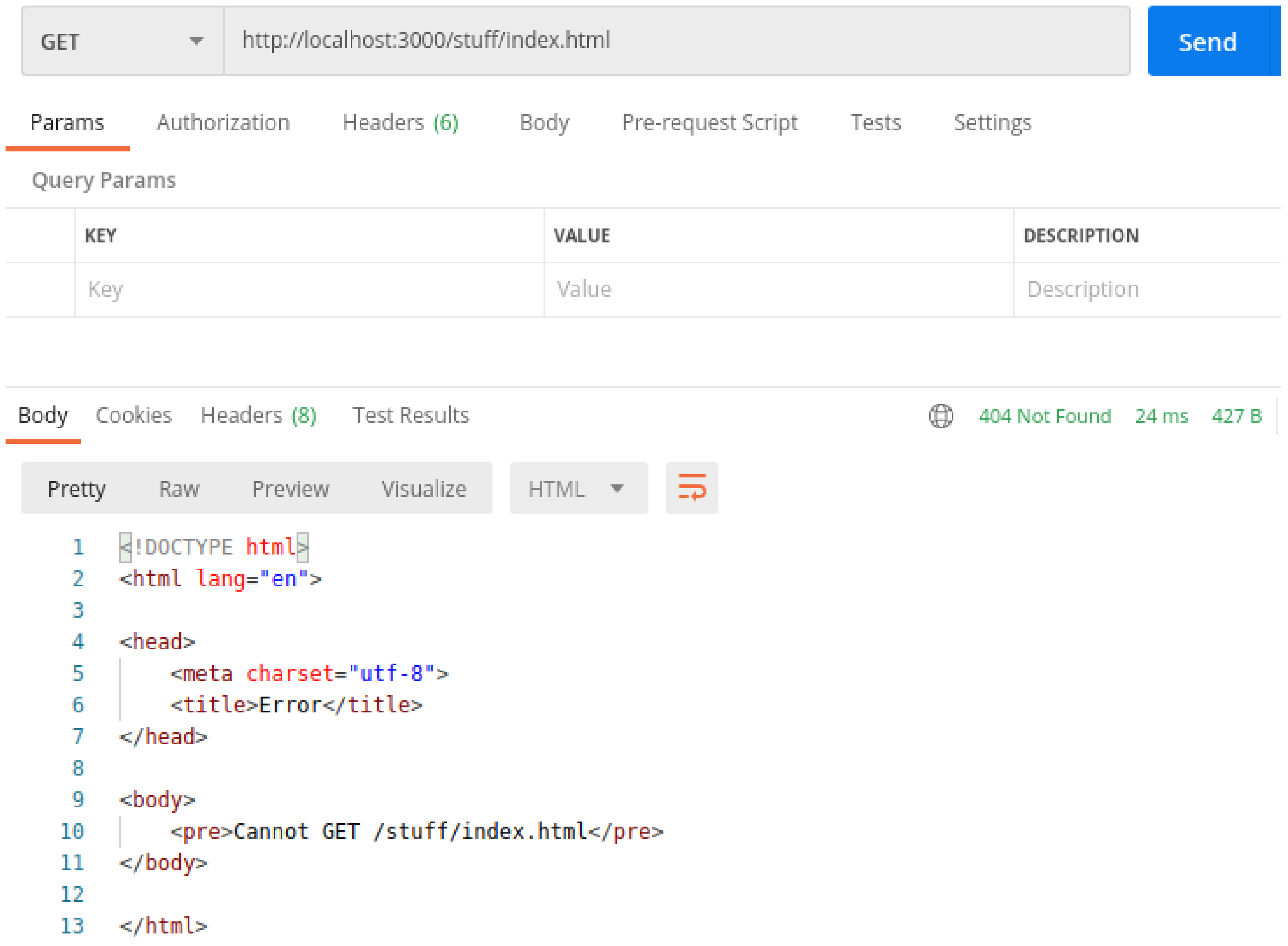
```
        res.locals.error = req.app.get('env') === 'development' ? err : {};

        // render the error page
        res.status(err.status || 500);
        res.render('error');
    }
```

To test this, we’re going to add an unprotected route (that’s recursively unprotected), and then try to get something we know isn’t there. In our jwt module, let’s add a new route with a regex that allows trailing stuff.



```
return expressJwt( options: { secret, algorithms: ['HS256'] }).unless({
    path: [
        '/auth/authenticate',
        '/about',
        '/about/',
        { url: /^\/stuff\/.*/, methods: ['GET'] }
    ]
});
```

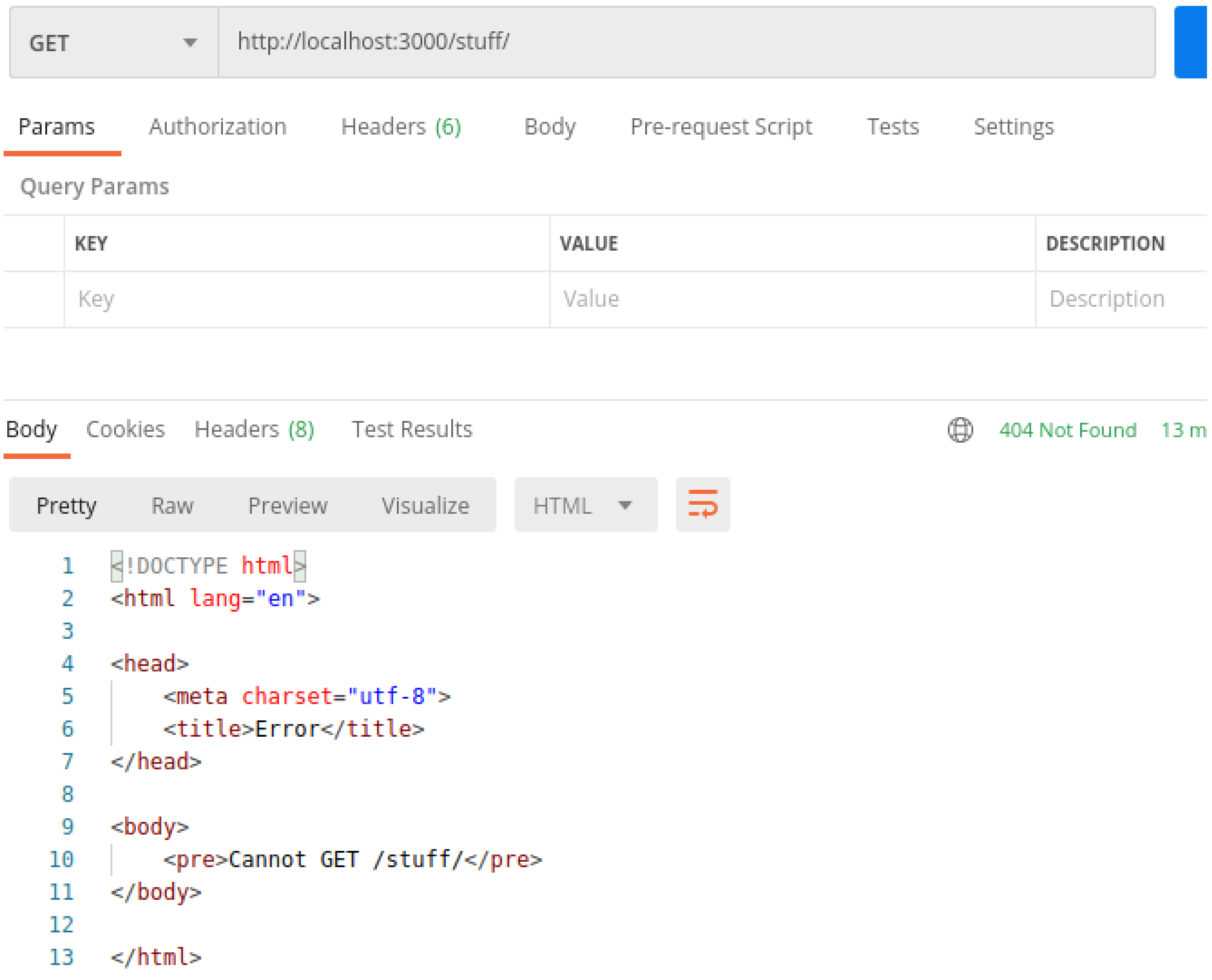
Test it with Postman:



	KEY	VALUE	DESCRIPTION
	Key	Value	Description

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <title>Error</title>
</head>

<body>
    <pre>Cannot GET /stuff/index.html</pre>
</body>

</html>
```

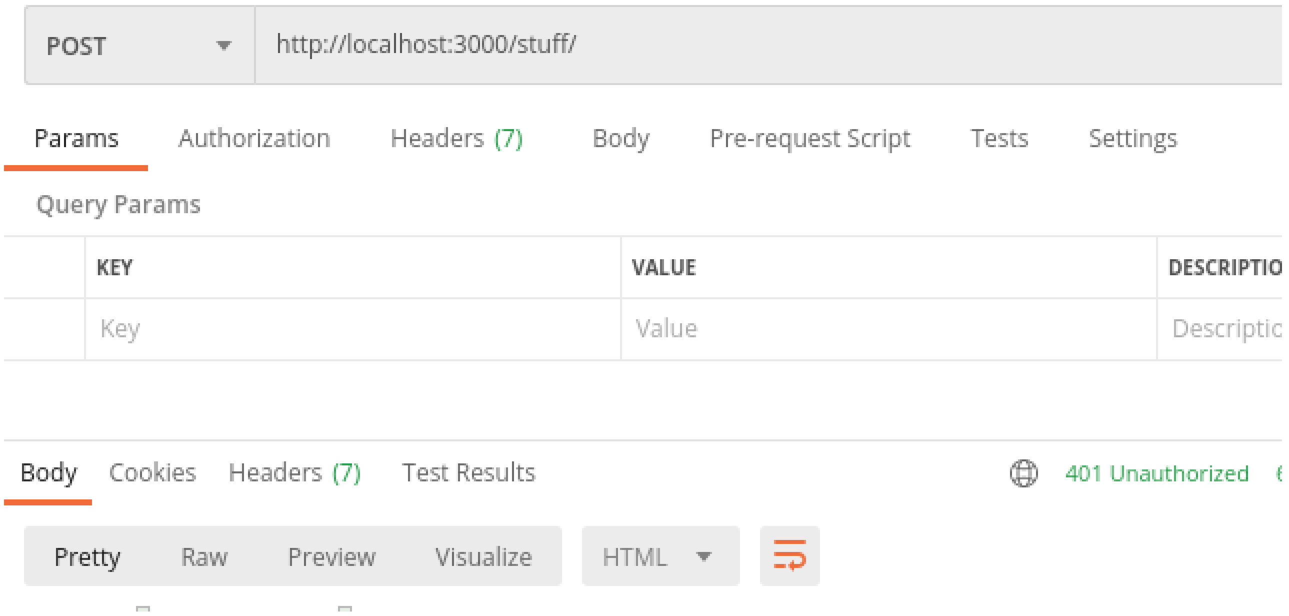
We got a 404, just like we expect. Let's try a get and post to the root of that path, just to see if our methods restriction worked in the unauthenticated route.



http://localhost:3000/stuff/

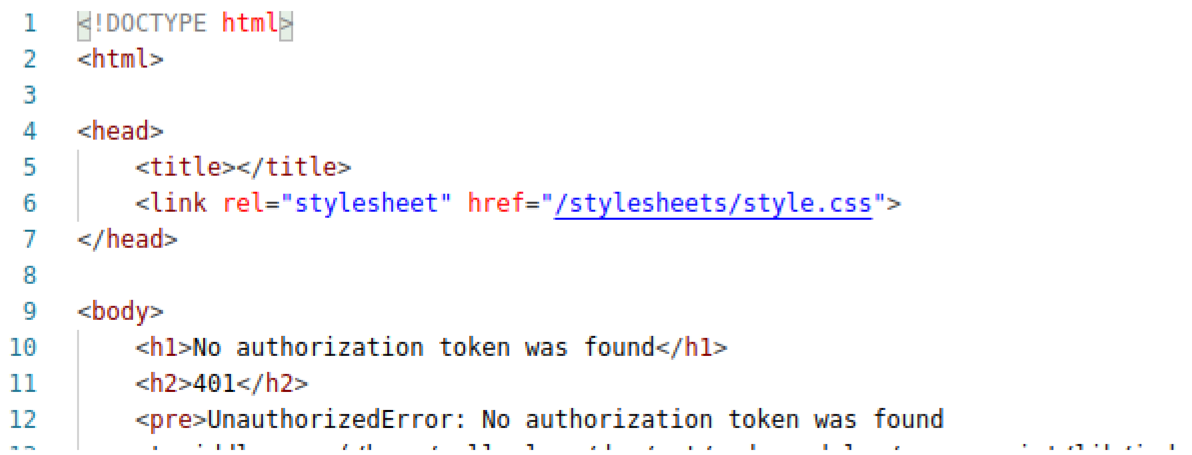
	KEY	VALUE	DESCRIPTION
	Key	Value	Description

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <title>Error</title>
</head>

<body>
    <pre>Cannot GET /stuff/</pre>
</body>

</html>
```



http://localhost:3000/stuff/

	KEY	VALUE	DESCRIPTIO
	Key	Value	Descriptic

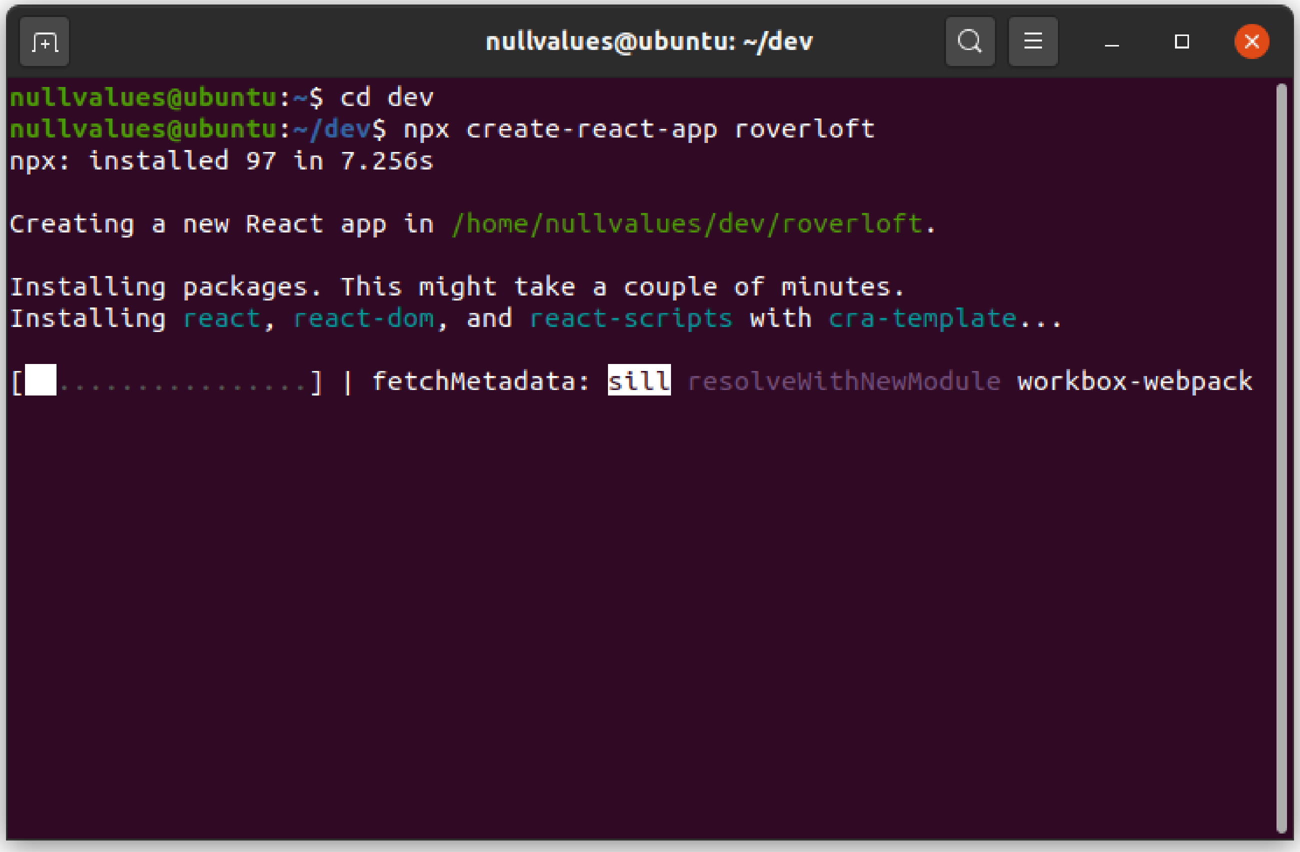

```
<!DOCTYPE html>
<html>

<head>
    <title></title>
    <link rel="stylesheet" href="/stylesheets/style.css">
</head>

<body>
    <h1>No authorization token was found</h1>
    <h2>401</h2>
    <pre>UnauthorizedError: No authorization token was found
```

Get gives us a 404, Post gives us a 401. Looks right.

So there you have it, an authenticated API using JWT that can be hooked up to a front end. This JWT section legitimately took me about five or six hours of dedicated work, but you could follow the steps in less time now that I’ve done the troubleshooting, and assuming you don’t hit problems of your own.

Adding a React Front End

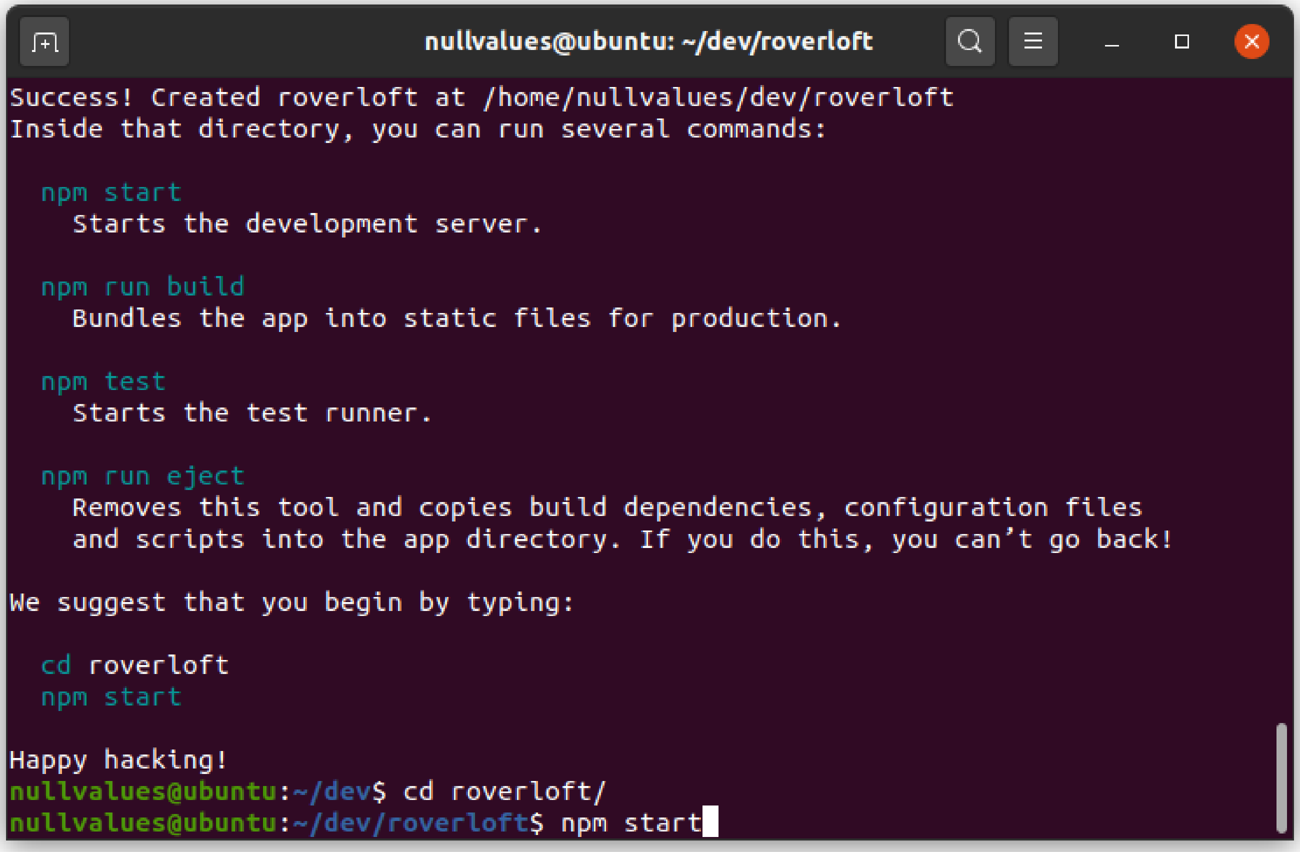
React offers a a single-command (npx create-react-app [appname]) setup for a new application front end which you can use as a jumping off point for all manner of sins. Let’s do it for a project we’ll call React Over LOFT (roverloft), asssuming we’ll use this to model our LOFT prototype.



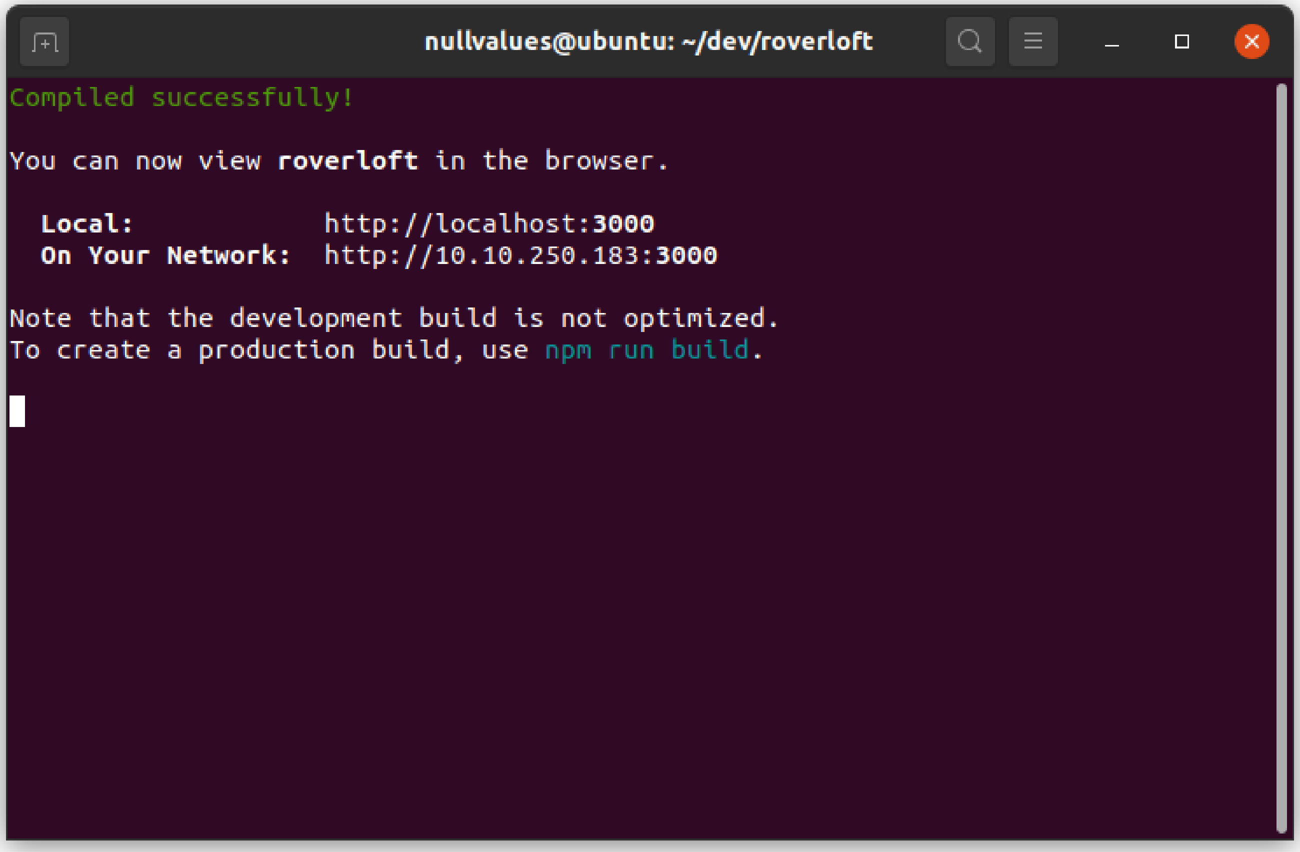
```
nullvalues@ubuntu:~$ cd dev
nullvalues@ubuntu:~/dev$ npx create-react-app roverloft
npx: installed 97 in 7.256s

Creating a new React app in /home/nullvalues/dev/roverloft.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

[█................] | fetchMetadata: sill resolveWithNewModule workbox-webpack
```

Simple, but kinda has a developed by ex-PHP developers feel to it. It’s heavy (takes minutes to run). Let’s see how this goes; might be great, because let’s be fair, it’s creating the scaffold and installing it. That’s cool.

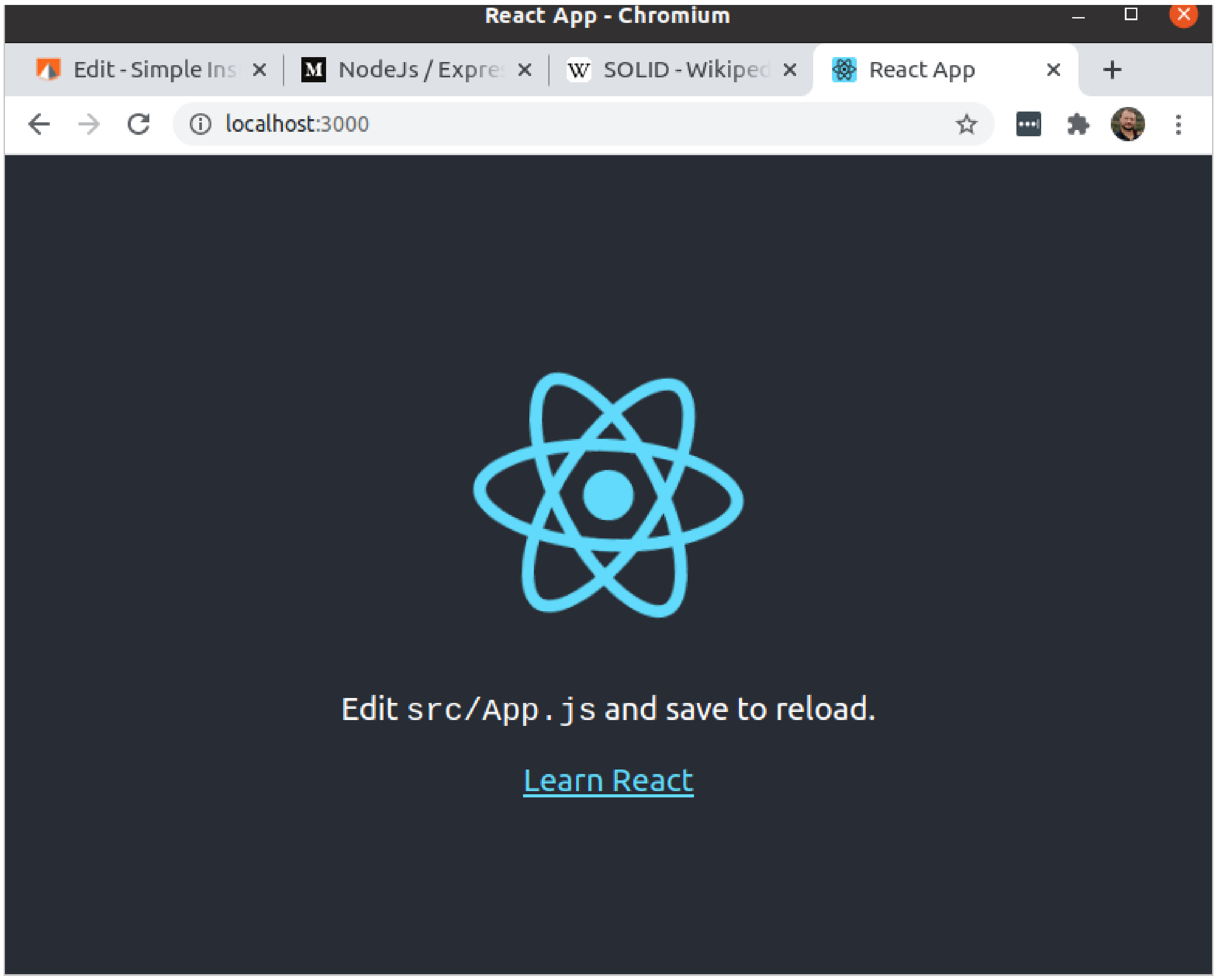
We start it with:



```
Success! Created roverloft at /home/nullvalues/dev/roverloft
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can’t go back!

We suggest that you begin by typing:

  cd roverloft
  npm start

Happy hacking!
nullvalues@ubuntu:~/dev$ cd roverloft/
nullvalues@ubuntu:~/dev/roverloft$ npm start
```



```
Compiled successfully!

You can now view roverloft in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://10.10.250.183:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```



I wish I'd had this when I was 25. Cool. Let's see what we've got. They said to look in App.js, but like every good webserver, we start with a file called index.html in the root. This is actually in public. Can't do much there, but did change the name to React over LOFT so everyone knows it's custom and I can feel a sense of ownership.

I looked at index.js, and it looks simple too, just injects a rendered app into the root element in the index.html file. There is an item in there for later thought: serviceWorker.register(), which we might want for offline access to our stuffs. What's interesting is that this scaffold doesn't rely on JSX, which seems to be the React darling of the moment, and also what our demo react-jwt is based on. Let's just standardize on JSX to make the comparisons easy. Let's rename index.js to index.js.old, and then we'll create an index.jsx file and and add a simple render element that should do the same job.

Mine looks like this:

```
import React from 'react';
import { render } from 'react-dom';

import { Loft } from './Loft';

render(
    <Loft />,
    document.getElementById('root')
);
```

And let’s create a home for our app that’s not just “App”, so we get some context. Let’s call it “Loft”. We’ll create a blank Loft.jsx and index.js in that folder.

In the index.js, do an export of the Loft jsx file.

```
export * from './Loft'
```

We’re going to pretty much cold spike the React-JWT example application module, and reduce it some to start so we can get our arms around React. We’ll start here, and hope it works:

```
import React from 'react';
import { Router, Route, Link } from 'react-router-dom';
import { LoginPage } from '../LoginPage';

import { createBrowserHistory } from 'history';
export const history = createBrowserHistory();

class Loft extends React.Component {

    render() {
        return (
            <Router history={history}>
                <div>
                    <p>Appropriate header message here...</p>
                    <Route path="/login" component={LoginPage} />
                </div>
            </Router>
        );
    }
}

export { Loft };
```

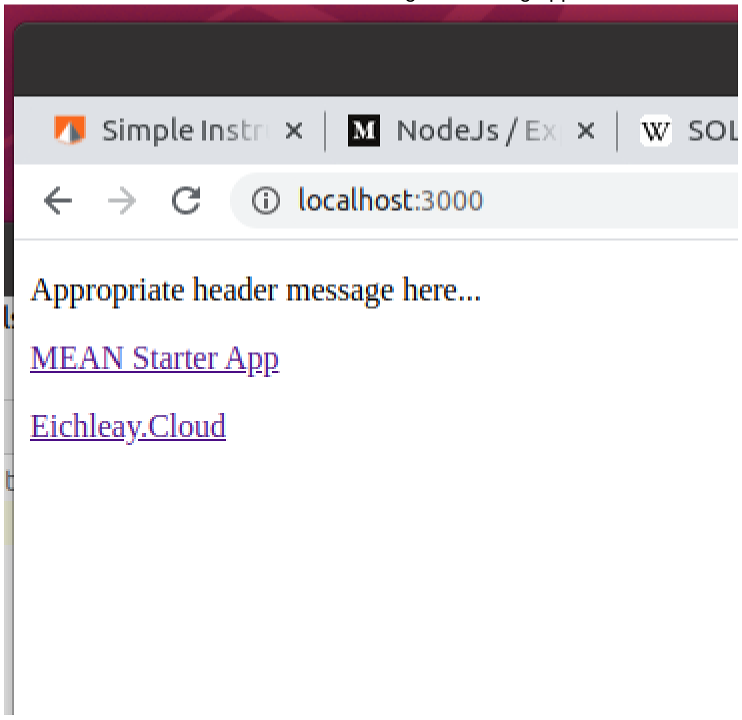
You’ll notice that we need a login directory with same setup as we had for the Loft directory. The index.js just exports the JSX module.

```
export * from './LoginPage'
```

And the LoginPage.jsx we’re going to cold spike from our buddy Jason too, just to see what we can get going. You’ll notice that we need to use npm to install formik and yup. I did this by just clicking on them in webstorm and installing the dev dependency. npm install [thename] should do the same. We’re going to start with just some plain old scrool HTML and see what renders.

```
import React from 'react';
import { Formik, Field, Form, ErrorMessage } from 'formik';
import * as Yup from 'yup';

class LoginPage extends React.Component {
    render() {
        return (
            <div>
                <h2>Login</h2>
                <table>
                    <tr>
                        <td>Username</td><td><input type={<textarea
name="username" id="username" cols="30" rows="1"></textarea>}/></td>
                    </tr>
                    <tr>
                        <td>Password</td><td><input type={<textarea
name="userpass" id="userpass" cols="30" rows="1"></textarea>}/></td>
                    </tr>
                    <tr>
                        <td colSpan="2"><input type="button" name="
submit" value="Login"/></td>
                    </tr>
                </table>
            </div>
        )
    }
}

export { LoginPage };
```

We start Loft and at the document root we get a working application and not much else.

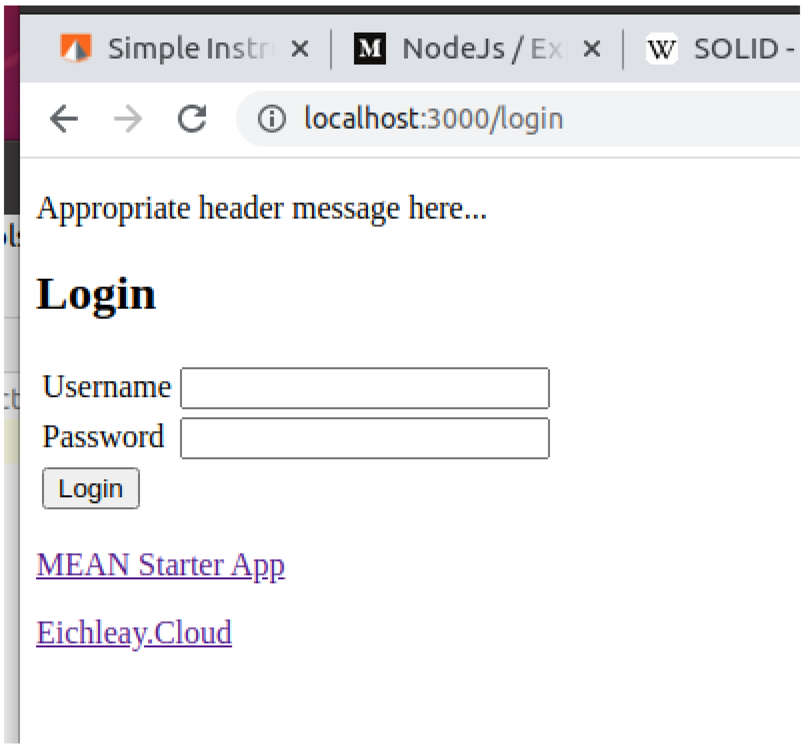


Appropriate header message here...

MEAN Starter App

Eichleay.Cloud

That seems right. Let’s test the only route we have:



Appropriate header message here...

Login

Username

Password

Login

MEAN Starter App

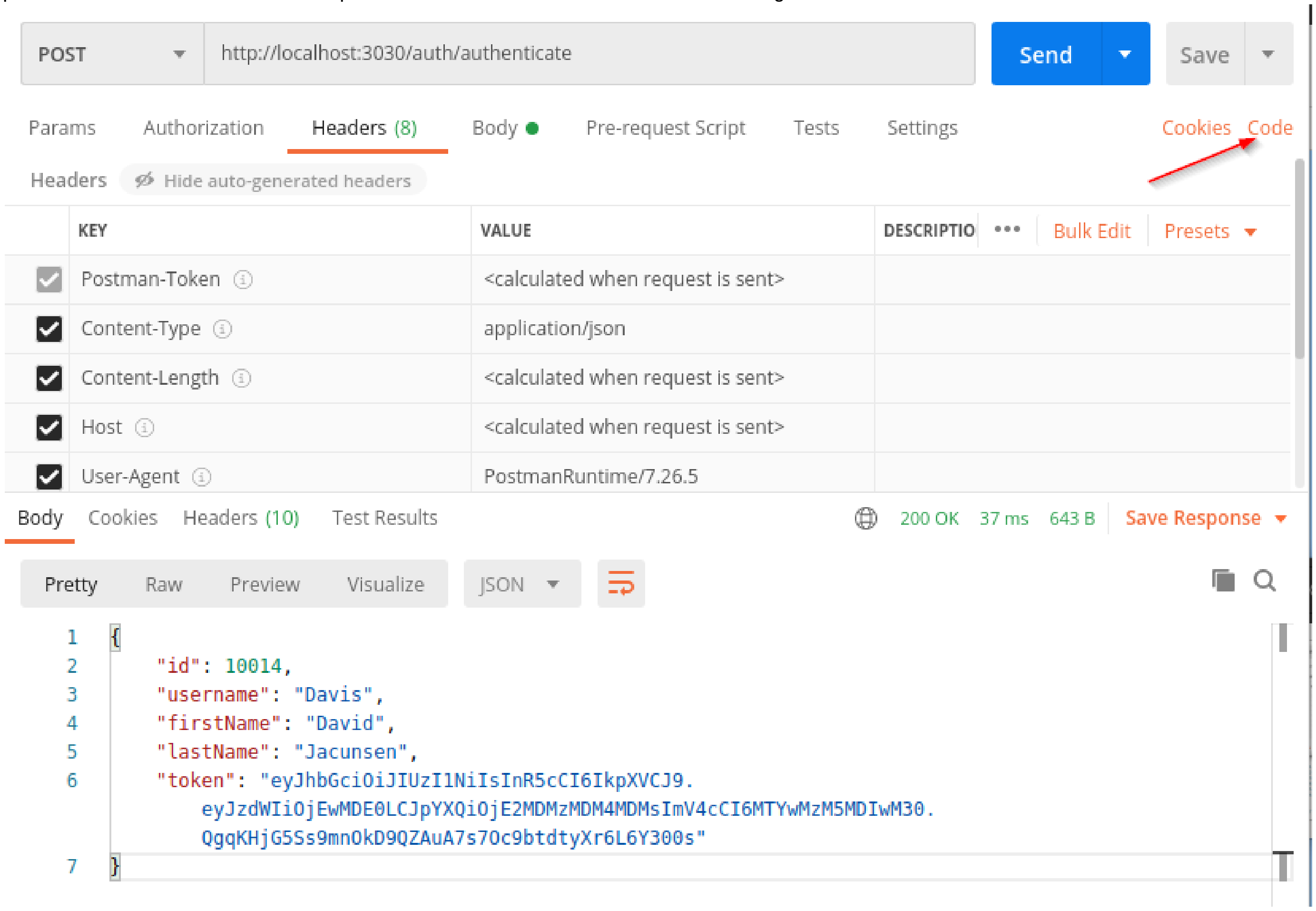
Eichleay.Cloud

That looks pretty promising for a single page app, as you can probably guess by now that we can wrap in some conditionals (like login status) and control what's displayed. We're at the point where we need to be able to hook up to an authentication service to provide our credentials and handle the authentication response. I like the conventions Mr. Watmore used, prefacing internal modules with and underbar, and creating a folder each for services and helpers. Let's create a "_service" folder so we can start working out authentication details. Then create a JS file within called authentication.service.js. Personally, I like longer names in cases like this, as many applications deal with both authentication and authorization, so while it was tempting to shorten to just auth.js, Watmore's convention makes sense to me. In order to make this authentication services usable, you'll need an export statement (`export * from './authentication.service';`) in an index.js file in the services directory as well.

Make sure to do an *npm install rxjs* to make sure the import works. Here's how mine looks:

```
import { BehaviorSubject } from 'rxjs';

let apiUrl = "http://localhost:3030";

const currentUserSubject = new BehaviorSubject(JSON.parse(localStorage.
getItem('currentUser')));

export const authenticationService = {
    login,
    logout,
    currentUser: currentUserSubject.asObservable(),
    get currentUserValue () { return currentUserSubject.value }
};

function login(username, userpass) {

    let myHeaders = new Headers();
    myHeaders.append("Content-Type", "application/json");

    let raw = JSON.stringify({"username":username,"userpass":userpass});
    let requestOptions = {
        method: 'POST',
        headers: myHeaders,
        body: raw,
        redirect: 'follow'
    };

    return fetch(`${apiUrl}/auth/authenticate`, requestOptions)
        .then(handleResponse)
        .then(user => {
            // store user details and jwt token in local storage to
keep user logged in between page refreshes
            localStorage.setItem('currentUser', JSON.stringify(user));
            currentUserSubject.next(user);

            return Promise.resolve(user);
        })
        .catch(error => console.log('error', error));
}

function logout() {
    // remove user from local storage to log user out
    localStorage.removeItem('currentUser');
    currentUserSubject.next(null);
}

export function handleResponse(response) {

    return response.text().then(text => {
        const data = text && JSON.parse(text);
        if (!response.ok) {
            /* I'm not sure I agree with this approach...let's just
comment out for now and see if there's
               some truth to the need.  I can't think of a reason that
an authorization restriction should
               *force* a logout.  But maybe there's more to this we'll
discover later.
            if ([401, 403].indexOf(response.status) !== -1) {
                // auto logout if 401 Unauthorized or 403 Forbidden
response returned from api
                authenticationService.logout();
                window.location.reload();
            }
            */

            const error = (data && data.message) || response.statusText;
            return Promise.reject(error);
        } else {
        }

        return data;
    });
}
```

This is largely borrowed from Jason's work as it's simple and hard to improve on right now. I did use a couple of shortcuts to prove to myself I knew how it worked. For one, I used postman to generate the Javascript Fetch statement for the login function, and then modified to use our passed variables. It also lacked a couple of Promise return values which I had to debug.



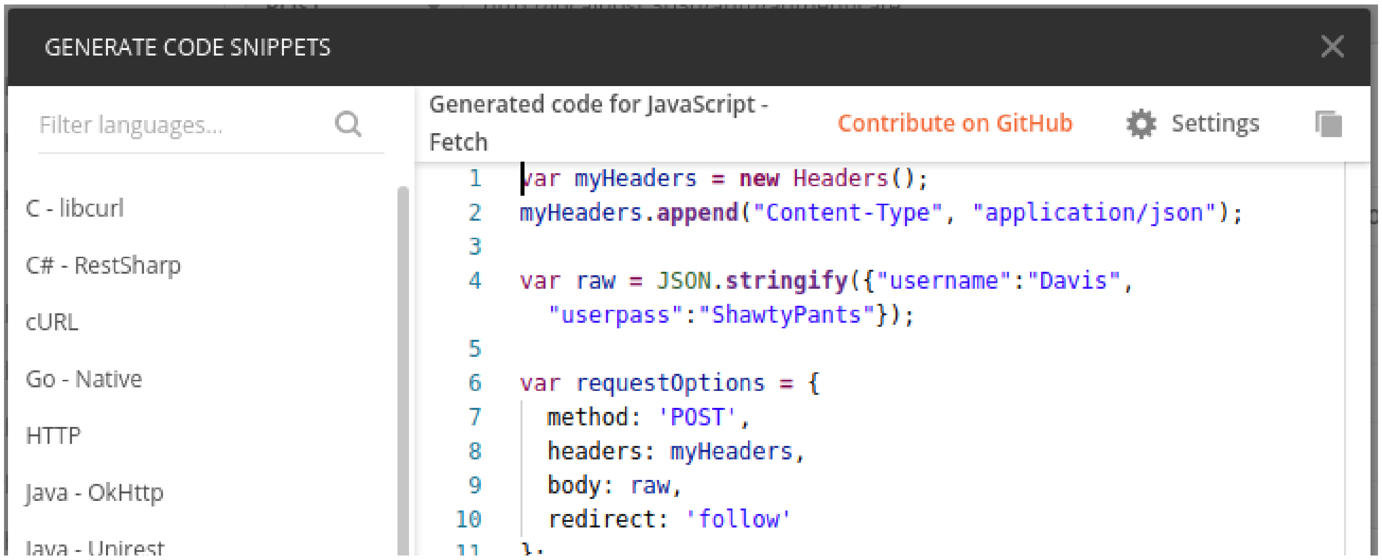
POST http://localhost:3030/auth/authenticate

Headers (8)

	KEY	VALUE
✓	Postman-Token	<calculated when request is sent>
✓	Content-Type	application/json
✓	Content-Length	<calculated when request is sent>
✓	Host	<calculated when request is sent>
✓	User-Agent	PostmanRuntime/7.26.5

200 OK 37 ms 643 B

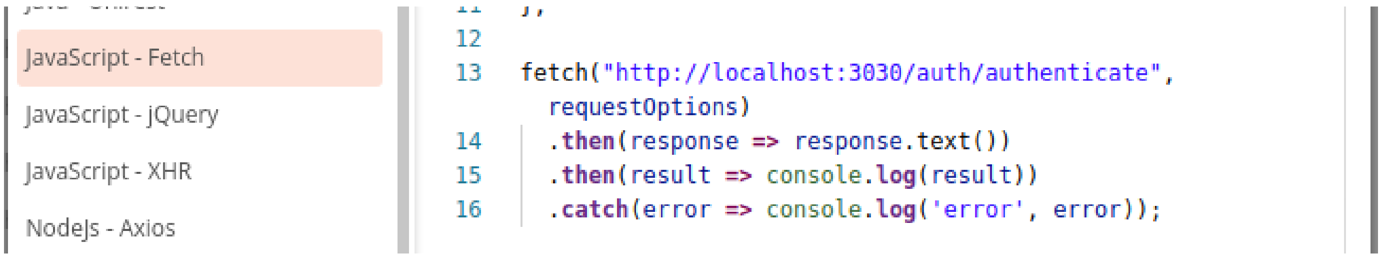
```
{
    "id": 10014,
    "username": "Davis",
    "firstName": "David",
    "lastName": "Jacunsen",
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJzdWIiOjEwMDE0LCJpYXQiOjE2MDMzMDM4MDMsImV4cCI6MTYwMzM5MDIwM30.
        QgqKHjG5Ss9mn0kD9QZAuA7s7Oc9btdtyXr6L6Y300s"
}
```



GENERATE CODE SNIPPETS

Generated code for JavaScript - Fetch

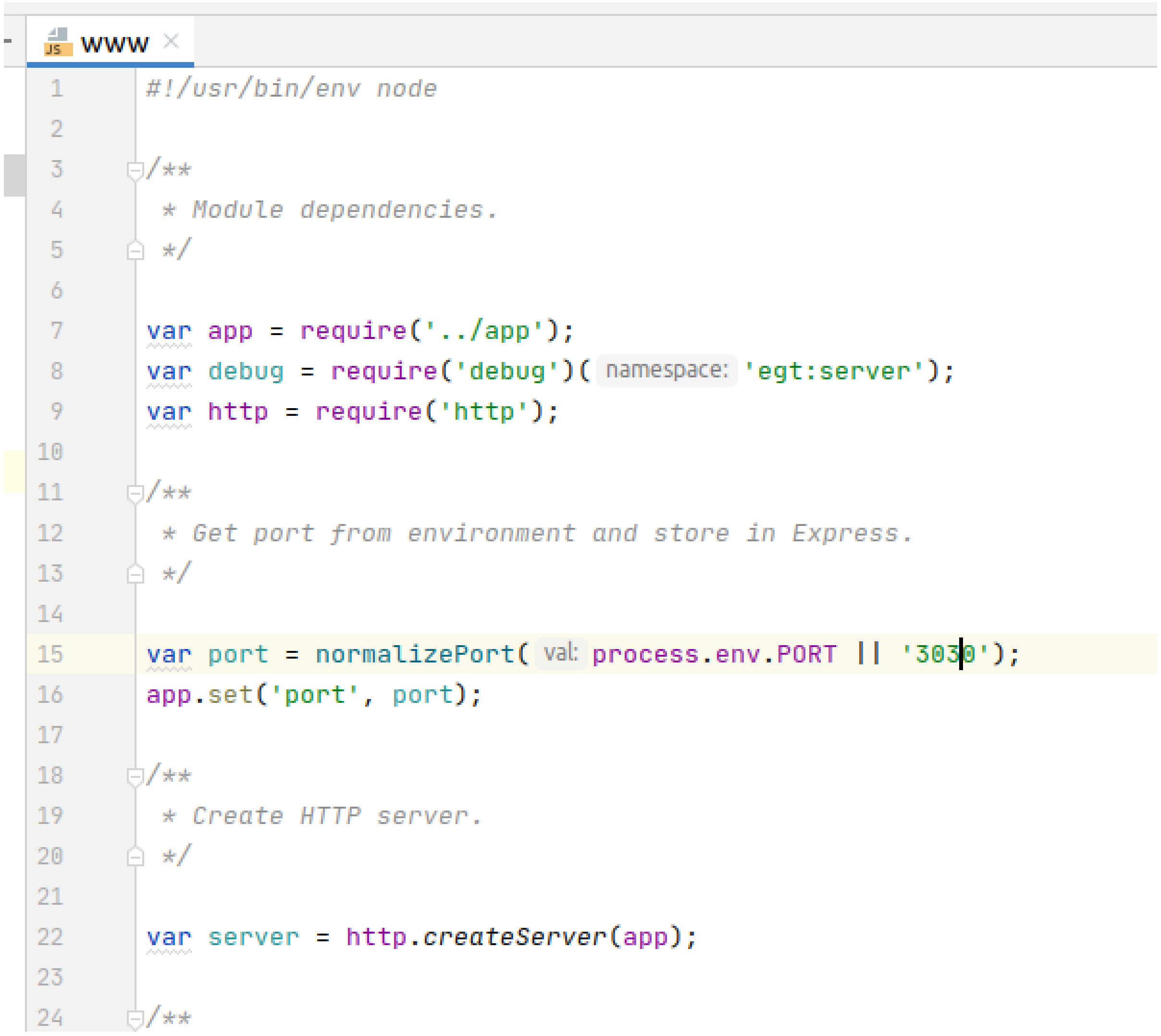
```
var myHeaders = new Headers();
myHeaders.append("Content-Type", "application/json");

var raw = JSON.stringify({"username":"Davis",
  "userpass":"ShawtyPants"});

var requestOptions = {
  method: 'POST',
  headers: myHeaders,
  body: raw,
  redirect: 'follow'
};
```



```
};

fetch("http://localhost:3030/auth/authenticate",
  requestOptions)
  .then(response => response.text())
  .then(result => console.log(result))
  .catch(error => console.log('error', error));
```

I did also simplify his example slightly so we don't depend on an external config file and response handler. And I did make one change that's significant: it affects our previous API server. You've probably noticed by now that we have a port collision brewing, as both of these applications run on 3000 by default. Let's let our React front end run on 3000, and we'll change our API to 3030 (as reflected in apiUrl variable).

In our API application, in the bin/www file, make the following change:



```
#!/usr/bin/env node

/**
 * Module dependencies.
 */

var app = require('../app');
var debug = require('debug')( namespace: 'egt:server');
var http = require('http');

/**
 * Get port from environment and store in Express.
 */

var port = normalizePort( val: process.env.PORT || '3030');
app.set('port', port);

/**
 * Create HTTP server.
 */

var server = http.createServer(app);

/**
```

Now, API will start up on 3030 for our prototype. We'll move this to a normal port later (or more likely a virtual server with a fully qualified url on default port of 80).

We need to work backward for a second here: we need a place for an authenticated user to land. We'll need two things:

- A route in our Loft.jsx. This will be our default "home", so we'll use "/" as the route.
- A matching module for that route. I'm going to call mine Root, and create a matching folder, index.js (with export statement) and Root.jsx as we did for the Loft module.

I've simplified some imports that are probably better normalized and imported as in Jason's example, but aren't necessary at this point in this prototype. Here's my Loft module now:

```
import React from 'react';
import {Router, Route, Link, Redirect} from 'react-router-dom';
import { LoginPage } from '../LoginPage';
import { Root } from '../Root';
import { authenticationService } from '../_services';
import { createBrowserHistory } from 'history';

export const history = createBrowserHistory();
export const PrivateRoute = ({ component: Component, ...rest }) => (
    <Route {...rest} render={props => {
        const currentUser = authenticationService.currentUserValue;
        if (!currentUser) {
            // not logged in so redirect to login page with the return
url
            return <Redirect to={{ pathname: '/login', state: { from:
props.location } }} />
        }

        // authorised so return component
        return <Component {...props} />
    }} />
)

class Loft extends React.Component {

    render() {
        return (
            <Router history={history}>
                <div>
                    <p>Appropriate header message here...</p>
                    <PrivateRoute exact path="/" component={Root} />
                    <Route path="/login" component={LoginPage} />
                </div>
            </Router>
        );
    }
}

export { Loft };
```

And here’s a really simple Root module to match the route:

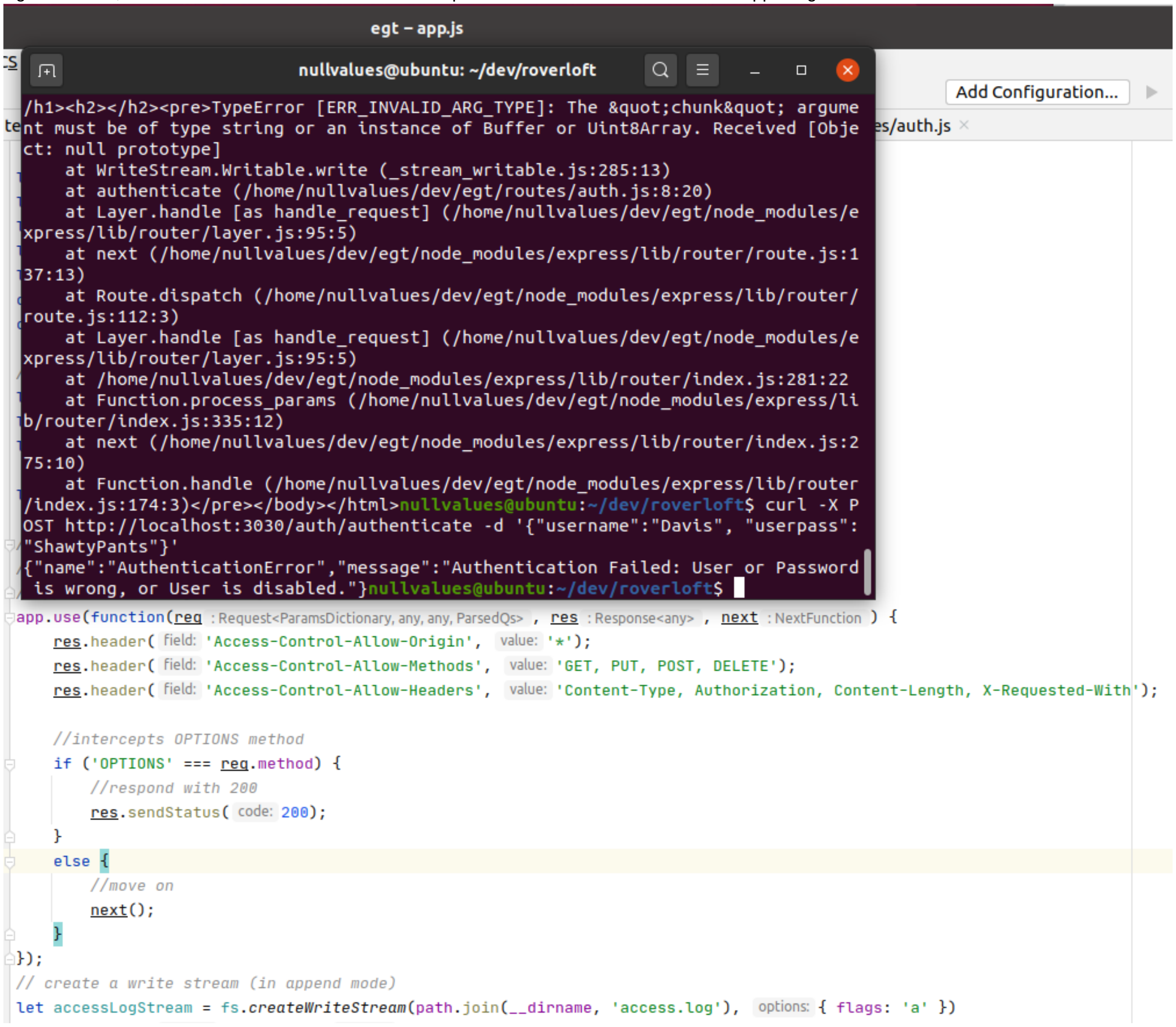
```
import React from 'react';

class Root extends React.Component {

    render() {
        return (
            <div>
```

```
                Authenticated Application Root
            </div>
        );
    }
}

export { Root };
```

The last change we need to make is to make our LoginPage reactive to the submission request. Watmore uses something called Formik to handle the form rendering and processing, which looks nice, but I was hoping for something a bit simpler to start with. That markup seems to get a bit further into the weeds for a prototype than I'd like, let's see if we can connect our form to the authentication service with some basic Javascript and Old School HTML. This is my LoginPage.jsx:

```
import React from 'react';
import { authenticationService } from '../_services';

class LoginPage extends React.Component {
    constructor(props) {
        super(props);

        // redirect to home if already logged in
        if (authenticationService.currentUserValue) {
            this.props.history.push('/');
        }
    }
    render() {
        return (
            <div>
                <h2>Login</h2>
                <table>
                    <tbody>
                    <tr>
                        <td>Username</td><td><input type="textarea"
name="username" id="username" cols="30" rows="1"/></td>
                    </tr>
                    <tr>
                        <td>Password</td><td><input type="password"
name="userpass" id="userpass" cols="30" rows="1"/></td>
                    </tr>
                    <tr>
                        <td colSpan="2"><input type="button" value="
Login" onClick={() => {
                            let user = document.getElementById
('username').value;
                            let pass = document.getElementById
('userpass').value;
                            authenticationService.login(user, pass)
                                .then(
                                    user => {
                                        const { from } = this.props.
location.state || { from: { pathname: "/" } };
                                        this.props.history.push(from);
                                    }
                                );
                        }}/></td>
                    </tr>
                    </tbody>
                </table>
            </div>
        )
    }
}

export { LoginPage };
```

Let's face it, I had some trouble here getting React to sync up with the API. The Login function wasn't being understood as a JSON POST, so I was getting a 200 response from an OPTIONS method. Puzzling for a few hours. As a result I improved the logging a bit so I could tail the access log on the API, as well as added some Access Control options so I could discern what was happening.

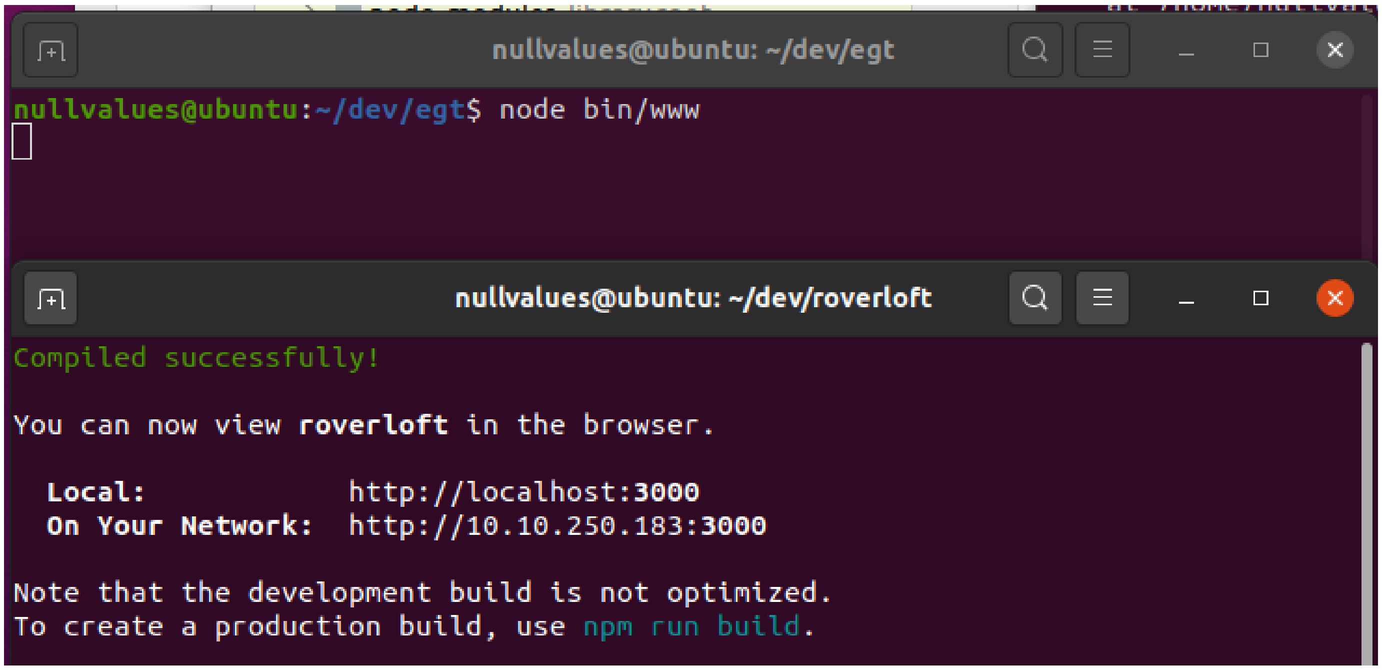


```
/h1><h2></h2><pre>TypeError [ERR_INVALID_ARG_TYPE]: The &quot;chunk&quot; argume
nt must be of type string or an instance of Buffer or Uint8Array. Received [Obje
ct: null prototype]
    at WriteStream.Writable.write (_stream_writable.js:285:13)
    at authenticate (/home/nullvalues/dev/egt/routes/auth.js:8:20)
    at Layer.handle [as handle_request] (/home/nullvalues/dev/egt/node_modules/e
xpress/lib/router/layer.js:95:5)
    at next (/home/nullvalues/dev/egt/node_modules/express/lib/router/route.js:1
37:13)
    at Route.dispatch (/home/nullvalues/dev/egt/node_modules/express/lib/router/
route.js:112:3)
    at Layer.handle [as handle_request] (/home/nullvalues/dev/egt/node_modules/e
xpress/lib/router/layer.js:95:5)
    at /home/nullvalues/dev/egt/node_modules/express/lib/router/index.js:281:22
    at Function.process_params (/home/nullvalues/dev/egt/node_modules/express/li
b/router/index.js:335:12)
    at next (/home/nullvalues/dev/egt/node_modules/express/lib/router/index.js:2
75:10)
    at Function.handle (/home/nullvalues/dev/egt/node_modules/express/lib/router
/index.js:174:3)</pre></body></html>nullvalues@ubuntu:~/dev/roverloft$ curl -X P
OST http://localhost:3030/auth/authenticate -d '{"username":"Davis", "userpass":
"ShawtyPants"}'
{"name":"AuthenticationError","message":"Authentication Failed: User or Password
 is wrong, or User is disabled."}nullvalues@ubuntu:~/dev/roverloft$
```

```
app.use(function(req :Request<ParamsDictionary, any, any, ParsedQs> , res :Response<any> , next :NextFunction ) {
    res.header( field: 'Access-Control-Allow-Origin',  value: '*');
    res.header( field: 'Access-Control-Allow-Methods',  value: 'GET, PUT, POST, DELETE');
    res.header( field: 'Access-Control-Allow-Headers',  value: 'Content-Type, Authorization, Content-Length, X-Requested-With');

    //intercepts OPTIONS method
    if ('OPTIONS' === req.method) {
        //respond with 200
        res.sendStatus( code: 200);
    }
    else {
        //move on
        next();
    }
});
// create a write stream (in append mode)
let accessLogStream = fs.createWriteStream(path.join(__dirname, 'access.log'),  options: { flags: 'a' })
```

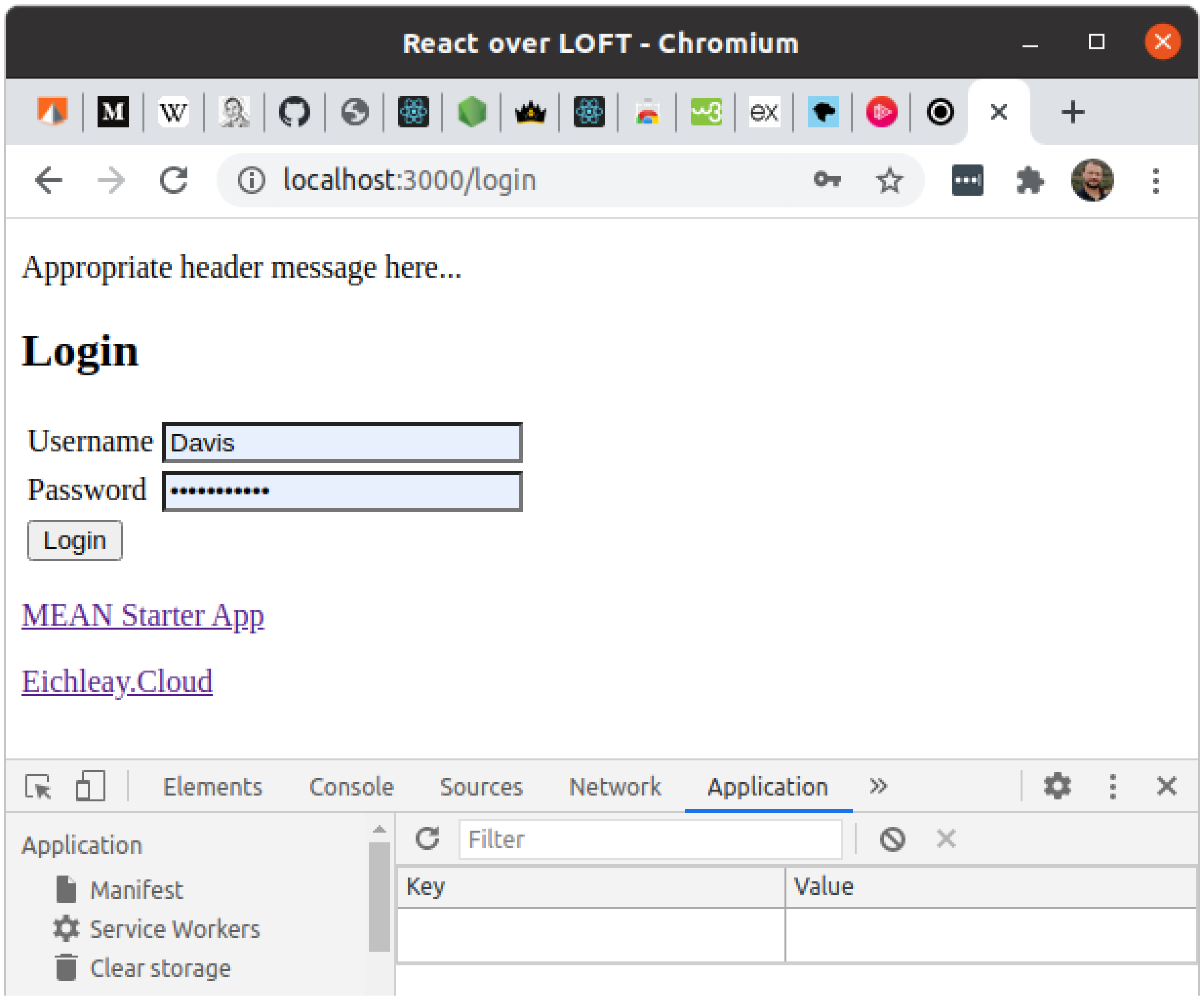
```
app.use(logger( format: 'combined',  options: { stream: accessLogStream }));
```

I fiddled for a while and it turned out to be that using a simple HTML form submission wasn't what the API was expecting. As soon as I got the data into a json document body, it worked just fine. To be fair, I think I could have changed the application type to '`application/x-www-form-urlencoded`' and modified the API to match, but that didn't seem to be an honest division of application layers. After that troubleshooting, I started both the API and the React front end:



```
nullvalues@ubuntu: ~/dev/egt
nullvalues@ubuntu:~/dev/egt$ node bin/www
```

```
nullvalues@ubuntu: ~/dev/roverloft
Compiled successfully!

You can now view roverloft in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://10.10.250.183:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```



Appropriate header message here...

Login

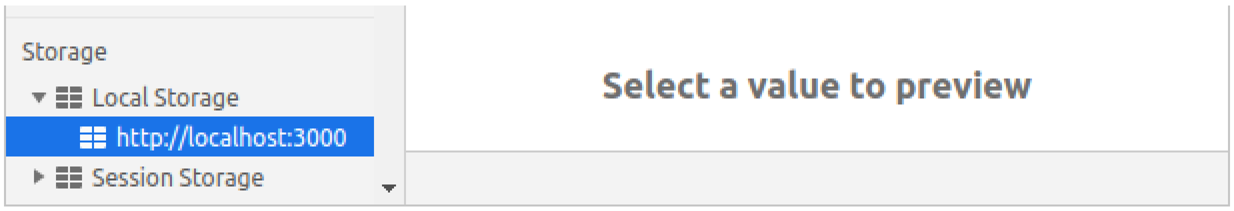
Username	Davis
Password	••••••••••••

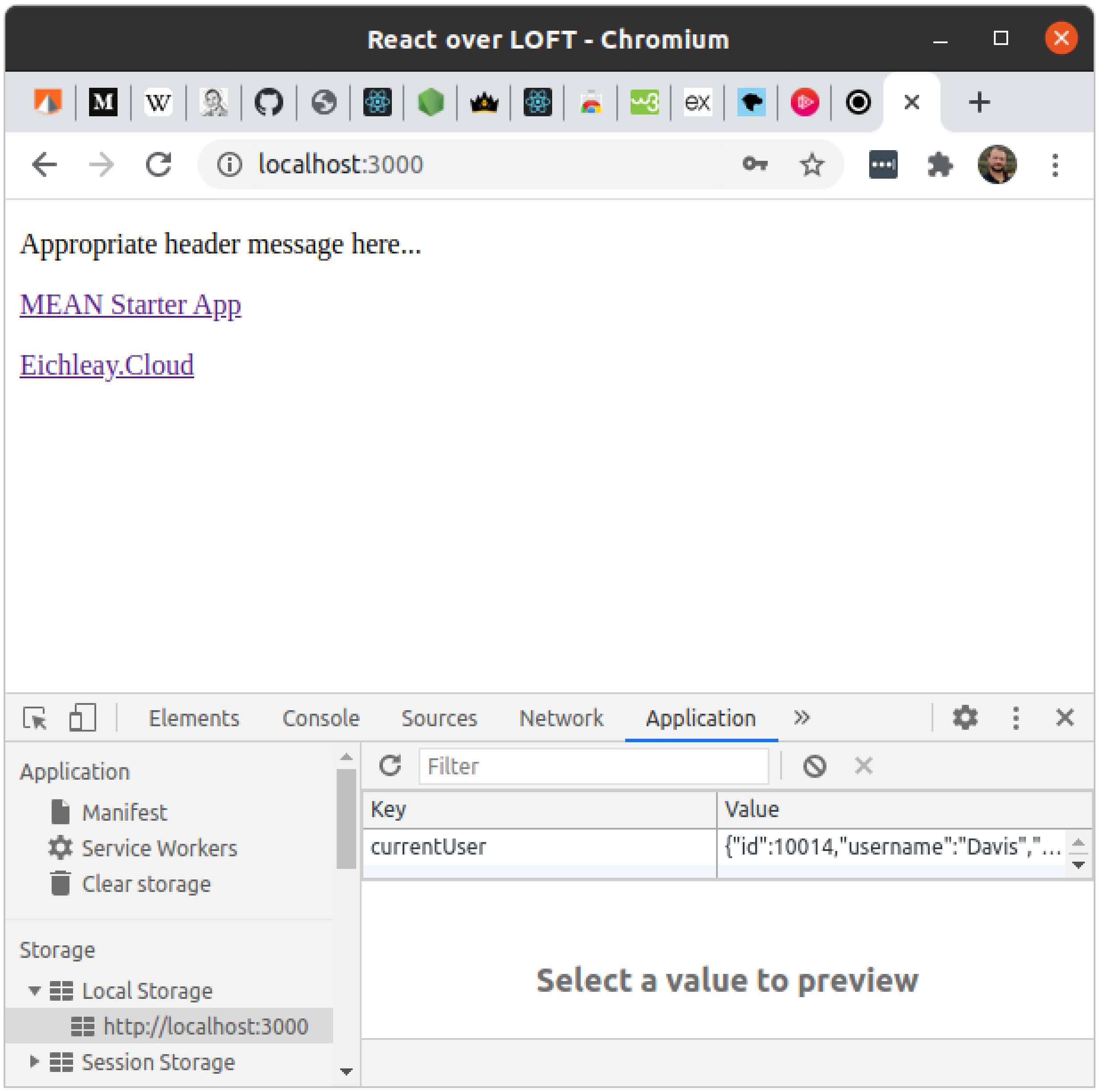
Login

MEAN Starter App

Eichleay.Cloud

Key	Value





Appropriate header message here...

MEAN Starter App

Eichleay.Cloud

Key	Value
currentUser	{"id":10014,"username":"Davis","...

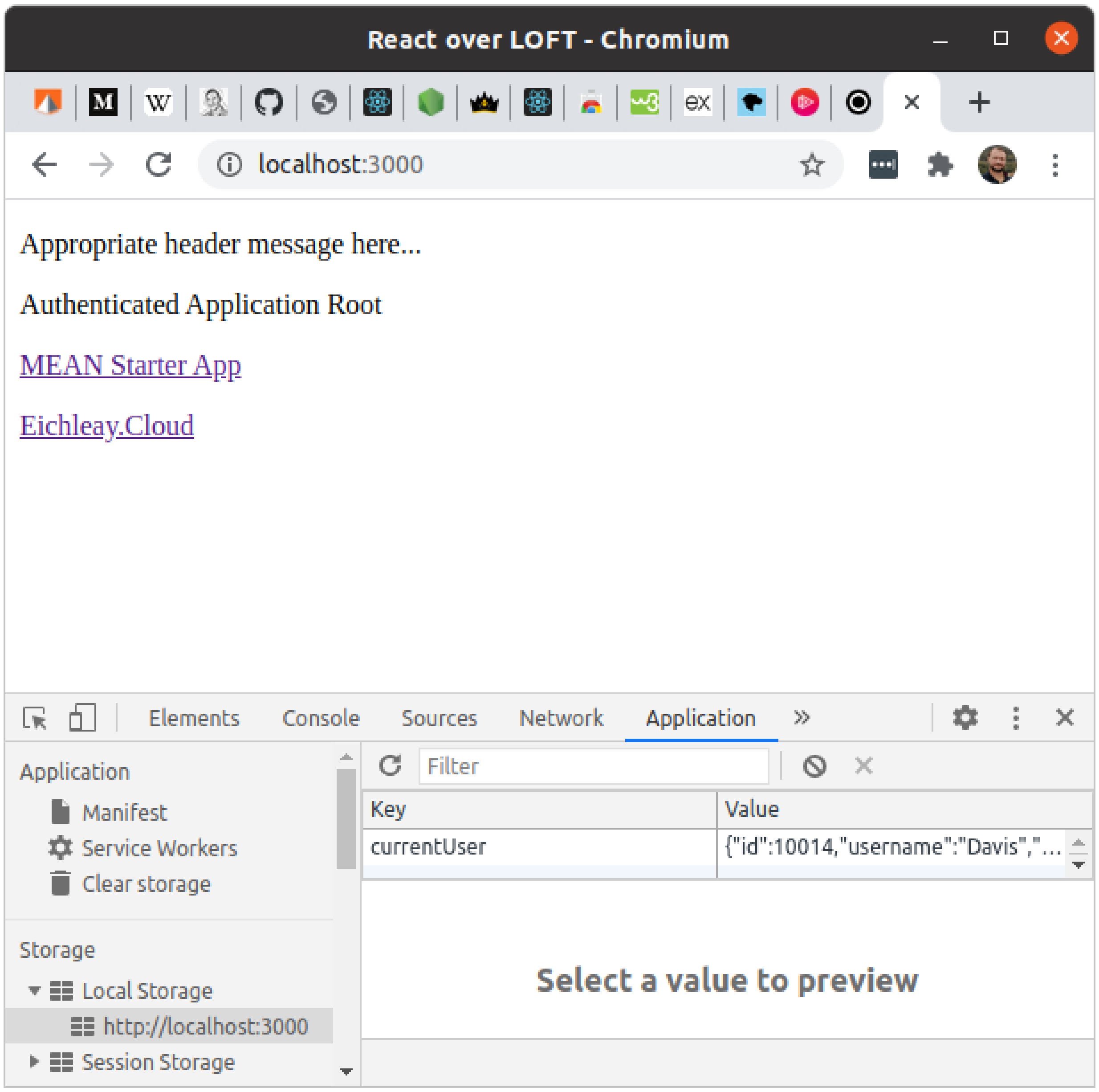
Success! Sorta. Here's the good news:

- Roverloft is able to submit some information to the API.
- The API understands the JSON document provided by React and validated the user to return a JWT, just as we did wth Postman previously.
- React understands that it received a valid token from the API.

Okay, criticisms:

- I had to refresh to the Root route to get our Authenticated Application Root to show up. But that's almost certainly a React internal thing we haven't dug into yet.

So, here's proof of being able to navigate to the Authenticated Application Root and of a valid bearer token which closely resembles what we get directly from the API in Postman.



Appropriate header message here...

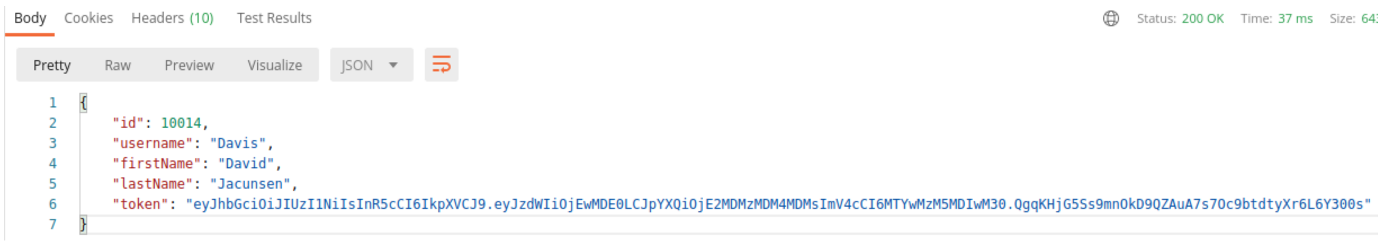
Authenticated Application Root

MEAN Starter App

Eichleay.Cloud

Key	Value
currentUser	{"id":10014,"username":"Davis","...

```
▼{id: 10014, username: "Davis", firstName: "David", lastName: "Jacunsen",…}
   firstName: "David"
   id: 10014
   lastName: "Jacunsen"
   token: "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjEwMDE0LCJpYXQiOjE2MDMzMTM3OTEsImV4cCI6MTYwMzQwMDE5MX0.6B-mDPj79GkUKFnZVPaZQZf5lspSJHFQ_h0MoBlCuvs"
   username: "Davis"
```



```
{
    "id": 10014,
    "username": "Davis",
    "firstName": "David",
    "lastName": "Jacunsen",
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOjEwMDE0LCJpYXQiOjE2MDMzMDM4MDMsImV4cCI6MTYwMzM5MDIwM30.QgqKHjG5Ss9mnOkD9QZAuA7s7Oc9btdtyXr6L6Y300s"
}
```

Okay, it's not doing a lot just yet. Let's fix some stuff and get it to pull a protected route from our API. If we can get that far, we're done for now. For the sake of early simplicity, we de-normalized some things our buddy Jason had normalized which are now going to look idiotic if we copy and paste. Let's fix them.

Authenticated Retrieval from the API

Let’s create a file in _services called mechanics.js. Let’s move the handleResponse function into our mechanics module.

```
export const mechanics = {
    handleResponse
};

function handleResponse(response) {
    return response.text().then(text => {
        const data = text && JSON.parse(text);
        console.log("Data is: ", data);
        if (!response.ok) {
            /* I'm not sure I agree with this approach...let's just 
comment out for now and see if there's
               some truth to the need.  I can't think of a reason that 
an authorization restriction should
               *force* a logout.  But maybe there's more to this we'll 
discover later.
            if ([401, 403].indexOf(response.status) !== -1) {
                // auto logout if 401 Unauthorized or 403 Forbidden 
response returned from api
                authenticationService.logout();
                window.location.reload();
            }
            */

            const error = (data && data.message) || response.statusText;
            return Promise.reject(error);
        }

        return data;
    });
}
```

I notice we’re going to need to provide the token in our header if we’re going to go much further, so let’s copy and paste Mr. Watmore’s authHeader function into our authentication.service module where it seems to fit nicely. I made some small changes to the return value that are commented in the code.

```
import { BehaviorSubject } from 'rxjs';
import { mechanics } from '../_services'

let apiUrl = "http://localhost:3030";

const currentUserSubject = new BehaviorSubject(JSON.parse(localStorage.
getItem('currentUser')));

export const authenticationService = {
    login,
    logout,
    currentUser: currentUserSubject.asObservable(),
    get currentUserValue () { return currentUserSubject.value },
    authHeader,
    authString
};

function login(username, userpass) {

    let myHeaders = new Headers();
    myHeaders.append("Content-Type", "application/json");

    let raw = JSON.stringify({"username":username,"userpass":userpass});

    let requestOptions = {
        method: 'POST',
        headers: myHeaders,
        body: raw,
        redirect: 'follow'
    };

    return fetch(`${apiUrl}/auth/authenticate`, requestOptions)
        .then(mechanics.handleResponse)
        .then(user => {
            // store user details and jwt token in local storage to
keep user logged in between page refreshes
            localStorage.setItem('currentUser', JSON.stringify(user));
            currentUserSubject.next(user);

            return Promise.resolve(user);
        })
        .catch(error => console.log('error', error));
}

function logout() {
    // remove user from local storage to log user out
    localStorage.removeItem('currentUser');
    currentUserSubject.next(null);
}

function authHeader() {
    // return authorization header with jwt token
    const currentUser = currentUserSubject.value;
    if (currentUser && currentUser.token) {
        // note the change here from Jason's Example:
        // return { Authorization: `Bearer ${currentUser.token}` };
        // we're going to return a string that we can append to a
header in a more normalized way
        return `Bearer ${currentUser.token}`;
    } else {
        return '';
    }
}
```

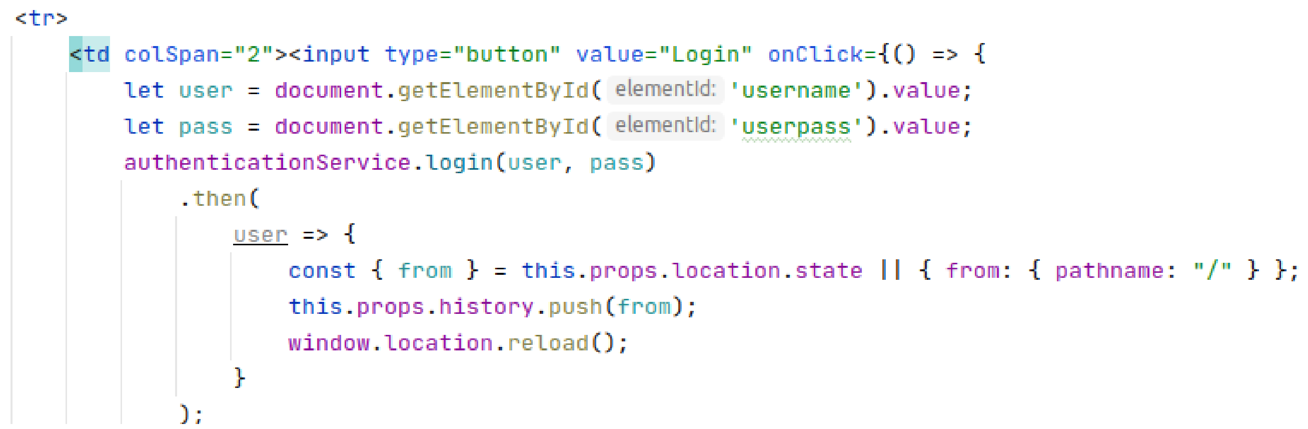
Now let’s create another file in _services called behaviors.js that will start to flesh out the application logic in our React application.

```
import { authenticationService } from '../_services';
import { mechanics } from "../_services";

let apiUrl = "http://localhost:3030";
let exesPath = "/about/exes";

export const xBehavior = {
    getExes
};

function getExes() {
    let myHeaders = new Headers();
    myHeaders.append("Content-Type", "application/json");
    myHeaders.append("Authorization", authenticationService.
authHeader());

    let requestOptions = {
        method: 'GET',
        headers: myHeaders
    };

    return fetch(`${apiUrl}${exesPath}`, requestOptions)
        .then(mechanics.handleResponse);

}
```

And let’s see if adding an evaluation of the current user state fixes our problem with needing to refresh after login to get to the application root. I’m making an educated guess that Jason did that by adding the following to the application file. In our case, the Loft component in Loft.jsx.:

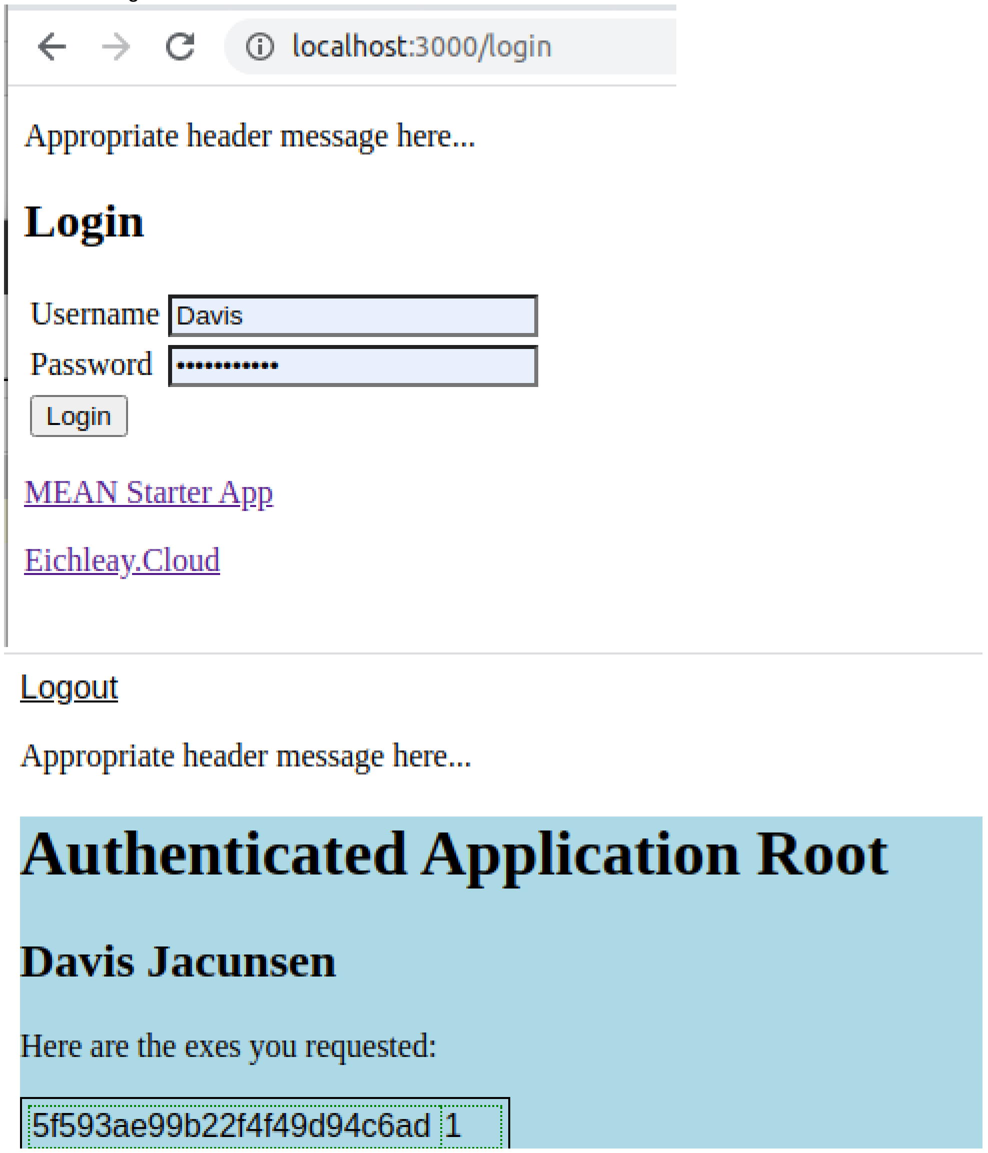
```
    constructor(props) {
        super(props);

        this.state = {
            currentUser: null
        };
    }

    componentDidMount() {
        authenticationService.currentUser.subscribe(x => this.setState
({ currentUser: x }));
    }
```

Let’s add a small modification to our LoginPage.jsx to refresh the page if the login succeeds. Perhaps it’s inefficient, but it should fix our problem of the root module not showing to add a window.location.reload(); to a successful login.



```
<tr>
    <td colSpan="2"><input type="button" value="Login" onClick={() => {
        let user = document.getElementById( elementId: 'username').value;
        let pass = document.getElementById( elementId: 'userpass').value;
        authenticationService.login(user, pass)
            .then(
                user => {
                    const { from } = this.props.location.state || { from: { pathname: "/" } };
                    this.props.history.push(from);
                    window.location.reload();
                }
            );
```

And here we go:



localhost:3000/login

Appropriate header message here...

Login

Username Davis

Password ••••••••••••

Login

MEAN Starter App

Eichleay.Cloud

Logout

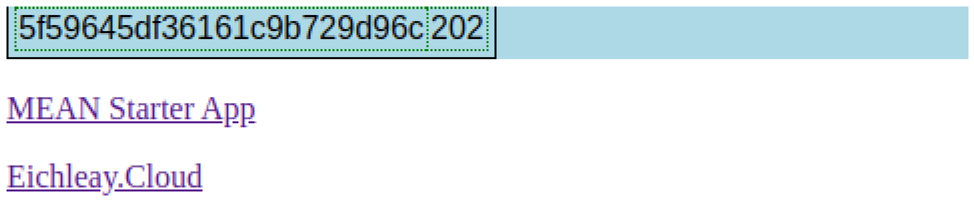
Appropriate header message here...

Authenticated Application Root

Davis Jacunsen

Here are the exes you requested:

5f593ae99b22f4f49d94c6ad	1

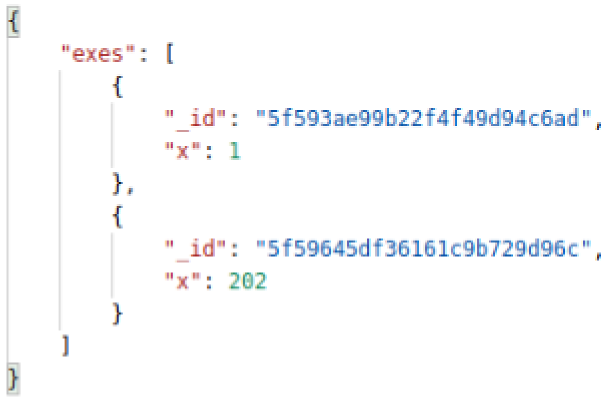


5f59645df36161c9b729d96c	202

MEAN Starter App

Eichleay.Cloud

This should look like a formatted version of our authenticated request from Postman we got earlier:



```
{
    "exes": [
        {
            "_id": "5f593ae99b22f4f49d94c6ad",
            "x": 1
        },
        {
            "_id": "5f59645df36161c9b729d96c",
            "x": 202
        }
    ]
}
```

That’s pretty much it. Everything else is events and UI design on the application side, and business logic and API design on the data side, which is more than we promised for a simple, authenticated application. (We might try to hook this up to an SSO service in the near future that would replace our Node token provider and be backed by an AD or LDAP directory service.)