

# Authenticate a Node.js API with JSON Web Tokens



## [Introduction](#)

Authentication is one of the big parts of every application. Security is always something that is changing and evolving. In the past, we have gone over [Node authentication](#) using the great [Passport](#) npm package.

Those articles used the session based authentication however, which has problems when we talk about scaling web services and creating an API that can be consumed across many devices and services.

As a primer to this article, go ahead and get yourself familiar with [token based authentication principles](#) and the standard used for token based authentication, [JSON Web Tokens](#).

Now that we've got all the important information about token based authentication out of the way, let's build a very simple Node API and use tokens to authenticate users that request access.

## [What We'll Be Building](#)

We'll build a quick API using Node and Express and we'll be using [POSTman](#) to test it.

The main workflow of this is that we will:

1. Have unprotected and protected routes
2. A user will authenticate by passing in a name and a password and get back a token
3. The user will store this token on their client-side and send it for every request
4. We will validate this token, and if all is good, pass back information in JSON format

Our API will be built with:

- normal routes (not authenticated)
- route middleware to authenticate the token
- route to authenticate a user and password and get a token
- authenticated routes to get all users

**Further Reading:** For some information on Express routing and middleware, check out our article on [Express 4.0 routing](#).

We know what we're going to build, now let's get to it!

### [Getting Started](#)

Let's take a look at our file structure for our Node application. This will be simplified and we'll be putting a lot of stuff into the `server.js` file.

## File Structure

```
- app/  
----- models/  
----- user.js  
- config.js  
- package.json  
- server.js
```

For a fully fledged application, you'd want to move some of this out of the main file and into separate files (particularly the routes).

### [Set Up Our Node Application \(package.json\)](#)

First, we need to set up our `package.json` file. This is our beginning file for our Node application.

```
{
  "name": "node-token-jwt",
  "main": "server.js"
}
```

Now that we have our `package.json` set, let's install our packages.

```
$ npm install express body-parser morgan mongoose jsonwebtoken --save
```

- **express** is the popular Node framework
- **mongoose** is how we interact with our MongoDB database
- **morgan** will log requests to the console so we can see what is happening
- **body-parser** will let us get parameters from our POST requests
- **jsonwebtoken** is how we create and verify our JSON Web Tokens

The `--save` modifier will also save these packages to our `package.json` file. How convenient!

Next let's take care of a few files that we'll need for our project.

## User Model (`app/models/user.js`)

The user model that we define will be used when creating and getting users. To create a Mongoose model, let's create the file `app/models/user.js`

```
// get an instance of mongoose and mongoose.Schema
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

// set up a mongoose model and pass it using module.exports
module.exports = mongoose.model('User', new Schema({
  name: String,
  password: String,
```

```
admin: Boolean
}});
```

The other file we'll need to create is our `config.js` file. This is where we can store different variables and configuration for our application.

## Config File (config.js)

For this file, **you will need to create MongoDB database**. You can either create one locally or easily use one online at [modulus.io](https://modulus.io) for free. Either way, you will be able to get a URI string to use as your database configuration.

```
module.exports = {

  'secret': 'ilovescotchscotch',
  'database': 'mongodb://noder:noderauth&54;proximus.modulusmongo.net:27017/so9pojyN'

};
```

- **secret**: used when we create and verify JSON Web Tokens
- **database**: the URI with username and password to your MongoDB installation

With all that out of the way, we can get to the big parts of our tutorial. We still haven't defined our main file (`server.js`), so let's get to that.

### [The Actual Node Application \(server.js\)](#)

In this file, we will:

**Grab All the Packages** This will include the packages we installed earlier (express, body-parser, morgan, mongoose, and jsonwebtoken) and also we'll be grabbing the model and config that we created.

**Configure Our Application** We will set our important variables, configure our packages, and connect to our database here.

**Create Basic Routes** These are the unprotected routes like the home page (`http://localhost:8080`). We'll also create a `/setup` route here so that we can create a sample user in our new database.

**Create API Routes** This includes the following routes:

- `POST http://localhost:8080/api/authenticate` Check name and password against the database and provide a token if authentication successful. This route will not require a token because this is where we get the token.
- `GET http://localhost:8080/api` Show a random message. This route is protected and will require a token.
- `GET http://localhost:8080/api/users` List all users. This route is protected and will require a token.

With those things in our mind, let's start our `server.js` file:

```
// =====  
// get the packages we need =====  
// =====  
var express      = require('express');  
var app          = express();  
var bodyParser   = require('body-parser');  
var morgan       = require('morgan');  
var mongoose     = require('mongoose');  
  
var jwt          = require('jsonwebtoken'); // used to create, sign, and verify tokens  
var config = require('./config'); // get our config file  
var User        = require('./app/models/user'); // get our mongoose model  
  
// =====  
// configuration =====  
// =====  
var port = process.env.PORT || 8080; // used to create, sign, and verify tokens  
mongoose.connect(config.database); // connect to database  
app.set('superSecret', config.secret); // secret variable  
  
// use body parser so we can get info from POST and/or URL parameters  
app.use(bodyParser.urlencoded({ extended: false }));  
app.use(bodyParser.json());  
  
// use morgan to log requests to the console  
app.use(morgan('dev'));
```

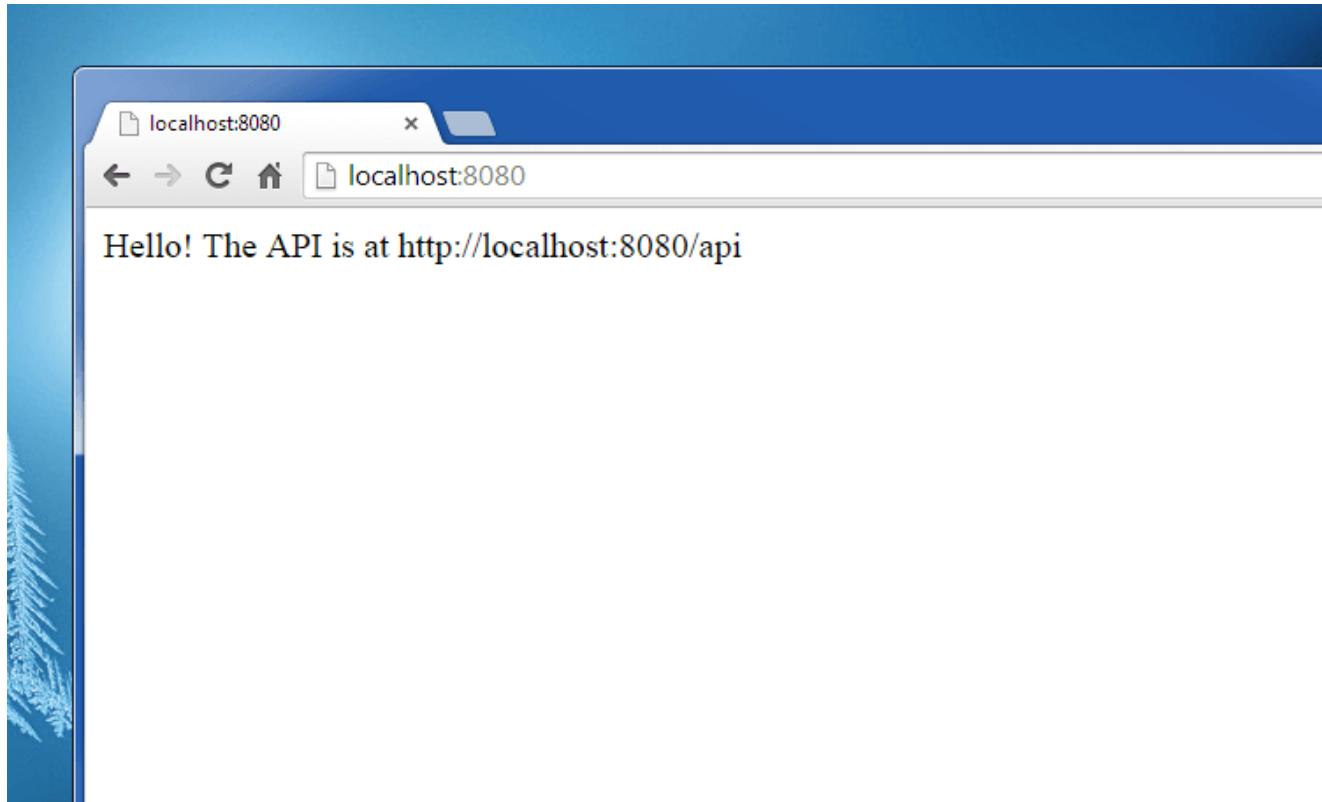
```
// =====  
// routes =====  
// =====  
// basic route  
app.get('/', function(req, res) {  
  res.send('Hello! The API is at http://localhost:' + port + '/api');  
});  
  
// API ROUTES -----  
// we'll get to these in a second  
  
// =====  
// start the server =====  
// =====  
app.listen(port);  
console.log('Magic happens at http://localhost:' + port);
```

With that, we should be able to start up our Node server (make sure you have a valid database configured in `config.js`). Now if we start the server with:

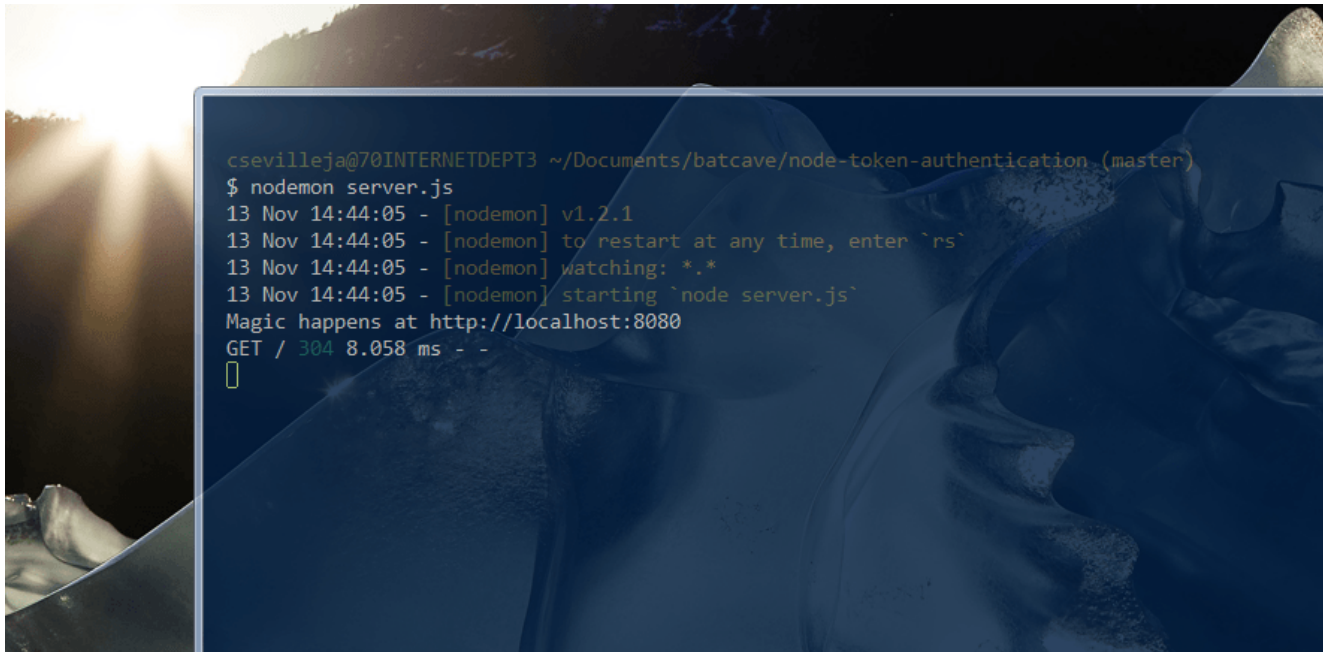
```
$ node server.js
```

**Tip:** Use nodemon to have your server restart on file changes. Install nodemon using `npm install -g nodemon`. Then start your server with `nodemon server.js`.

We should be able to go to our browser and see the message from the route we created. Go to <http://localhost:8080> and you'll see:



As a bonus, since we used morgan, we are able to see the request logged to our console, which helps with development.



```
csevilleja@70INTERNETDEPT3 ~/Documents/batcave/node-token-authentication (master)
$ nodemon server.js
13 Nov 14:44:05 - [nodemon] v1.2.1
13 Nov 14:44:05 - [nodemon] to restart at any time, enter `rs`
13 Nov 14:44:05 - [nodemon] watching: *.*
13 Nov 14:44:05 - [nodemon] starting `node server.js`
Magic happens at http://localhost:8080
GET / 304 8.058 ms - -
[]
```

### [Creating a Sample User](#)

Now we know our application is up and running! Let's create a sample user using the model that we created earlier.

This is a very simple process, we'll just create a quick route that will create a user of our choosing.

Here's that route. Just add it to the routes section of our `server.js`:

```
app.get('/setup', function(req, res) {

  // create a sample user
  var nick = new User({
    name: 'Nick Cerminara',
    password: 'password',
    admin: true
  });

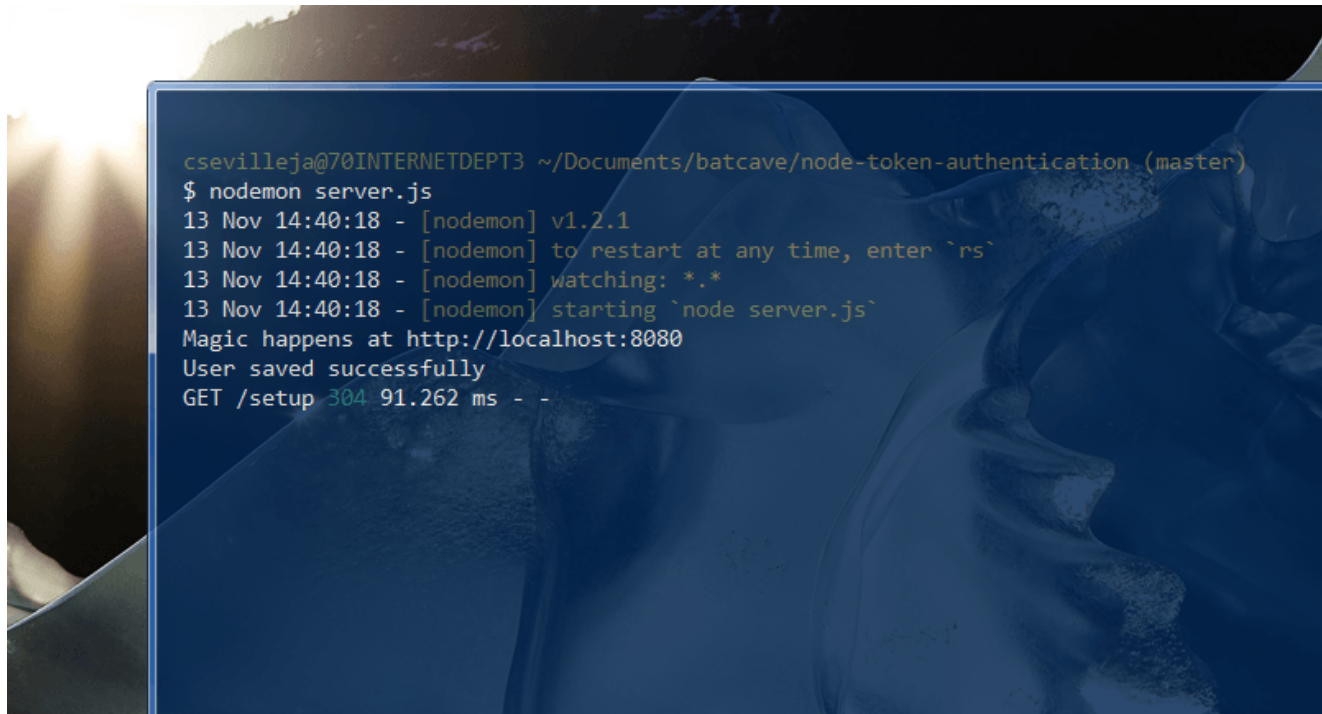
  // save the sample user
  nick.save(function(err) {
```



```
if (err) throw err;

console.log('User saved successfully');
res.json({ success: true });
});
});
```

It's important to note that we would **never** save a password to the database plain text like this. You would protect your passwords by hashing it. Go ahead and visit your application at `http://localhost:8080/setup`. You should see the message 'User saved successfully' logged to your console and the JSON object with `{ success: true }` in your browser.



```
csevilleja@70INTERNETDEPT3 ~/Documents/batcave/node-token-authentication (master)
$ nodemon server.js
13 Nov 14:40:18 - [nodemon] v1.2.1
13 Nov 14:40:18 - [nodemon] to restart at any time, enter `rs`
13 Nov 14:40:18 - [nodemon] watching: *.*
13 Nov 14:40:18 - [nodemon] starting `node server.js`
Magic happens at http://localhost:8080
User saved successfully
GET /setup 304 91.262 ms - -
```

We'll now create a route to get the users in our database and return them as JSON.

### [Showing Our User](#)

Now let's create our API routes and create one to return all our users in JSON format. We'll use an instance of the Express router for this. We'll place a few placeholders so we can see where things will go. Here is the code for that:

```
// API ROUTES -----

// get an instance of the router for api routes
var apiRoutes = express.Router();

// TODO: route to authenticate a user (POST http://localhost:8080/api/authenticate)

// TODO: route middleware to verify a token

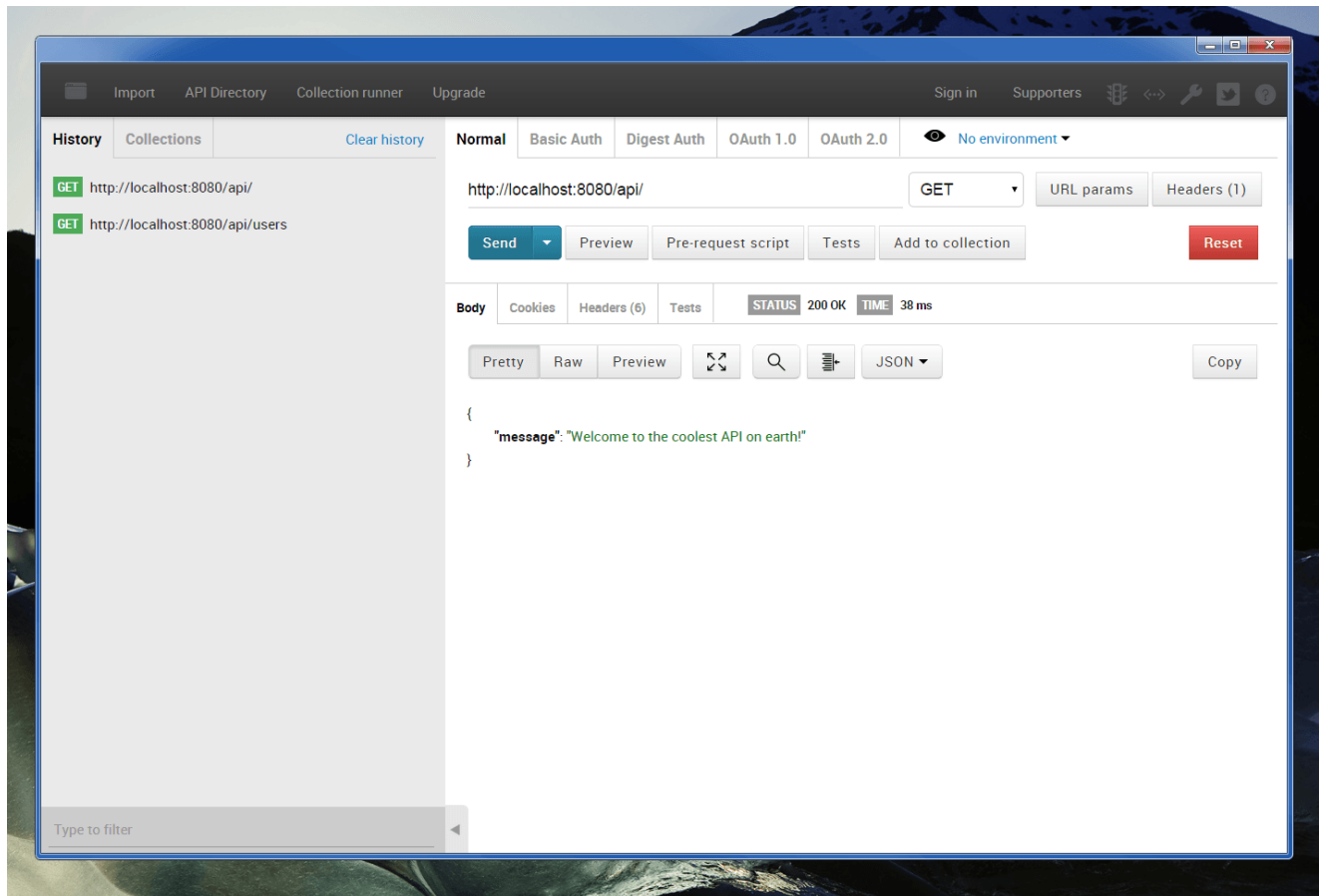
// route to show a random message (GET http://localhost:8080/api/)
apiRoutes.get('/', function(req, res) {
  res.json({ message: 'Welcome to the coolest API on earth!' });
});

// route to return all users (GET http://localhost:8080/api/users)
apiRoutes.get('/users', function(req, res) {
  User.find({}, function(err, users) {
    res.json(users);
  });
});

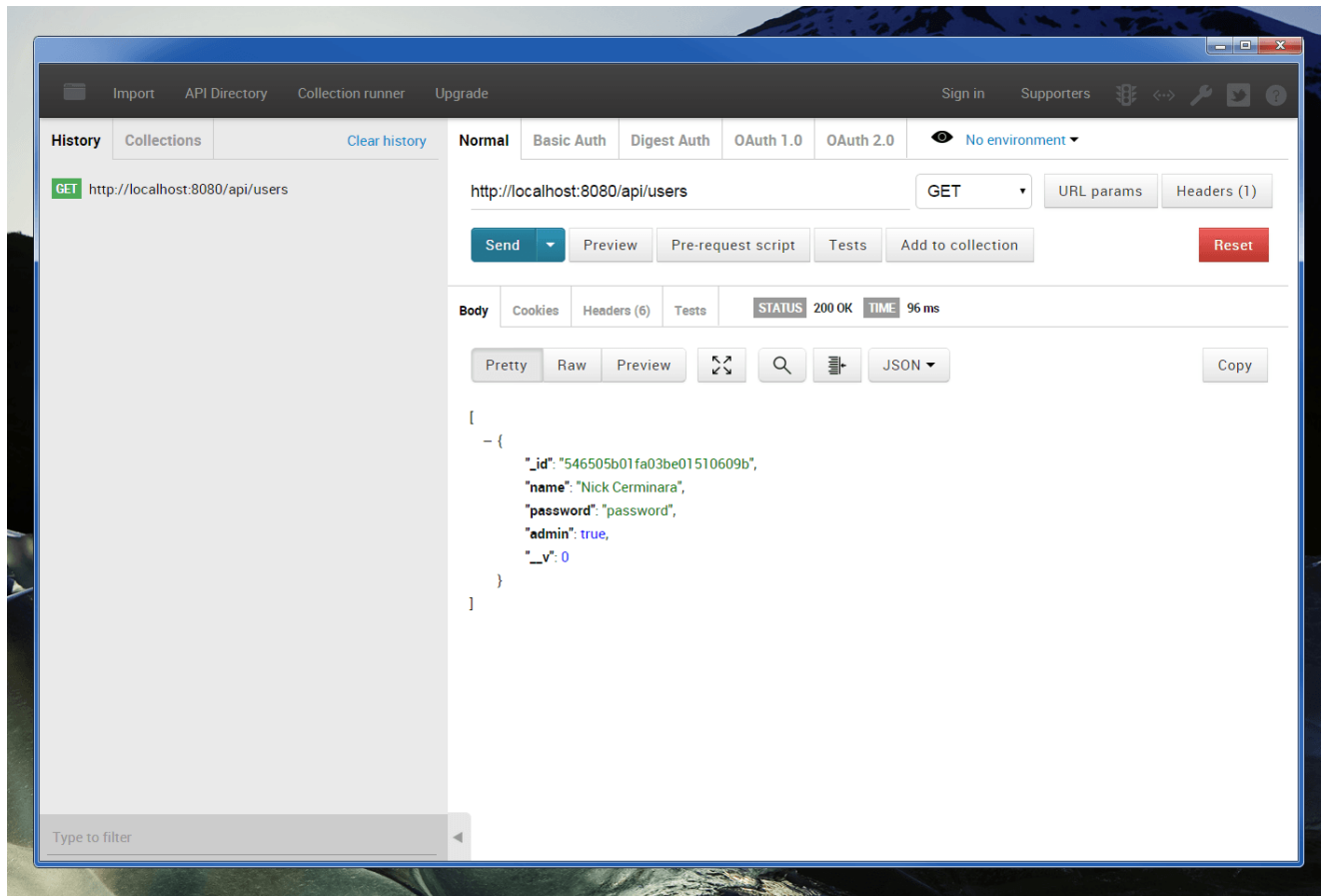
// apply the routes to our application with the prefix /api
app.use('/api', apiRoutes);
```

We now have two routes that we can use. We can see these in our browser again, but at this point, let's switch over to POSTman.

For the URL, use: `http://localhost:8080/api/` and we will be able to see the message.



Then we can go to: `http://localhost:8080/api/users` and see the list of users.



This is great that we have been able to create a user and show them. We probably don't want any random person to see our list of users however.

Next, let's make sure that we can **authenticate a user** and then **protect those routes** using Express route middleware and requiring a token.

### [Authenticating and Creating a Token](#)

Let's make our `POST http://localhost:8080/api/authenticate` route where we will accept a name and a password (probably from a form). If the name and password validate, then we will create a token and pass that back.

Once the user has that token, they will store it client side and pass it with every request for information after that and we will validate the token on every request using route middleware.

Here's the code for our POST route:

```
// API ROUTES -----

// get an instance of the router for api routes
var apiRoutes = express.Router();

// route to authenticate a user (POST http://localhost:8080/api/authenticate)
apiRoutes.post('/authenticate', function(req, res) {

  // find the user
  User.findOne({
    name: req.body.name
  }, function(err, user) {

    if (err) throw err;

    if (!user) {
      res.json({ success: false, message: 'Authentication failed. User not found.' });
    } else if (user) {

      // check if password matches
      if (user.password !== req.body.password) {
        res.json({ success: false, message: 'Authentication failed. Wrong password.' });
      } else {

        // if user is found and password is right
        // create a token
        var token = jwt.sign(user, app.get('superSecret'), {
          expiresInMinutes: 1440 // expires in 24 hours
        });

        // return the information including token as JSON
        res.json({
          success: true,
```

```
        message: 'Enjoy your token!',
        token: token
    });
}

}

});
});

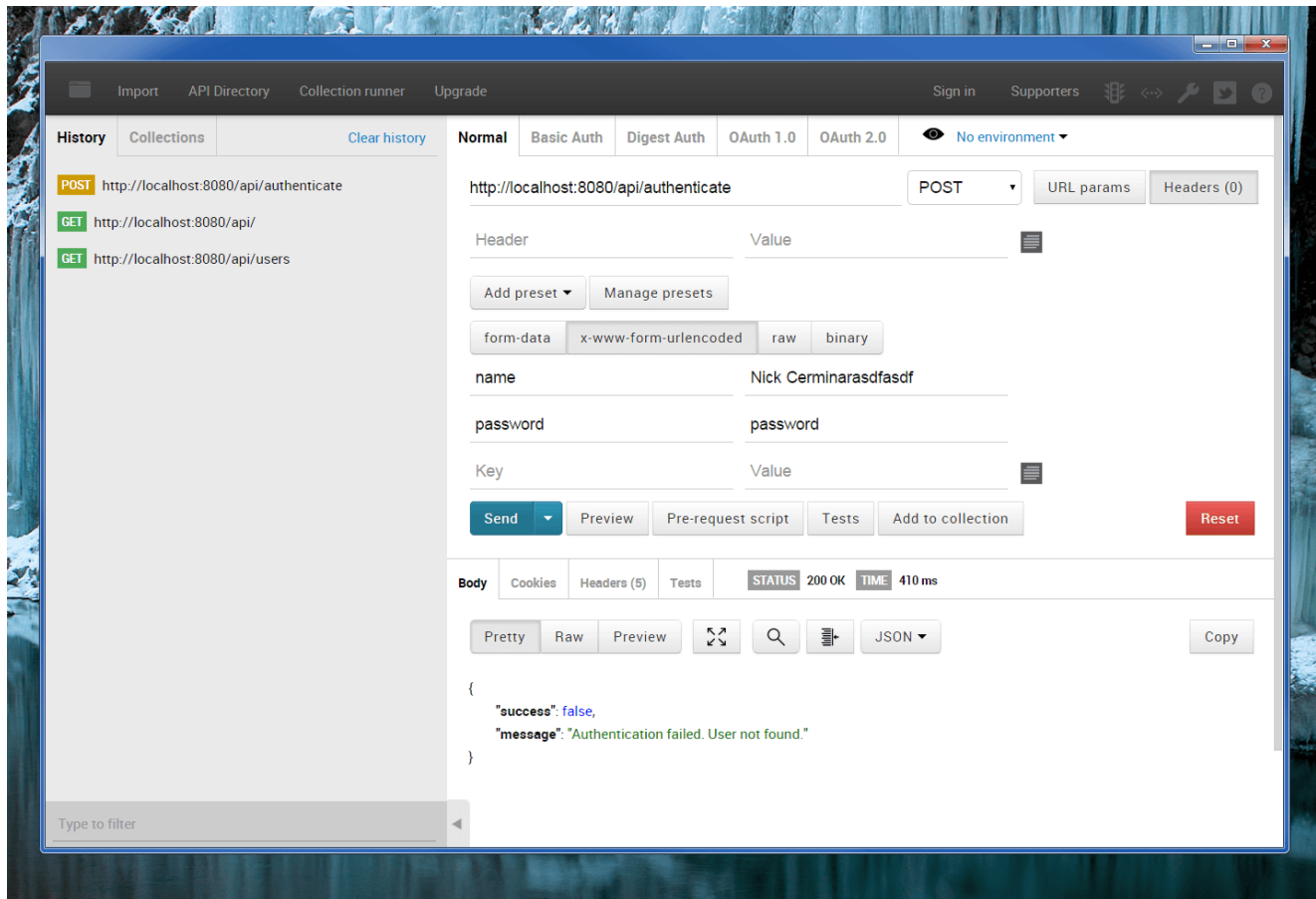
...
```

With this code, we can check our user and password and pass back a token in a JSON response. We are using **mongoose to find the user** and **jsonwebtoken to create the token**.

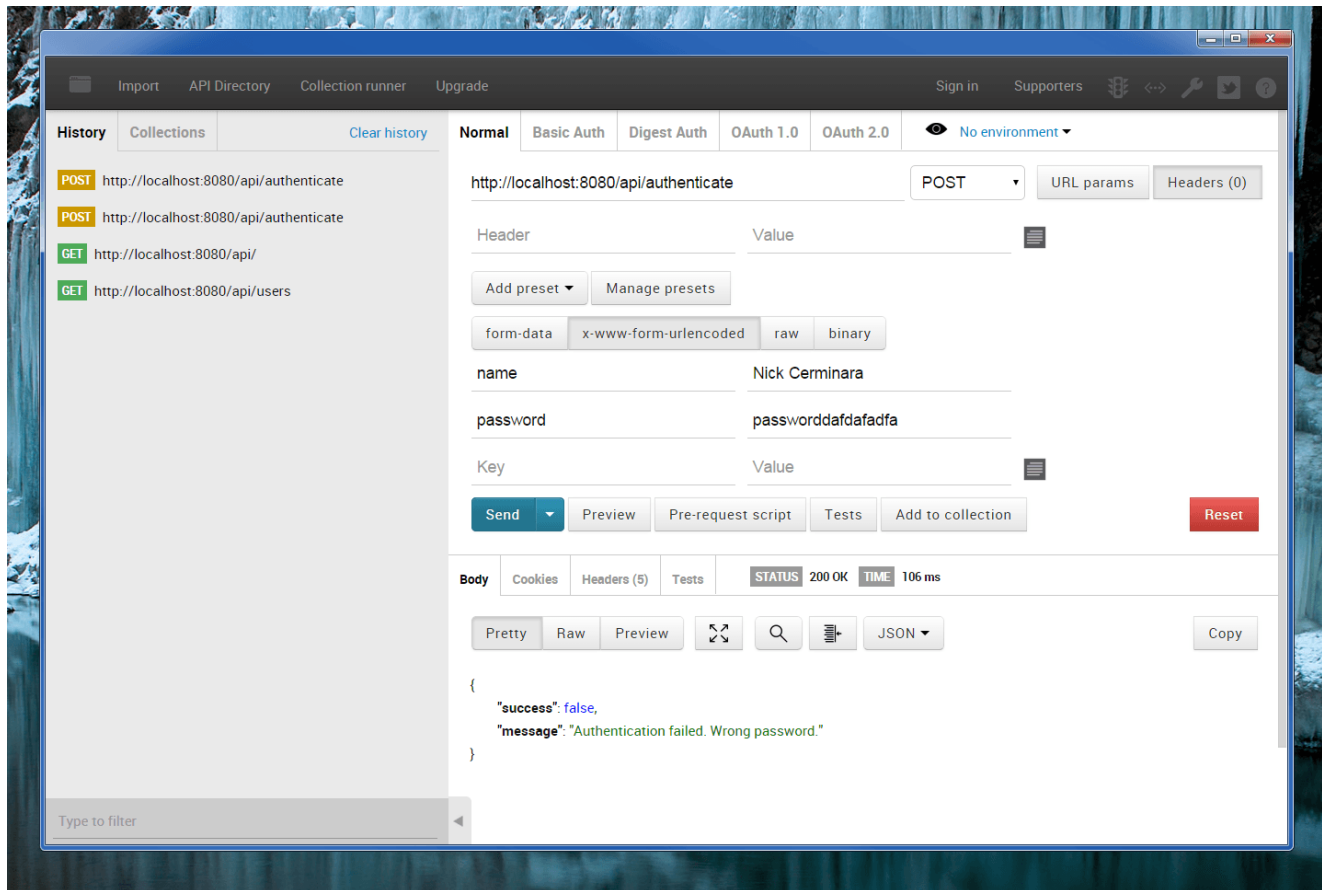
Let's test this out using POSTman. Change your HTTP request to POST and add the name and password parameters to `x-www-form-urlencoded`. This is how we simulate information coming through a form POST.

Remember the user we created earlier had a name of **Nick Cerminara** and a password of **password**. (super safe, I know)

Here's the route using the wrong name:

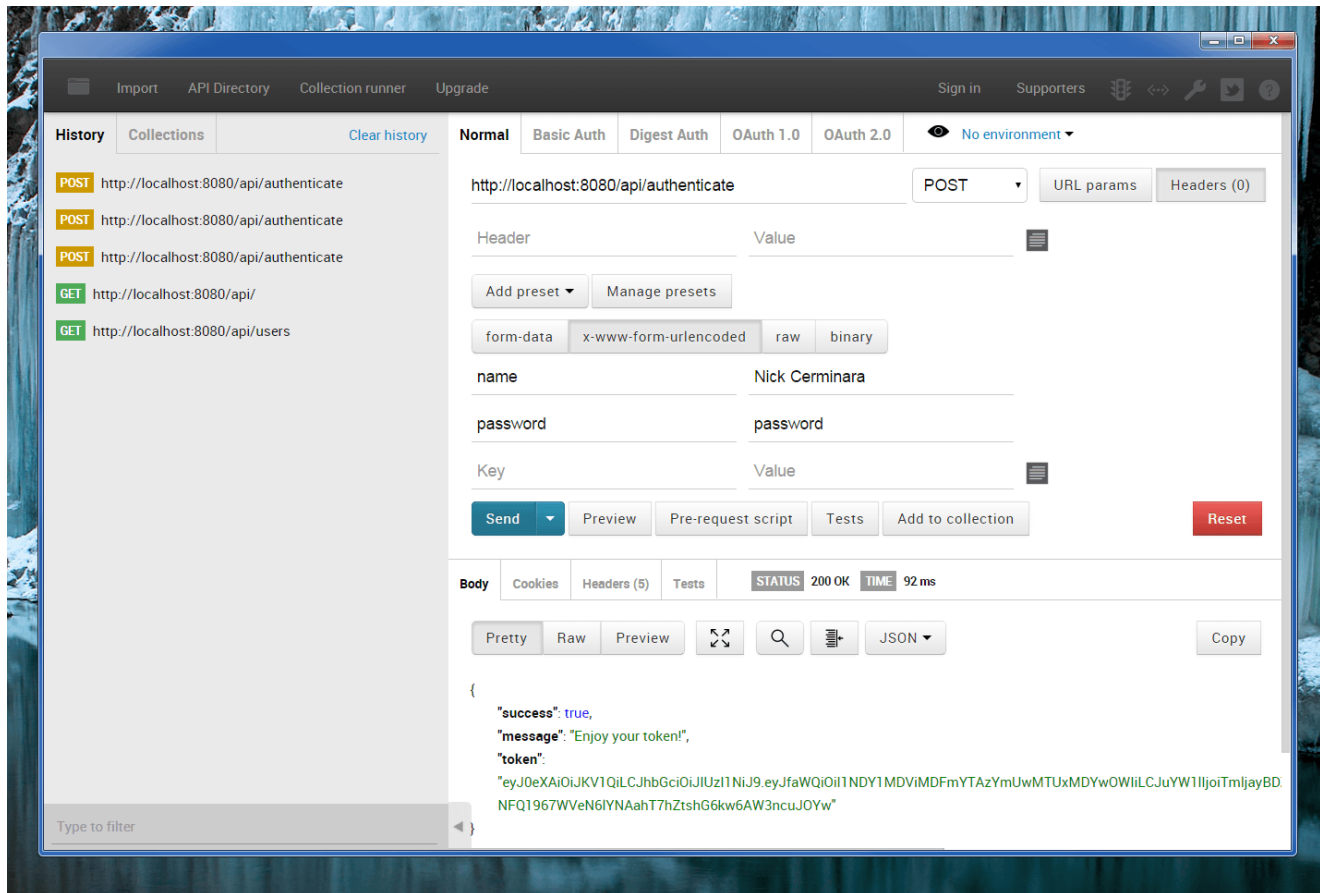


Using the right name and wrong password:



When everything goes well:





You can see that in our response, we are given our token! Now let's copy and paste that token somewhere safe until we are able to use it after we create our route middleware next.

### [Route Middleware to Protect API Routes](#)

At this point, we have 3 routes defined on our API routes (`/api/authenticate`, `/api`, and `/api/users`). Let's create route middleware to protect the last 2 routes. We won't want to protect the `/api/authenticate` route so what we'll do is place our middleware beneath that route. Order is important here.

Let's take a look at the code:

```
// API ROUTES -----

// get an instance of the router for api routes
```

```
var apiRoutes = express.Router();

// route to authenticate a user (POST http://localhost:8080/api/authenticate)
...

// route middleware to verify a token
apiRoutes.use(function(req, res, next) {

  // check header or url parameters or post parameters for token
  var token = req.body.token || req.query.token || req.headers['x-access-token'];

  // decode token
  if (token) {

    // verifies secret and checks exp
    jwt.verify(token, app.get('superSecret'), function(err, decoded) {
      if (err) {
        return res.json({ success: false, message: 'Failed to authenticate token.' });
      } else {
        // if everything is good, save to request for use in other routes
        req.decoded = decoded;
        next();
      }
    });
  } else {

    // if there is no token
    // return an error
    return res.status(403).send({
      success: false,
      message: 'No token provided.'
    });
  }
});
```

```
    }  
  });  
  
  // route to show a random message (GET http://localhost:8080/api/  
  ...  
  
  // route to return all users (GET http://localhost:8080/api/users)  
  ...  
  
  // apply the routes to our application with the prefix /api  
  app.use('/api', apiRoutes);
```

We are using the `jsonwebtoken` package again, but this time we are going to verify the token that was passed in. It is important that our secret used here matches the secret that was used to create the token. We are also making sure to send the right HTTP response code as 403 forbidden and our user was not authenticated to view any data.

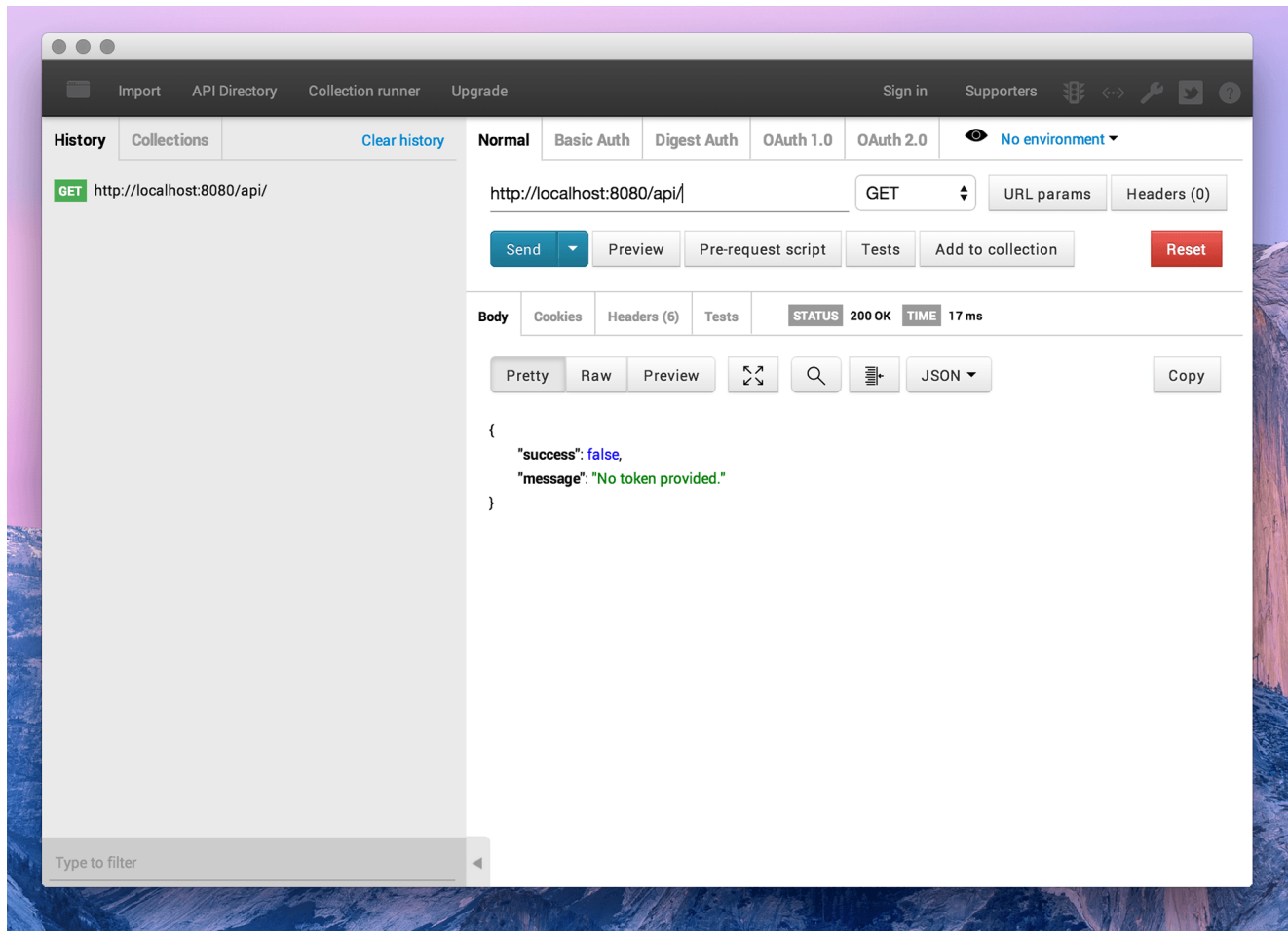
If everything looks good the token was able to be verified, we'll take the information that came out of the token and pass it to the other routes in the `req` object.

### [Testing Our Middleware](#)

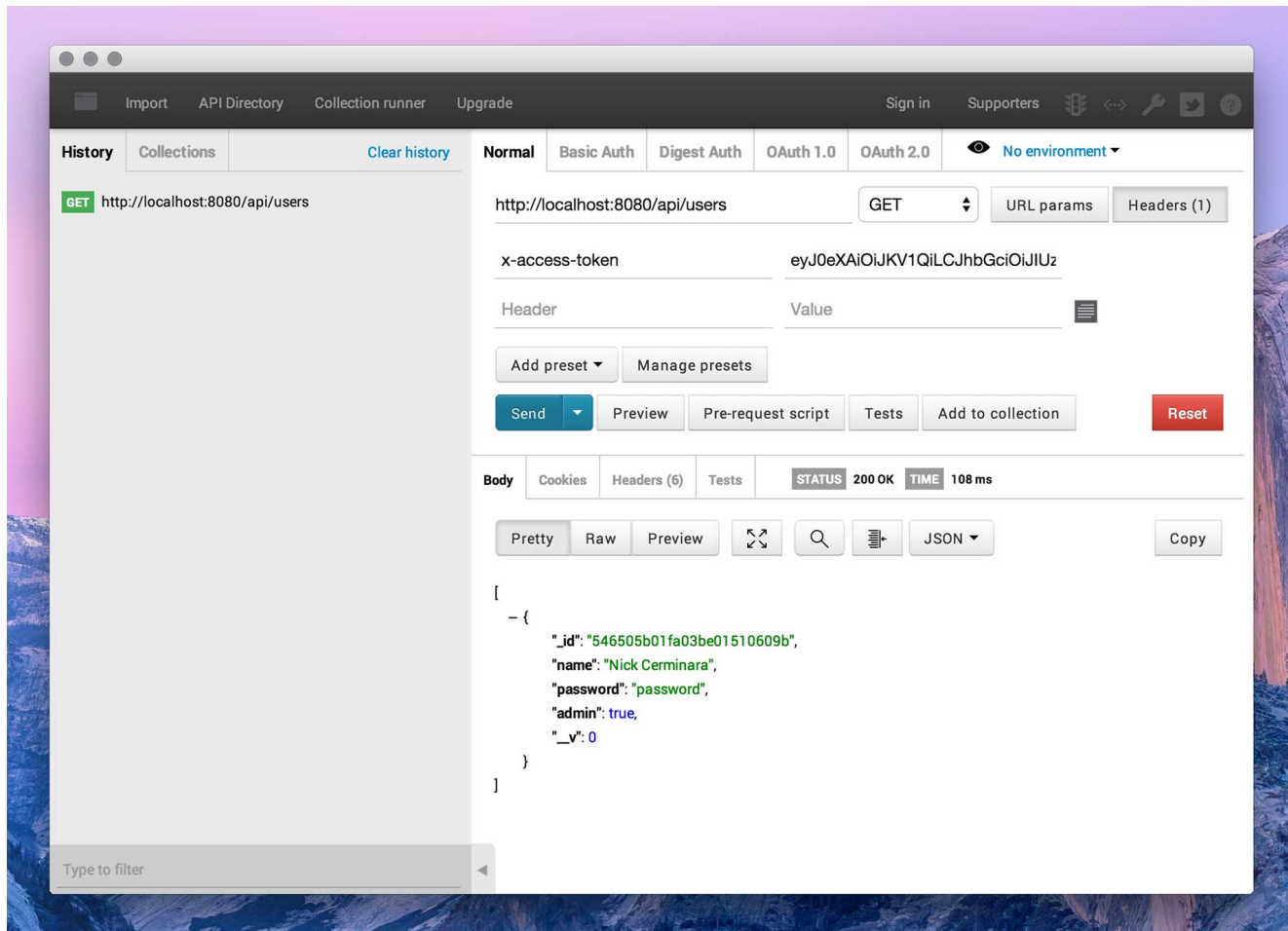
We have built our middleware that our Node application will run through before it gets to the routes that we want authenticated.

Let's go into POSTman again and try to access both routes without passing a token.

This is our route without the token just accessing the base api route of `http://localhost:8080/api:`

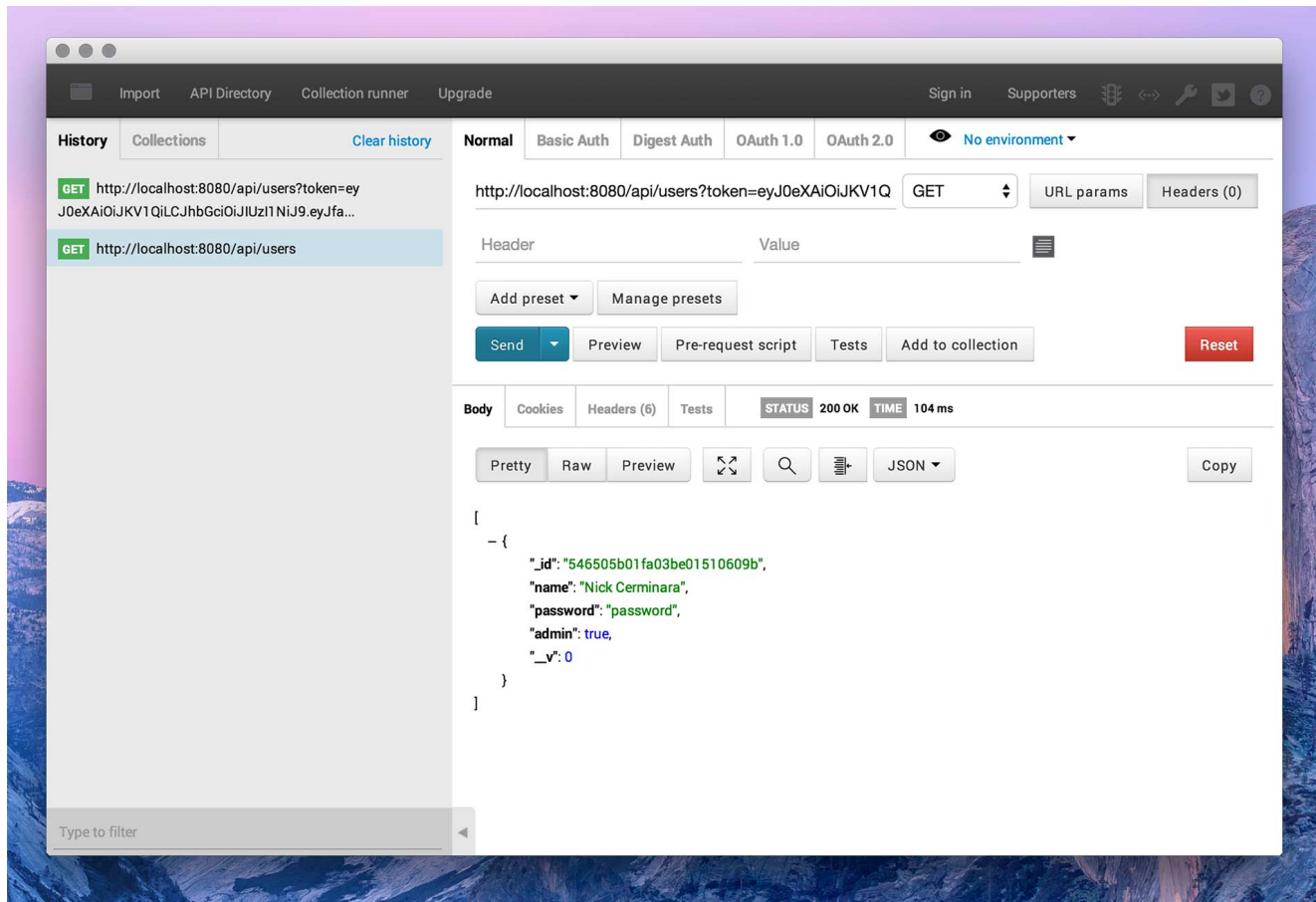


Now let's pass in the token that was created before in our HTTP header as `x-access-token` accessing the users list at `http://localhost:8080/api/users`:



We can also pass it in as a URL parameter by going to: `http://localhost:8080/api/users?`

`token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJfaWQiOiJ1NDY1MDViMDFmYTZmYUwMTUxMDYwOWIiLCJuYW11IjoiTmljayBDZXJtaW5hcmEiLCJwYXNzd29yZCI6InBhc3N3b3JkIiwiaWVWRtaW4iOiOnRydWUsIl9fdil6MH0.ah-NFQ1967WVeN6IYNAahT7hZtshG6kw6AW3ncuJOYw`



## Conclusion

This is a good look at how we can protect routes and our Node API using JSON Web Tokens. This can be expanded into a much larger scoped project like providing permission specific tokens and creating a more robust and feature filled API.

Hopefully this look has given you a good understanding of how the routes are used (especially middleware), tokens are created, and requests to the API all come together.