SEPTEMBER 4, 2017 / #NODEJS

Securing Node.js RESTful APIs with JSON Web Tokens

Adnan Rahić

Have you ever wondered how authentication works? What's behind all the complexity and abstractions. Actually, nothing special. It's a way of encrypting a value, in turn creating a unique token that users use as an identifier. This token verifies your identity. It can authenticate who you are, and authorize various resources you have access to. If you by any chance don't know any of these keywords, be patient, I'll explain everything below.

This will be a step by step tutorial of how to add token based authentication to an existing REST API. The authentication strategy in question is JWT (JSON Web Token). If that doesn't tell you much, it's fine. It was just as strange for me when I first heard the term.

What does JWT actually mean in a down to earth point of view? Let's break down what the official definition states:

> JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted.
> - **Internet Engineering Task Force (IETF)**

That was a mouthful. Let's translate that to English. A JWT is an encoded string of characters which is safe to send between two computers if they both have HTTPS. The token represents a value that is accessible only by the computer that has access to the secret key with which it was encrypted. Simple enough, right?

What does this look like in real life? Let's say a user wants to sign in to their account. They send a request with the required credentials such as email and password to the server. The server checks to see if the credentials are valid. If they are, the server creates a token using the desired payload and a secret key. This string of characters that results from the encryption is called a token.

Then the server sends it back to the client. The client, in turn, saves the token to use it in every other request the user will send. The practice of adding a token to the request headers is as way of authorizing the user to access resources. This is a practical example of how JWT works.

Okay, that's enough talk! The rest of this tutorial will be coding, and I'd love if you would follow along and code alongside me, as we progress. Every snippet of code will be followed by an explanation. I believe the best way of understanding it correctly will be to code it yourself along the way.

Before I begin, there are some things you need to know about Node.js and some EcmaScript standards I'll be using. I will not be using ES6, as it is not as beginner friendly as traditional JavaScript. But, I will expect you already know how to build a RESTful API with Node.js. If not, you can take a detour and check this out before proceeding.

Also, the whole demo is on GitHub if you wish to see it in its entirety.

Let's start writing some code, shall we?

Well, not yet actually. We need to set up the environment first. The code will have to wait at least a couple more minutes. This part is boring so to get up and running quick we'll clone the repository from the tutorial above. Open up a terminal window or command line prompt and run this command:

```
git clone https://github.com/adnanrahic/nodejs-restful-api.git
```

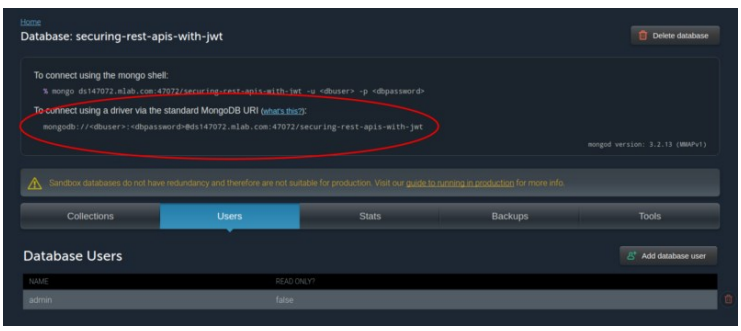
You'll see a folder appear, open it up. Let's take a look at the folder structure.

```
> user
  - User.js
  - UserController.js
- db.js
- server.js
- app.js
- package.json
```

We have a user folder with a model and a controller, and basic CRUD already implemented. Our **app.js** contains the basic configuration. The **db.js** makes sure the application connects to the database. The **server.js** makes sure our server spins up.

Go ahead and install all required Node modules. Switch back to your terminal window. Make sure you're in the folder named '**nodejs-restful-api**' and run `npm install`. Wait a second or two for the modules to install. Now you need to add a database connection string in **db.js**.

Jump over to mLab, create an account if you do not already have one, and open up your database dashboard. Create a new database, name it as you wish and proceed to its configuration page. Add a database user to your database and copy the connection string from the dashboard to your code.



All you need to do now is to change the placeholder values for `<dbuser>` and

`<dbpassword>`. Replace them with the username and password of the user you created for the database. A detailed step by step explanation of this process can be found in the tutorial linked above.

Let's say the user I created for the database is named `wally` with a password of `theflashisawesome`. Having that in mind, the **db.js** file should now look something like this:

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://wally:theflashisawesome@ds147072.mlab.com:47072/securing-rest-
```

Go ahead and spin up the server, back in your terminal window type `node server.js`. You should see `Express server listening on port 3000` get logged to the terminal.

Finally, some code.

Let's start out by brainstorming about what we want to build. First of all we want to add user authentication. Meaning, implementing a system for registering and logging users in.

Secondly, we want to add authorization. The act of granting users the permission to access certain resources on our REST API.

Start out by adding a new file in the root directory of the project. Give it a name of **config.js**. Here you'll put configuration settings for the application. Everything we need at the moment is just to define a secret key for our JSON Web Token.

Disclaimer: Have in mind, under no circumstances should you ever, (EVER!) have your secret key publicly visible like this. Always put all of your keys in environment variables! I'm only writing it like this for demo purposes.

```
// config.js
module.exports = {
  'secret': 'supersecret'
};
```

With this added you're ready to start adding the authentication logic. Create a folder named **auth** and start out by adding a file named **AuthController.js**. This controller will be home for our authentication logic.

Add this piece of code to the top of the **AuthController.js**.

```
// AuthController.js

var express = require('express');
var router = express.Router();
var bodyParser = require('body-parser');
router.use(bodyParser.urlencoded({ extended: false }));
router.use(bodyParser.json());
var User = require('../user/User');
```

Now you're ready to add the modules for using JSON Web Tokens and encrypting passwords. Paste this code into the **AuthController.js**:

```
var jwt = require('jsonwebtoken');
var bcrypt = require('bcryptjs');
var config = require('../config');
```

Open up a terminal window in your project folder and install the following modules:

```
npm install jsonwebtoken --save
npm install bcryptjs --save
```

That's all the modules we need to implement our desired authentication. Now you're ready to create a `/register` endpoint. Add this piece of code to your **AuthController.js**:

```
router.post('/register', function(req, res) {

  var hashedPassword = bcrypt.hashSync(req.body.password, 8);

  User.create({
    name : req.body.name,
    email : req.body.email,
    password : hashedPassword
  },
  function (err, user) {
    if (err) return res.status(500).send("There was a problem registering the user.")
    // create a token
    var token = jwt.sign({ id: user._id }, config.secret, {
      expiresIn: 86400 // expires in 24 hours
    });
    res.status(200).send({ auth: true, token: token });
  });
});
```

Here we're expecting the user to send us three values, a name, an email and a password. We're immediately going to take the password and encrypt it with Bcrypt's hashing method. Then take the hashed password, include name and email and create a new user. After the user has been successfully created, we're at ease to create a token for that user.

The `jwt.sign()` method takes a payload and the secret key defined in **config.js** as parameters. It creates a unique string of characters representing the payload. In our case, the payload is an object containing only the id of the user. Let's write a piece of code to get the user id based on the token we got back from the register endpoint.

```
router.get('/me', function(req, res) {
  var token = req.headers['x-access-token'];
  if (!token) return res.status(401).send({ auth: false, message: 'No token provided.' });

  jwt.verify(token, config.secret, function(err, decoded) {
    if (err) return res.status(500).send({ auth: false, message: 'Failed to authenticate t

    res.status(200).send(decoded);
  });
});
```

Here we're expecting the token be sent along with the request in the headers. The default name for a token in the headers of an HTTP request is `x-access-token`. If there is no token provided with the request the server sends back an error. To be more precise, an `401 unauthorized` status with a response message of '**No token provided**'. If the token exists, the `jwt.verify()` method will be called. This method decodes the token making it possible to view the original payload. We'll handle errors if there are any and if there are not, send back the decoded value as the response.

Finally we need to add the route to the **AuthController.js** in our main **app.js** file. First export the router from **AuthController.js**:

```
// add this to the bottom of AuthController.js
module.exports = router;
```

Then add a reference to the controller in the main app, right above where you exported the app.

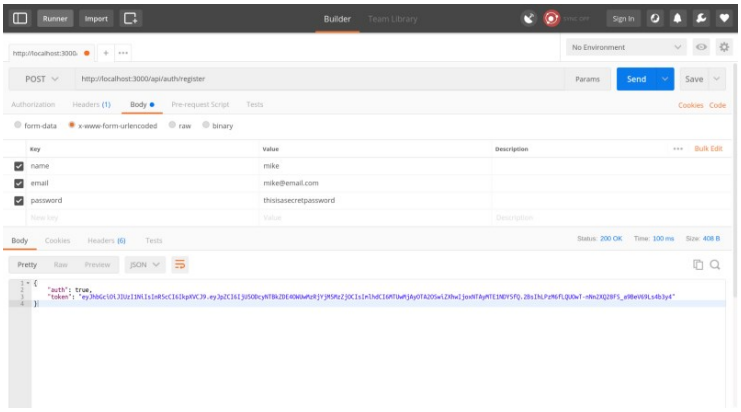
```
// app.js
var AuthController = require('./auth/AuthController');
app.use('/api/auth', AuthController);
module.exports = app;
```

Let's test this out. Why not?

Open up your REST API testing tool of choice, I use Postman or Insomnia, but any will do.

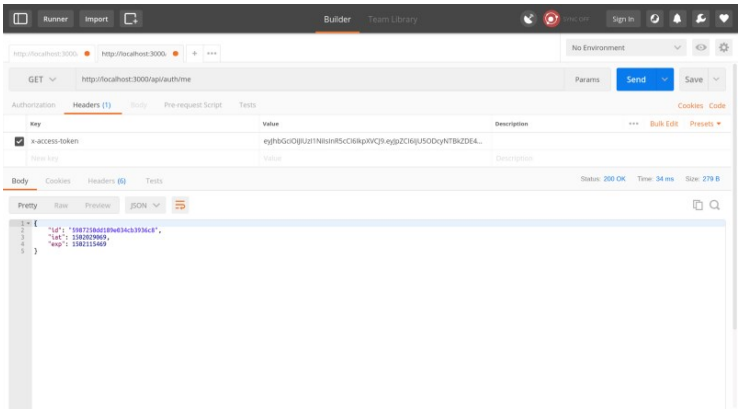
Go back to your terminal and run `node server.js`. If it is running, stop it, save all changes to you files, and run `node server.js` again.

Open up Postman and hit the register endpoint (`/api/auth/register`). Make sure to pick the POST method and `x-www-form-url-encoded`. Now, add some values. My user's name is Mike and his password is 'thisisasecretpassword'. That's not the best password I've ever seen, to be honest, but it'll do. Hit send!



/register

See the response? The token is a long jumbled string. To try out the `/api/auth/me` endpoint, first copy the token. Change the URL to `/me` instead of `/register`, and the method to GET. Now you can add the token to the request header.

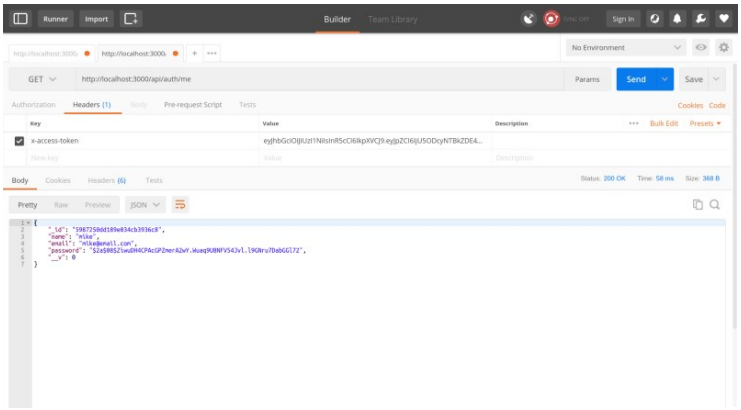


/me

Voilà! The token has been decoded into an object with an id field. Want to make sure that the id really belongs to Mike, the user we just created? Sure you do. Jump back into your code editor.

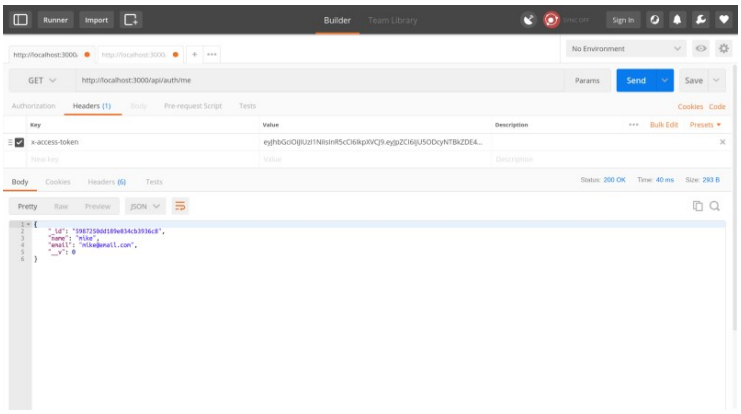
```
// in AuthController.js change this line
res.status(200).send(decoded);

// to
User.findById(decoded.id, function (err, user) {
  if (err) return res.status(500).send("There was a problem finding the user.");
  if (!user) return res.status(404).send("No user found.");

  res.status(200).send(user);
});
```

Now when you send a request to the `/me` endpoint you'll see:



The response now contains the whole user object! Cool! But, not good. The password should never be returned with the other data about the user. Let's fix this. We can add a projection to the query and omit the password. Like this:

```
User.findById(decoded.id,
  { password: 0 }, // projection
  function (err, user) {
    if (err) return res.status(500).send("There was a problem finding the user.");
    if (!user) return res.status(404).send("No user found.");

    res.status(200).send(user);
});
```



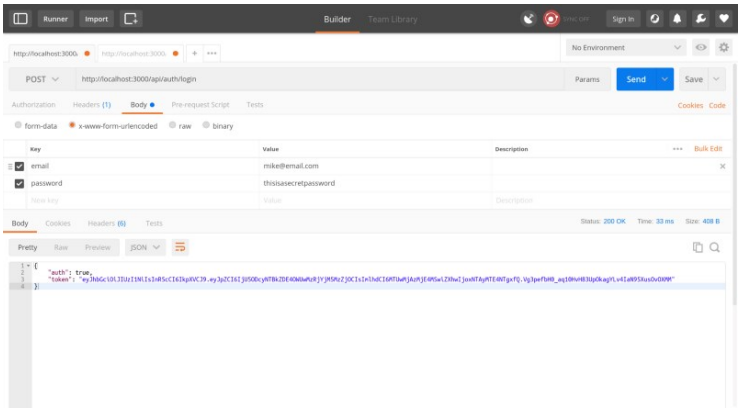
That's better, now we can see all values except the password. Mike's looking good.

Did someone say login?

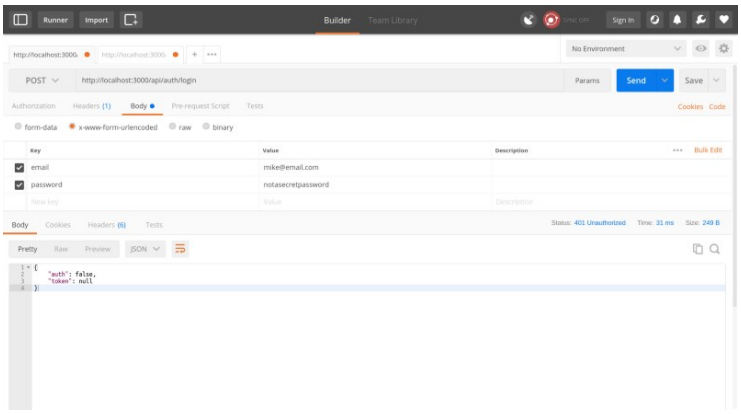
After implementing the registration, we should create a way for existing users to log in. Let's think about it for a second. The register endpoint required us to create a user, hash a password, and issue a token. What will the login endpoint need us to implement? It should check if a user with the given email exists at all. But also check if the provided password matches the hashed password in the database. Only then will we want to issue a token. Add this to your **AuthController.js**.

```
router.post('/login', function(req, res) {

  User.findOne({ email: req.body.email }, function (err, user) {
    if (err) return res.status(500).send('Error on the server.');
    if (!user) return res.status(404).send('No user found.');

    var passwordIsValid = bcrypt.compareSync(req.body.password, user.password);
    if (!passwordIsValid) return res.status(401).send({ auth: false, token: null });

    var token = jwt.sign({ id: user._id }, config.secret, {
      expiresIn: 86400 // expires in 24 hours
    });

    res.status(200).send({ auth: true, token: token });
  });

});
```

First of all we check if the user exists. Then using Bcrypt's `.compareSync()` method we compare the password sent with the request to the password in the database. If they match we `.sign()` a token. That's pretty much it. Let's try it out.



Cool it works! What if we get the password wrong?



Great, when the password is wrong the server sends a response status of `401 unauthorized`. Just what we wanted!

To finish off this part of the tutorial, let's add a simple logout endpoint to nullify the token.

```
// AuthController.js
router.get('/logout', function(req, res) {
  res.status(200).send({ auth: false, token: null });
});
```

Disclaimer: The logout endpoint is not needed. The act of logging out can solely be done through the client side. A token is usually kept in a cookie or the browser's localstorage. Logging out is as simple as destroying the token on the client. This `/logout` endpoint is created to logically depict what happens when you log out. The token gets set to `null`.

With this we've finished the **authentication** part of the tutorial. Want to move on to the authorization? I bet you do.

Do you have permission to be here?

To comprehend the logic behind an authorization strategy we need to wrap our head around something called **middleware**. Its name is self explanatory, to some extent, isn't it? Middleware is a piece of code, a function in Node.js, that acts as a bridge between some parts of your code.

When a request reaches an endpoint, the router has an option to pass the request on to the next middleware function in line. Emphasis on the word **next**! Because that's exactly what the name of the function is! Let's see an example. Comment out the line where you send back the user as a response. Add a `next(user)` right underneath.

```
router.get('/me', function(req, res, next) {

  var token = req.headers['x-access-token'];
  if (!token) return res.status(401).send({ auth: false, message: 'No token provided.' });

  jwt.verify(token, config.secret, function(err, decoded) {
    if (err) return res.status(500).send({ auth: false, message: 'Failed to authenticate t

    User.findById(decoded.id,
    { password: 0 }, // projection
    function (err, user) {
      if (err) return res.status(500).send("There was a problem finding the user.");
      if (!user) return res.status(404).send("No user found.");

      // res.status(200).send(user); Comment this out!
      next(user); // add this line
    });
  });
});

// add the middleware function
router.use(function (user, req, res, next) {
  res.status(200).send(user);
});
```

> ***Middleware*** functions are functions that have access to the request object (`req`), the response object (`res`), and the `next` function in the application's request-response cycle. The `next` function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.
> - Using middleware, expressjs.com

Jump back to postman and check out what happens when you hit the `/api/auth/me` endpoint. Does it surprise you that the outcome is exactly the same? It should be!

Disclaimer: Go ahead and delete this sample before we continue as it is only used for demonstrating the logic of using `next()`.

Let's take this same logic and apply it to create a middleware function to check the validity of tokens. Create a new file in the **auth** folder and name it **VerifyToken.js**. Paste this snippet of code in there.

```
var jwt = require('jsonwebtoken');
var config = require('../config');

function verifyToken(req, res, next) {
  var token = req.headers['x-access-token'];
  if (!token)
    return res.status(403).send({ auth: false, message: 'No token provided.' });

  jwt.verify(token, config.secret, function(err, decoded) {
    if (err)
    return res.status(500).send({ auth: false, message: 'Failed to authenticate token.' })

    // if everything good, save to request for use in other routes
    req.userId = decoded.id;
    next();
  });
}

module.exports = verifyToken;
```

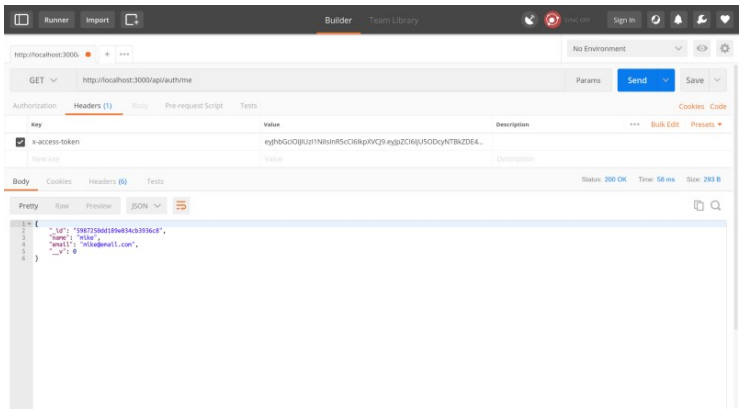
Let's break it down. We're going to use this function as a custom middleware to check if a token exists and whether it is valid. After validating it, we add the `decoded.id` value to the request (`req`) variable. We now have access to it in the next function in line in the request-response cycle. Calling `next()` will make sure flow will continue to the next function waiting in line. In the end, we export the function.

Now, open up the **AuthController.js** once again. Add a reference to

VerifyToken.js at the top of the file and edit the `/me` endpoint. It should now look like this:

```
// AuthController.js

var VerifyToken = require('./VerifyToken');

// ...

router.get('/me', VerifyToken, function(req, res, next) {

  User.findById(req.userId, { password: 0 }, function (err, user) {
    if (err) return res.status(500).send("There was a problem finding the user.");
    if (!user) return res.status(404).send("No user found.");

    res.status(200).send(user);
  });

});

// ...
```

See how we added `VerifyToken` in the chain of functions? We now handle all the authorization in the middleware. This frees up all the space in the callback to only handle the logic we need. This is an awesome example of how to write DRY code. Now, every time you need to authorize a user you can add this middleware function to the chain. Test it in Postman again, to make sure it still works like it should.



Feel free to mess with the token and try the endpoint again. With an invalid token, you'll see the desired error message, and be sure the code you wrote works the way you want.

Why is this so powerful? You can now add the `VerifyToken` middleware to any chain of functions and be sure the endpoints are secured. Only users with verified tokens can access the resources!

Wrapping your head around everything.

Don't feel bad if you did not grasp everything at once. Some of these concepts are hard to understand. It's fine to take a step back and rest your brain before trying again. That's why I recommend you go through the code by yourself and try your best to get it to work.

Again, here's the GitHub repository. You can catch up on any things you may have missed, or just get a better look at the code if you get stuck.

Remember, **authentication** is the act of logging a user in. **Authorization** is the act of verifying the access rights of a user to interact with a resource.

Middleware functions are used as bridges between some pieces of code. When used in the function chain of an endpoint they can be incredibly useful in authorization and error handling.

Hope you guys and girls enjoyed reading this as much as I enjoyed writing it. Until next time, be curious and have fun.

Do you think this tutorial will be of help to someone? Do not hesitate to share. If you liked it, please clap for me.

Adnan Rahić
Dev/Avocado at Sematext.com. Co-Founder at Bookvar.co. Author of "Serverless JavaScript by Example" - bit.ly/sls-js. Ex-Local leader at freeCodeCamp Sarajevo.