Node Express JSON Web Token

Implementing a Web API with Node, Express, MongoDB, and JWT Authentication

JWT authentication and authorization is a way for an application to handle users logging in, signing out, and validating permissions. This tutorial is an extension of the Node_Express_MongoDB_Mongoose tutorial.

Updating User Object

In the previous tutorial we ended with creating User mongoose model with a couple of properties. To further the tutorial into implementing JWT, we are going to need to make a few changes. In your user.js file, make the following changes that are highlighted below:

```
…
const userSchema = new mongoose.Schema({
  name: { type: String, required: true, minlength: 5, maxlength: 50 },
  email: { type: String, unique: true, required: true, minlength: 5, maxlengt
h: 255 },
  password: { type: String, required: true, maxlength: 1024, minlength: 5 },
  isGoldMember: { type: Boolean, default: false },
  shoppingCart: { type: [productSchema], default: [] },
});

const User = mongoose.model('User', userSchema);

function validateUser(user) {
  const schema = Joi.object({
    name: Joi.string().min(5).max(50).required(),
    email: Joi.string().min(5).max(255).required().email(),
    password: Joi.string().min(5).max(1024).required(),
  });
  return schema.validate(user);
}

exports.User = User;
exports.validateUser = validateUser;
```

Here we have added an email and password property to the userSchema object as well as updated our Joi validation function. The email and password will be used by a user to sign up and log into our application.

Creating User POST Route Handler

Now that we have created our user schema, we need a POST route endpoint where the client can request a new user to be created and added to our database. Within the users.js file add the following POST endpoint route handler:

```
router.post('/', async (req, res) => {
  try {
    const { error } = validateUser(req.body);

    if (error) return res.status(400).send(error.details[0].message);

    let user = await User.findOne({ email: req.body.email });
    if (user) return res.status(400).send('User already registered.');

    user = new User({
      name: req.body.name,
      email: req.body.email,
      password: req.body.password,
    });

    await user.save();
    return res.send({ _id: user._id, name: user.name, email: user.email });
  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});
```

First, we care calling the user validate function and checking if the body of the request is in the proper format.

If it is, we continue to query the database for the user based off the email. We are checking for the user based of the email because the email is a unique field in our schema.

Next, we check if the user returned from the query exists or not. If the user **does exist**, we return a status 400 accompanied with an error message letting the client know that the email the new user is trying to register with is already taken.

In the case that the email has not been taken, we then create a new user object with the properties for the body of the request. Once the new user object has been created, we call the '.save()' function and return the newly created user to the client.

Notice how we are not sending all the properties back to the client. Instead, we are only sending the user's _id, name, and email. This is because there is no need to send the password back to the client. In fact, you should never send the password back to the client.

As of right now, we are saving the password as plain text directly in the database. That means if someone malicious gained access to our database, he or she would be able to read all the users' passwords plainly. Rather than saving the passwords directly as they were sent to us, we are going to *hash* the passwords.

Hashing Passwords

It is important to store a user's password in a secure manner. Rather than saving the plain text (exact copy) of a password in the database, we are going to store a hashed version of the password. When we save a password as plain text in our database, any hacker who gains access to our database would have full access to all person's accounts. This is because the passwords are saved directly in the database as they were typed in.

For example, if you password was '1234', the plain text version in the database would look like this:

A better solution to saving passwords in a database are by using a *hash function*. Your application will take the password entered by the user and run it through a hashing function. This hash function will return a unique string of numbers and letters. The numbers and letters are much longer and different than the plain text version of the password. We then take this new string called the 'hashed password' and save it into our database.

The benefit of hashing a password is that a hacker can never "de-hash" a hashed password. Hashing is a one-way road where, once hashed, can never be de-hashed.

Once we save the hashed password to the database, when a user returns to the application and logs in, we take the password provided in the login form, re-hash it, and confirm that the hashes match up to the hashed passwords we have saved in the database.

For a visual explanation of how and why we hash passwords, we highly recommend watching this short video: https://www.youtube.com/watch?v=cczlpiiu42M

To hash our passwords, we will need to install a particularly useful and popular third-party package. In your terminal run the following command to install the bcrypt package.

```
npm i bcrypt
```

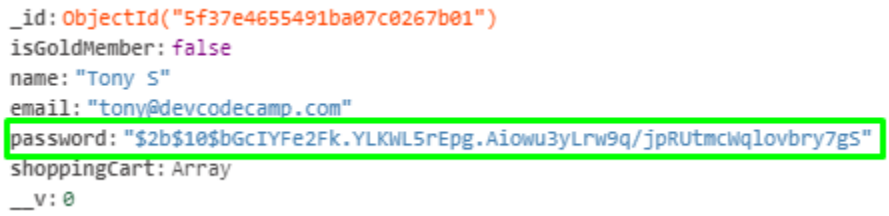
Bcrypt provides a variety of built in functionality needed to hash our passwords securely. We need to import the bcrypt module into the users.js file at the top. Add the following code to the users.js POST route handler as well:

```
const { User, validateUser} = require('../models/user');

const { Product, validateProduct } = require('../models/product');
const bcrypt = require('bcrypt');
const express = require('express');
const router = express.Router();

router.post('/', async (req, res) => {
  try {
    const { error } = validateUser(req.body);

    if (error) return res.status(400).send(error.details[0].message);

    let user = await User.findOne({ email: req.body.email });
    if (user) return res.status(400).send('User already registered.');

    const salt = await bcrypt.genSalt(10);
    user = new User({
      name: req.body.name,
      email: req.body.email,
      password: await bcrypt.hash(req.body.password, salt),
    });

    await user.save();
    return res.send({ _id: user._id, name: user.name, email: user.email });
  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});
```

To securely hash a password, we need a *salt*. A salt is a random string that is added before or after the password so the resulting hashed password will be different each time.

First, we create a salt by using the bcrypt module. Then, instead of saving 'req.body.password' directly to the user object, we pass in the 'req.body.password' and the salt to a hash function provided by bcrypt. This will successfully hash our password and add a bit of an extra random string (the salt) onto the beginning or end to make it completely unique.

If we test this endpoint in Postman you will notice the same '1234' password in the database is now:



Creating Login Functionality

Next, we are going to create our endpoint for allowing users to login to our application. In order to do this, we are going to create a new file in the routes folder. It is convention for create a file called 'auth' to handle login and logout operations for our application. Within the routes folder, create a file called 'auth.js' and add the following code:

```
const Joi = require('joi');
const express = require('express');
const router = express.Router();

function validateLogin(req) {
  const schema = Joi.object({
    email: Joi.string().min(5).max(255).required().email(),
    password: Joi.string().min(5).max(1024).required(),
  });
  return schema.validate(req);
}

module.exports = router;
```

IMPORTANT: make sure you add this new route to the index.js file for all request to '/api/auth'

Notice here the only thing we have included thus far is a validateLogin function. This function is responsible for validating the body of the request when someone attempts to login. As is with the majority of login forms, the client will send only an email and password. We then use the Joi module within this function to confirm that the body of the request contains these email and password properties before continuing.

Before we write the actual login function, lets add a couple imports we will need at the top of the file. Add the following modules to the auth.js file:

```
const Joi = require('joi');
const bcrypt = require('bcrypt');
const express = require('express');
const { User } = require('../models/user');
const router = express.Router();
```

Lets now create our login endpoint route handler. Add the following code above our new validateLogin function:

```
router.post('/', async (req, res) => {
  try {
    const { error } = validateLogin(req.body);
    if (error) return res.status(400).send(error.details[0].message);

    let user = await User.findOne({ email: req.body.email });
    if (!user) return res.status(400).send('Invalid email or password.');

    const validPassword = await bcrypt.compare(req.body.password, user.password);

    if (!validPassword) return res.status(400).send('Invalid email or password.')

    return res.send(true);
  } catch (ex) {
    return res.status(500).send(`Internal Server Error: ${ex}`);
  }
});
```

Perfect! First we are validating the body of the request with the validateLogin function that we just previously wrote.

Next, we search for the user based off the email that was provided on the client's login form. If the user **does not exist**, then we return the generic error message "Invalid email or password". We sent the error because the email provided does not match a user in the database, but we do not specify in the error that it was the email that was invalid. This is so we do not give hackers information about what emails exist or not and instead leave them guessing. That way they are unsure of whether the email or the password is valid.

In the case that the user does exist, we then call our handy bcrypt module once more. By utilizing the compare function, bcrypt will compare the two passwords. First, it will hash the password from the body of the request and see if it matches the hashed password in our user object that we just pulled from the database. This compare function will return a Boolean.

We check if the password is valid. If it is not valid, we send the same generic error message. If it is valid, we return true.

This would be a great point to stop and test a few of the endpoints we have created. Create a user and then attempt to login. You should receive 'true' as a response in Postman upon a successful login. In the next section we are going to replace responding with true to responding with a JSON Web Token.

JSON Web Tokens

A JSON Web Token (JWT) is a long string used to identify a user. You can think of this as a sort of passport that a user will use to validate their identity and gain access into a place or recourse within our application. Once a user successfully logs in to our application, we will send the client a newly generated JSON Web Token. The client is then responsible for storing the JWT and sending it back to the API within every request that the client makes. We will only allow the client to hit and access certain endpoints as long as they provide a valid JWT token within each request. In the case the user logs out, the client will delete the JSON Web Token that it is currently storing. That way, when the user re-visits the website, they will need to login once again in order to receive a new valid JSON Web Token.

A JSON Web Token consists of three specific parts: the header, payload, and signature. In this tutorial we will only be focusing on the payload portion of a JWT. If you want to review information about the header and signature, please revisit the JWT slide show.

The payload is going to hold public properties about the user. We will never store any private information in the payload since the JWT token can be decoded by anyone.

In order to generate our own JSON Web Token, we will need to install another third-party node module. Run the following command within your terminal:

```
npm i jsonwebtoken
```

Add the following import statements to the top of auth.js:

```
…
const config = require('config');
const jwt = require('jsonwebtoken');
const router = express.Router();
…
```

Here we have imported the ‘jsonwebtoken’ and ‘config’ modules into the auth.js file. Now that we have the ‘jsonwebtoken’ module imported into the file, we can generate a JSON Web Token to send to the client upon successful login.

Next, make the following highlighted changes to our post route handler in auth.js:

```
…
    if (!validPassword) return res.status(400).send('Invalid email or password.')

    const token = jwt.sign({ _id: user._id, name: user.name }, 'SomeSecretString'
);

    return res.send(token);
  } catch (ex) {
…
```

After all login validation and checks have passed, we call and use the JWT 'sign()' function to generate the token. The function expects an object that will populate the token payload (the token public properties). As you can see in our example, we are creating an object that has an _id and name properties. This means that the token that is generated will have a payload with those two properties. Those properties values are set to the id and the name of the user who just successfully signed in. We then send the token back to the client.

Lastly, the '.sign()' function expects a second argument. This second argument is a secret string. This string can be set to anything. It uses this secret string to hash the JWT's header and payload together to make the signature. This is important to keep hackers from trying to generate their own fake valid JSON Web Tokens. The string we use for this must remain secret to our application and we should never disclose this secret string to source controller or persons. Rather than storing the string directly in the function call parameter, it is best to keep it in a singular and secure place.

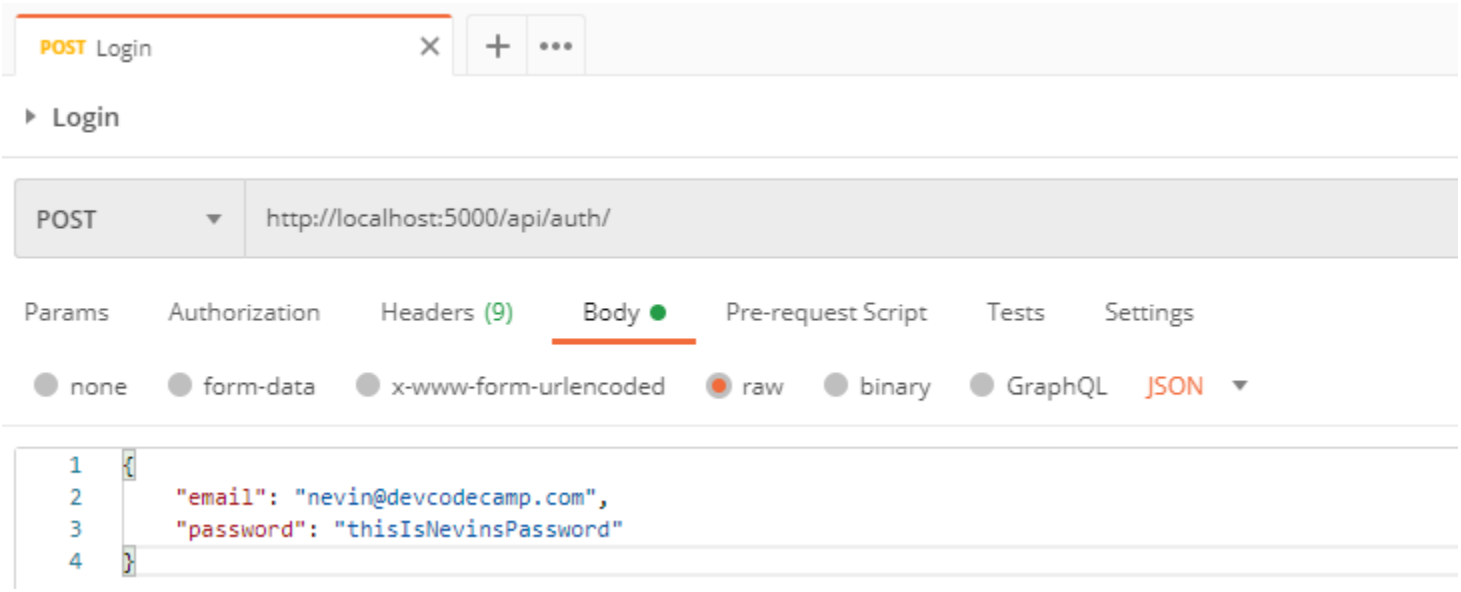
Go to your config folder and add a property to the JSON object in the default.json file:

```
{
  "mongoURI": "CONNECTION STRING HERE",
  "jwtSecret": "JWT SECRET HERE"
}
```

Finally, change the following line in the auth.js login POST route handler:

```
const token = jwt.sign({ _id: user._id, name: user.name }, config.get('jwtSecret'));
```

Let's test this out in Postman. Here is the request:



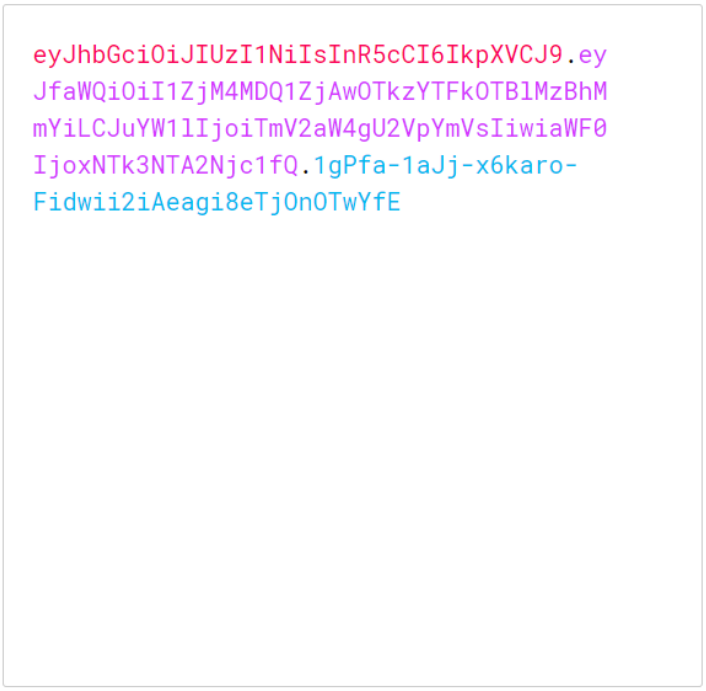
And here is the response:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI1ZjM4MDQ1ZjAwOTkzYTFkOTBlMzBhMmYiLCJuY
W1lIjoiTmV2aW4gU2VpYmVsIiwiaWF0IjoxNTk3NTA2Njc1fQ.1gPfa-1aJj-x6karo-
Fidwii2iAeagi8eTjOnOTwYfE
```

As we can see, the server responded to Postman with a long JSON Web Token string.

Next, copy the above web token (or your own if you are following along) and visit this website: https://jwt.io/#encoded-jwt

Here you can past the token in the encoded window and actually view the public payload properties and values!



You will also notice the 'iat' property that was included even though we did not include it in the 'jwt.sign()' function object. This iat property is a timestamp that will allow us to determine how old the token is. That way, if we wanted to implement some sore of auto sign out after a period, we could check that to confirm that it is within that desired time frame.

Editing Registration to Return Token

Typically, when you register for an application, you are immediately signed in. That is what we are going to tackle next. We are going to edit our user registration endpoint route handler to not only return the newly created user's details, but also immediately provide a valid token.

First, lets add the following imports to the top of the users.js file to account for the changes we are about to make:

```
…
const bcrypt = require('bcrypt');
const config = require('config');
const jwt = require('jsonwebtoken');
const express = require('express');
…
```

Add the following code to the users.js POST endpoint route handler for creating new users:

```
…
    email: req.body.email,
      password: await bcrypt.hash(req.body.password, salt),
    });

    await user.save();

    const token = jwt.sign(
            { _id: user._id, name: user.name },
            config.get('jwtSecret')
      );

    return res
      .header('x-auth-token', token)
      .header('access-control-expose-headers', 'x-auth-token')
      .send({ _id: user._id, name: user.name, email: user.email });
…
```

Here we are creating a token for the new user that was just created just like when a user logs in. What you will notice differently here is how we are sending that token back to the client.

Instead of (how we are sending it back in auth.js):

```
return res.send(token);
```

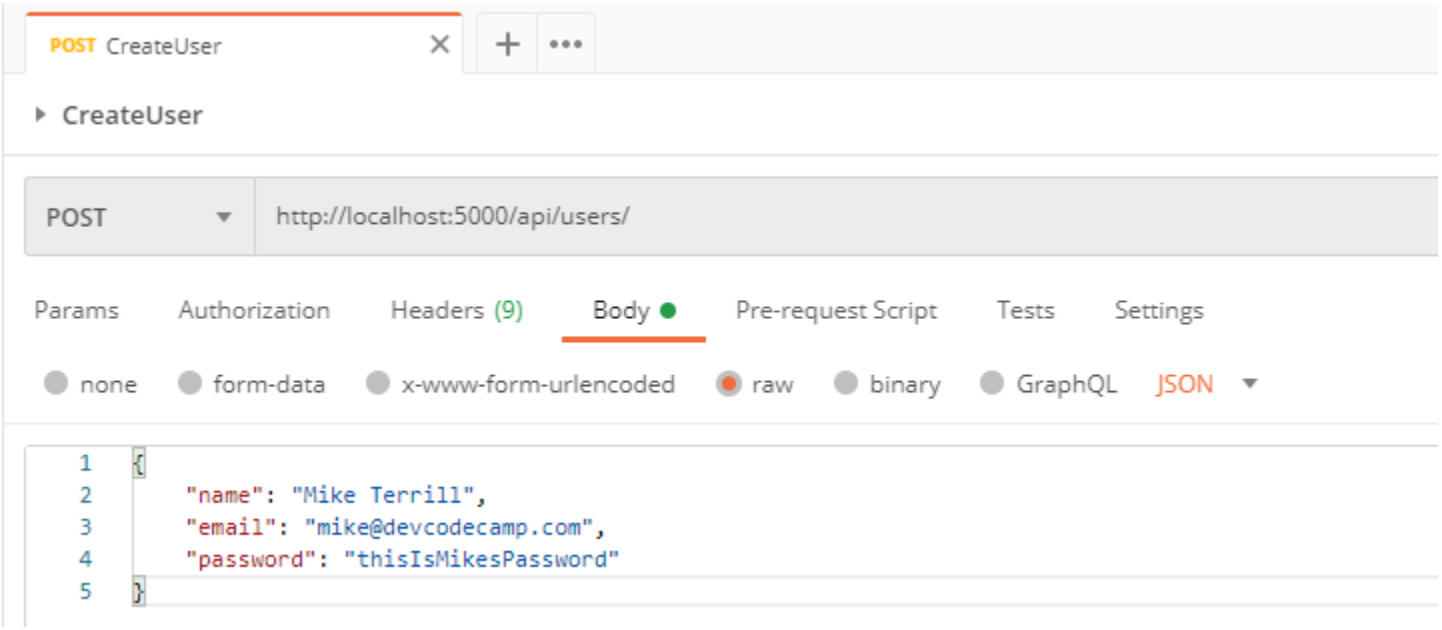
In users.js we are sending it back through a header:

```
    return res
      .header('x-auth-token', token)
      .header('access-control-expose-headers', 'x-auth-token')
      .send({ _id: user._id, name: user.name, email: user.email });
```

Headers are used as another means to pass data between requests and responses. Headers represent the meta-data associated with the API request or response.

In our case above, we are sending the newly created users id, name, and email back in the body of the response and sending the authentication token (JWT token) back in a custom defined header. It is common practice to return the response of an authentication token back in a header when different information is being sent in the body. The header with 'access-control-expose-headers' enables a client to have access to 'x-auth-token' header when receiving a response from the server.

Let's create a new user and see an example of the result in Postman. Here is the request:

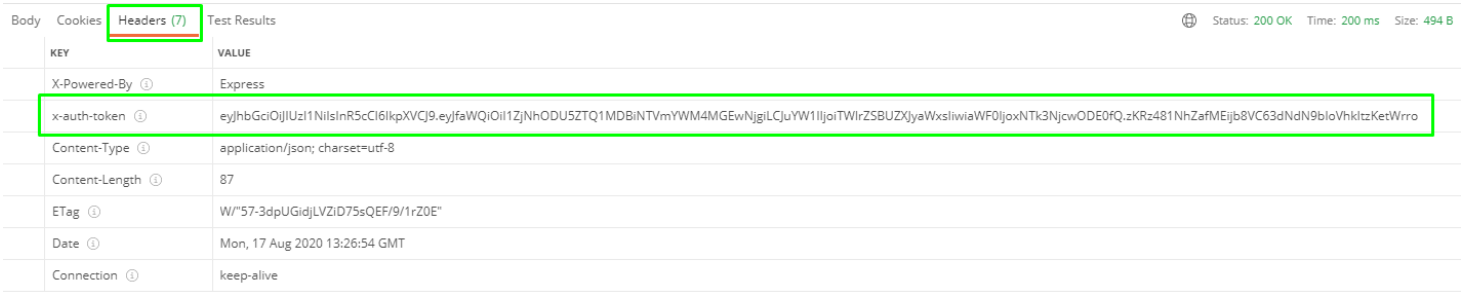


Here is the response:

Notice we are on the "Body" tab at the top. The server responded with the new user information directly in the body of the response as we specified with the line below:

```
return res
  .header('x-auth-token', token)
  .send({ _id: user._id, name: user.name, email: user.email });
```

But where is our header? Well if you notice on the picture above, there are two other tabs that contain more information about the response. One is "Cookies" and the other is "Headers". Let's take a look at the Headers tab: (TIP: zoom in on the PDF document to see the picture in more detail)



Notice how our new "x-auth-token" header is present with a JSON Web Token as its value! When consuming this API on a client side such as React or Angular, you will be able to read and extract the JSON Web Token from the header.

Extracting Token Logic to Mongoose Model

At this point, you may have noticed a small deficiency with our current token generation code. Each time we generate a token we include the '_id' and 'name' properties to the token's payload. We have this same logic on both the users.js and auth.js files. If we were to add another property to the token's payload, we would need to go to both files and make the appropriate changes. If I generated a token in 20 different places (not likely), we would have to go to all 20 locations to make sure they all had the same changes. In this section, we are going to extract this token generation logic to the User mongoose model.

To do this we are going to create a new method on the User model. Notice how I said method here rather than function. Because we are about to add a custom action (function) to the User model which is a class, we call it a method.

First, add the two new imports statements need for the user.js file in the models folder:

```
const config = require('config');
const jwt = require('jsonwebtoken');
```

Next, add the following highlighted to the same file:

```
…
  isGoldMember: { type: Boolean, default: false },
  shoppingCart: { type: [productSchema], default: [] },
});

userSchema.methods.generateAuthToken = function () {
  return jwt.sign({ _id: this._id, name: this.name }, config.get('jwtSecret'));
};

const User = mongoose.model('User', userSchema);
…
```

Here we are accessing the userSchema's methods property and adding our own new key value pair. We create a new property on the methods property called 'generateAuthToken' and set that equal to a function that returns a new JSON Web Token.

Now when we want to generate a new JWT token, we simply call this method that is present on our user object!

The benefit of this is now, in the case we want to add a new property to the token's payload, we can make the change here and it will be reflected everywhere we call the generateAuthToken method.

Replace the following line in both the auth.js and users.js files.

Replace:

```
const token = jwt.sign({ _id: user._id, name: user.name }, config.get('jwtSecret'));
```

With:

```
const token = user.generateAuthToken();
```

You can now **remove** these to module imports from the top of both the users.js and auth.js files:

```
const config = require('config');
const jwt = require('jsonwebtoken');
```

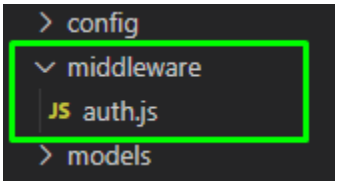
JWT Authorization Middleware

Now that we have completed all process regarding sending the JSON Web Token to the client, we need to run authorization checks on tokens that are sent back to the server. Remember, we provide a token to the client once they have successfully registered or logged in. Once a user has done either of those actions, we must require the client to send a valid JWT token back to the server with every request, so the server is aware of who is accessing the endpoint or resource.

This will be a perfect use case for middleware with our server application. Middleware is a function or set of functions that run before the request hits its desired endpoint. We are going to create a middleware function that confirms the JSON Web Token is present in the request and that the token is a valid token that was created by our server.

Whenever the client sends a token to our server, we expect it to come within the request's header. We are going to expect that the request has a header called 'x-auth-token' that contains a valid token that was generated by our server.

Add a new folder to your application called 'middleware' and create a new file called 'auth.js'. (although named the same, this auth.js file is different then the one we have in our routes folder):



Next, add the following code to the auth.js file that we just created:

```
const jwt = require('jsonwebtoken');
const config = require('config');

function auth(req, res, next) {

  const token = req.header('x-auth-token');
  if (!token) return res.status(401).send('Access denied. No token provided.');

  try {
    const decoded = jwt.verify(token, config.get('jwtSecret'));
    req.user = decoded;
    return next();
  } catch (ex) {
    return res.status(400).send('Invalid token.');
  }

}

module.exports = auth;
```

Lets go through this line by line. First, we are importing the config and jsonwebtoken modules because we will utilize them within our middleware function.

```
  const token = req.header('x-auth-token');
  if (!token) return res.status(401).send('Access denied. No token provided.');
```

Here we are accessing the value for the header 'x-auth-token'. We then check if a value for that header exists or not. If it does not exist, then the client did not send the token, so we proceed to send a status code of 401 and a message of 'Access denied'. (status code 401 means 'Unauthorized')

```
const decoded = jwt.verify(token, config.get('jwtSecret'));
req.user = decoded;
return next();
```

In the case that the token does exist, we then create a const called 'decoded' and set it equal to the result of the built in jsonwebtoken 'verify()' function. This verify function takes in two arguments. First, the token we retrieved from the header, and second, the same JWT secret we used to encode the token in the first place. The result of this function, if successful, will be the payload of the token. In our case, since our token payload is the user's object id and name, decoded will be an object that contains the user's object _id and name!

```
} catch (ex) {
  return res.status(400).send('Invalid token.');
}
```

You will notice that the above text is wrapped in a try catch statement. That is because if the verify function fails (token is invalid), it will throw an exception. That way in the catch we can return a response of status code 400 and a message of 'Invalid token'.

Applying JWT Auth to Specific Routes

We do not want to apply this new JWT authorization middleware globally to our application. There are some route endpoints that we do not want to require a token. These endpoints are endpoints such as the register and login endpoints. Instead, we are going to apply this JWT authorization to the specific endpoints we desire.

Up until this point, we have only used two arguments for the '.get', '.post', '.put', and '.delete' express functions.

Those two arguments are first the route:

```
router.get('/', async (req, res) => {});
```

and the route handler function:

```
router.get('/', async (req, res) => {});
```

These endpoint functions actually take an optional argument: middleware functions

```
router.get('/', <middleware functions here>, async (req, res) => {});
```

Lets now edit our shopping cart endpoints to only allow authorized users to perform CRUD operations on their shopping cart. Add the following import statement to the users.js file:

```
…
const bcrypt = require('bcrypt');
const auth = require('../middleware/auth');
const express = require('express');
…
```

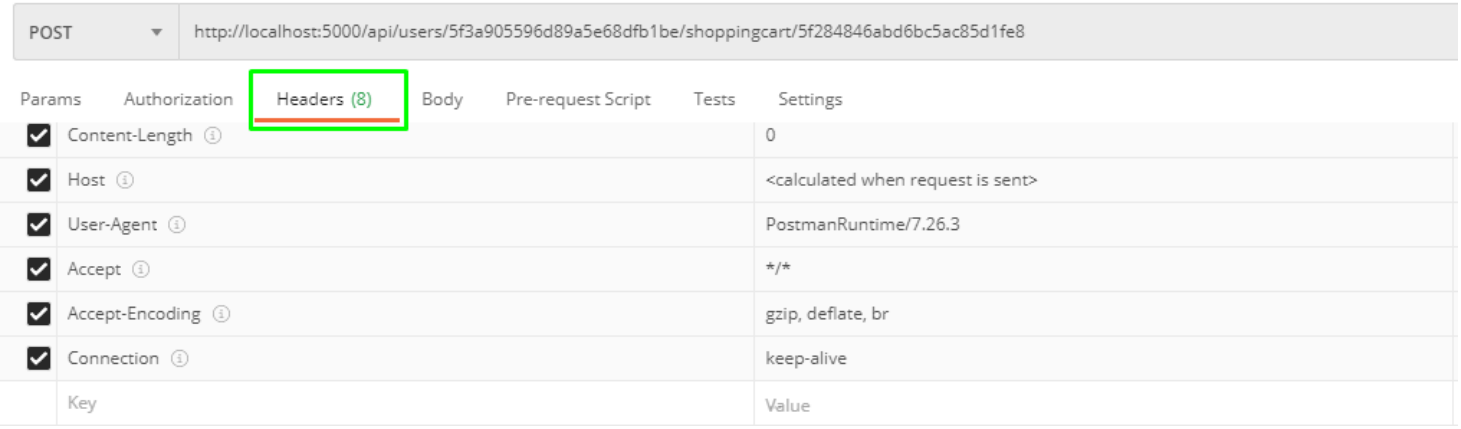
Next, add make the following changes to theses endpoint route handler functions:

```
router.post('/:userId/shoppingcart/:productId', auth, async (req, res) => {
```

```
router.put('/:userId/shoppingcart/:productId', auth, async (req, res) => {
```

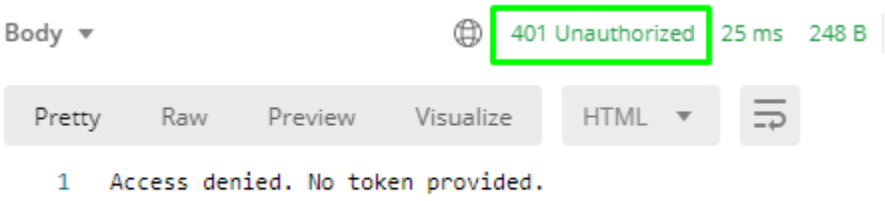
```
router.delete('/:userId/shoppingcart/:productId', auth, async (req, res) => {
```

Let's test this out. Here is a request adding a new product to a user's shopping cart:

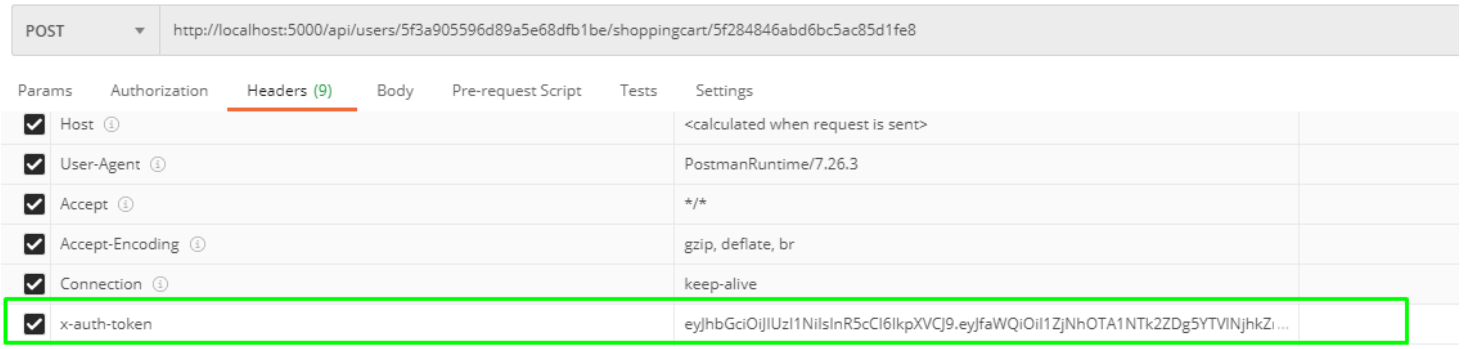


Notice that this request does not contain the 'x-auth-token' header with its value.

Here is the response:



Let's make the same request but this time with the header and token provided from a successful login. Here is the request:



Here is the response:

Here we have sent a valid JSON Web Token with our request to add a new product to a specific user. Because the authorization passed, it successfully added the product to the user's shopping cart!

Implementing Role Based Route Authentication

The last thing we are going to implement for our server is role-based route authentication. In the previous Mongoose MongoDB tutorial, we created endpoints to handle all CRUD operations for a products collection. In a real-world application, not every user should have permission to perform CRUD operations for products. Instead, we would only want the admin of the web application to have the permission to add, update, and delete products.

Let's add another property to our userSchema. Add the following code the user.js file:

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true, minlength: 5, maxlength: 50 },
  email: { type: String, unique: true, required: true, minlength: 5, maxlength: 2
55 },
  password: { type: String, required: true, maxlength: 1024, minlength: 5 },
  isGoldMember: { type: Boolean, default: false },
  shoppingCart: { type: [productSchema], default: [] },
  isAdmin: { type: Boolean, default: false },
});
```

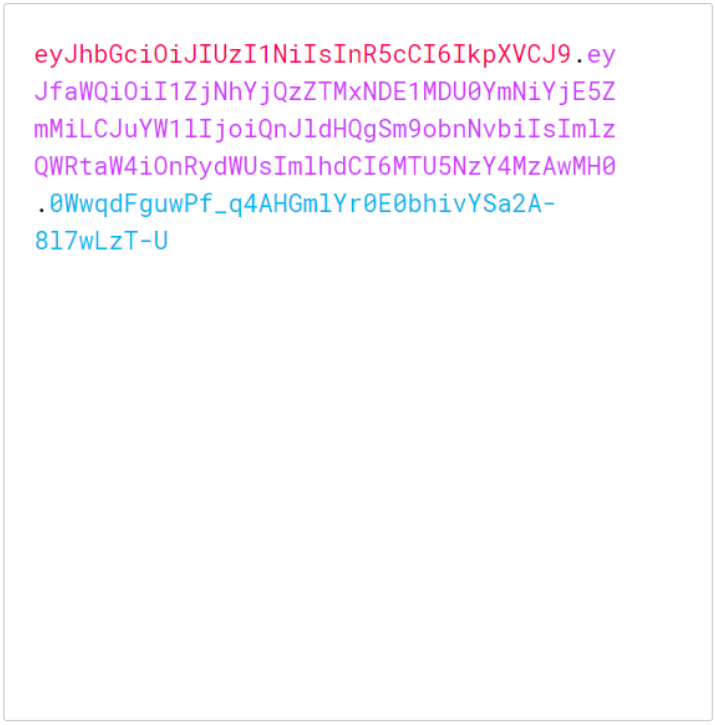
Now each user will have an isAdmin boolean which we will use to determine whether the user trying to access the endpoint is an admin or not.

In order for use to achieve this goal we need to now add the isAdmin property to the JWT token payload. Add the following changes to the generateAuthToken method on the user.js file:

```
userSchema.methods.generateAuthToken = function () {
  return jwt.sign({ _id: this._id, name: this.name, isAdmin: this.isAdmin },
                  config.get('jwtSecret'));
};
```

Create a new user and go into MongoDB to change the isAdmin property to 'true':

Next, login with the new user to receive a new JWT token and paste it in the encoded window at: https://jwt.io/#encoded-jwt



As we can see the payload now contains the isAdmin property!

Next, we need to create some middleware that we will use to verify the incoming request as an admin user or now. Create a new file in the middleware folder called 'admin.js' and add the following code:

```
function admin(req, res, next) {

  if (!req.user.isAdmin) return res.status(403).send('Access denied.');

  return next();
}

module.exports = admin;
```

First we check if the req.user.isAdmin property is false. If it is, we return a status code of 403 and a message of 'Access denied.'. If it is successful, we let the request continue down the request pipeline.

How does request have this user object? – If you go back to the auth middleware we wrote, if the token is valid, we gain access to the tokens payload and store it in a const called 'decoded'. We then create a new property on the request object called 'user' and assign it equal to the decoded const. The request then continues down the pipeline to the next middleware function, which in the case is this one, the req has a property equal to the token's payload.

Lets now edit the products.js endpoints to use both the auth and admin middleware.

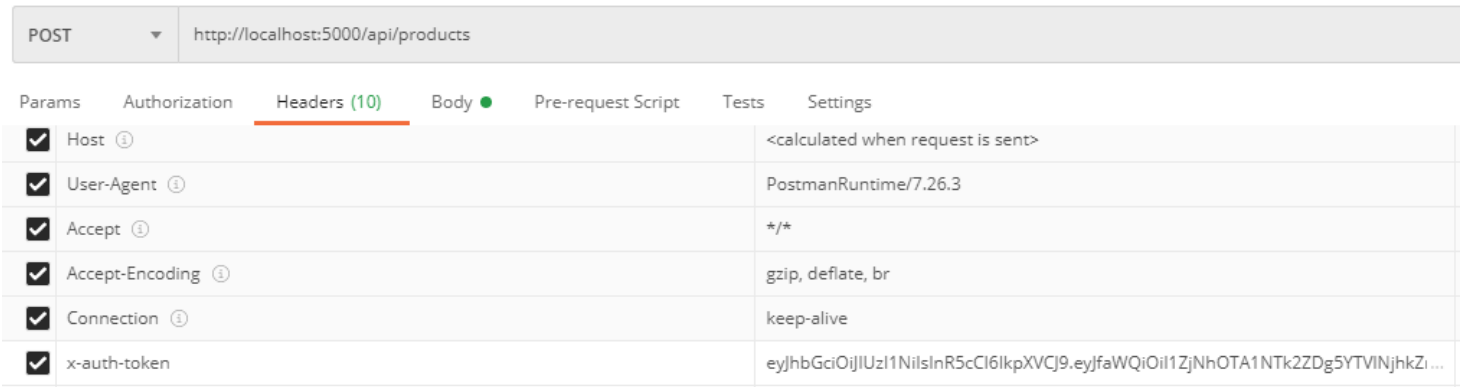
First, we need to bring in the auth and admin module imports at the top of the products.js file:

```
const { Product, validateProduct } = require('../models/product');
const auth = require('../middleware/auth');
const admin = require('../middleware/admin');
const express = require('express');
…
```

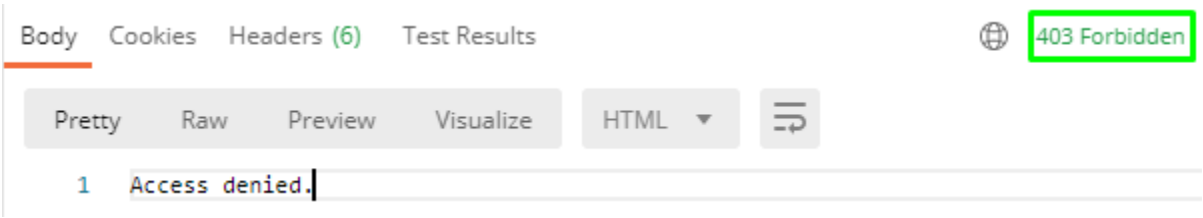
Here is an example of implementing this on the create product endpoint:

```
router.post('/', [auth, admin], async (req, res) => {
```

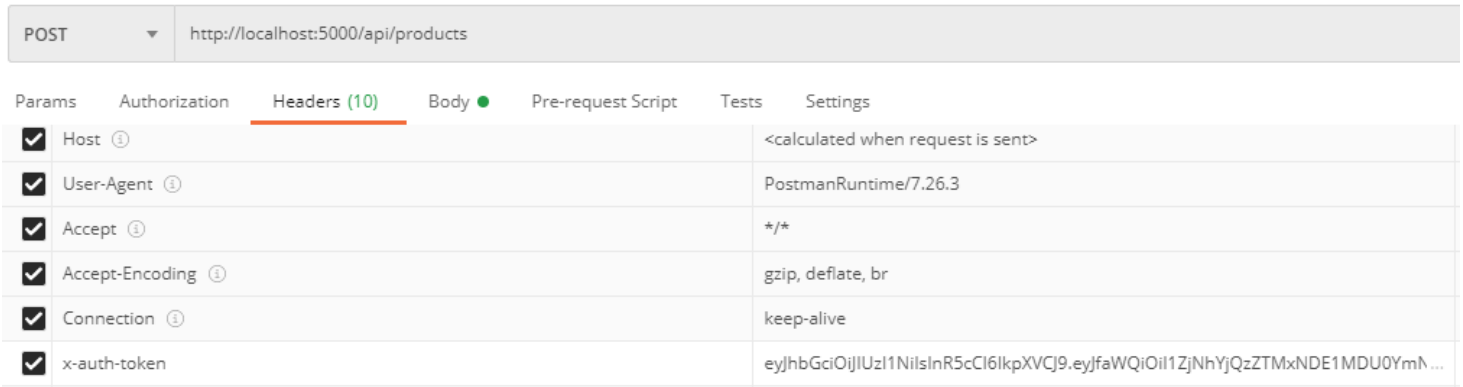
Finally, lets test this in Postman. Here is a request from someone who is not an admin: (the token in the request below is for a user who is not an admin)



And here is the response:



Finally, here is a request from Brett, our admin user: (the token here belongs to our admin user)



And here is the response:

Logging Users Out

Since the client is responsible for sending a valid JWT token for every authorized request, there is no implementation needed on our server side to log a user out. When a user logs out from the client side, the client is then responsible for deleting the token that it is currently storing. That way, a user is required to re-login to obtain a new valid token!

Conclusion

In this tutorial, we have learned about JSON Web Tokens and how to generate them for a client within an Node.js and Express.js application.

The topics we have covered include:

- Hashing Passwords
- Creating Login Functionality
- JSON Web Tokens
- Headers
- Custom Mongoose Model Methods
- JWT Authorization Middleware
- Applying Middleware to Specific Routes
- Implemented Role-Based Route Authentication