**Authentication and Authorization in Node.js**

[[Niman Ransindu](https://medium.com/@niman.ransindu?source=post_page-----9147e056fc7c--------------------------------)](https://medium.com/@niman.ransindu?source=post_page-----9147e056fc7c--------------------------------)

[Niman Ransindu](https://medium.com/@niman.ransindu?source=post_page-----9147e056fc7c--------------------------------)

·

Follow

5 min read

·

5 days ago

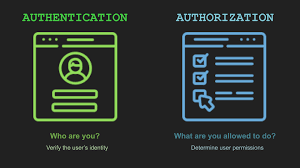
53

Making sure user data is secure is crucial in the field of web development. Verifying user identities through authentication and limiting access to particular features or resources through permission are two crucial components of this. Within this guide, we’ll explore a strong bcrypt authentication mechanism and implement authorization using JSON Web Tokens (JWT).



**Authentication** verifies a user’s identity, ensuring they are who they claim to be. It’s analogous to checking IDs at a club entrance. Only authorized individuals gain access.

**Authorization**, on the other hand, controls what a user can do within the application. Imagine a bank vault with individual safety deposit boxes. Each user has a key to access their own box, but not those of others.



<https://medium.com/geekculture/authentication-and-authorization-a5a2eafdde16>

**1. Authentication with Bcrypt**

**What is Bcrypt?**

* Brief introduction to Bcrypt and its role in secure password storage.
* It converts passwords into unreadable hashes, protects against brute-force attacks, and eliminates the need for plain text storage. It’s widely used in web applications and offers better security than older hashing algorithms.

For the implementation below, let’s assume a user model for a better understanding.

//models  
  
//creating user model for use in database usercollection  
const mongoose=require("mongoose");  
  
  
//create the user template  
const userSchema=new mongoose.Schema({  
 name:{  
 type:String,  
 reuired:true,  
 trim:true  
  
 },  
 email:{  
 type:String,  
 reuired:true,  
 unique:true,  
 trim:true  
 },  
 password:{  
 type:String,  
 trim:true,  
 required:true,  
 },  
 tokens:[ //token array []  
 {  
 token:{  
 type:String,  
 }  
 }  
 ]  
});  
  
//create the model using template  
const User=mongoose.model("User",userSchema);  
  
//export the model  
module.exports=User;

**Integrating Bcrypt in Node.js**

**installing the Bcrypt library using npm**

npm install bcrypt

**Integrating Bcrypt with the user model for password hashing during user registration.**

//models  
  
const bcrypt = require('bcrypt');  
const saltRounds = 10;  
  
// ... (User model definition)  
  
// Middleware for hashing password before saving to the database  
userSchema.pre('save', async function (next) {  
 const user = this;  
  
 if (user.isModified('password')) {  
 user.password = await bcrypt.hash(user.password, saltRounds);  
 }  
  
 next();  
});

***What is saltRounds?***

salt rounds refer to the number of times the hashing algorithm iterates over the password and its salt. This iteration process increases the complexity of the hash and makes it computationally expensive for attackers to crack.

**Verifying Passwords During Authentication**

//models  
  
// Adding a static method for finding user by credentials  
userSchema.statics.findByCredentials = async function (email, password) {  
 try {  
 const user = await this.findOne({ email });  
 if (!user) {  
 throw new Error("UserNotFound!");  
 }  
  
 const isMatch = await bcrypt.compare(password, user.password);  
  
 if (!isMatch) {  
 throw new Error("PasswordIsWrong!");  
 }  
  
 return user;  
 } catch (error) {  
 throw new Error(error.message);  
 }  
};

**2. Authorization with JSON Web Tokens (JWT)**

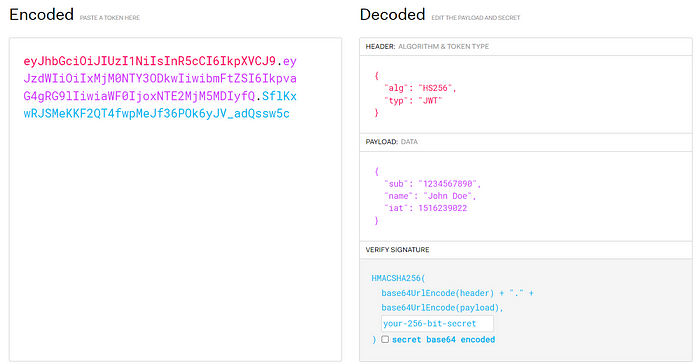
**What are JSON Web Tokens?**

JWT is a most popular choice for securing node.js APIs. It allows you to issue tokens upon successful authentication and validate those tokens on the following requests.

Structure of a JWT

A JWT consists of three parts separated by dots

1. Header: Contains information about the token type and signing algorithm used.
2. Payload: Contains claims, which are statements about the user or application. These claims can be used for authentication and authorization purposes.
3. Signature: Ensures the integrity of the token and prevents tampering.



<https://jwt.io/#debugger-io>

**Implementation Steps**

**Install jsonwebtoken**

npm install jsonwebtoken

**Extend the user model to include a method for generating JWT tokens.**

//models  
  
const jwt = require('jsonwebtoken');  
const secretKey = 'myscrect'; // Replace with a secure key  
  
// Method to generate JWT token for user  
userSchema.methods.generateAuthToken = async function () {  
 const user = this;  
 const token = jwt.sign({ \_id: user.\_id.toString() }, secretKey);  
 user.tokens = user.tokens.concat({ token });  
 await user.save();  
 return token;  
};

**Implementing JWT Authentication Middleware**

Now, let’s create a middleware function auth that verifies JWT tokens in incoming requests.

//middleware  
  
const jwt = require("jsonwebtoken");  
const User = require("../models/user");  
  
const auth = async (req, res, next) => {  
 try {  
 console.log('>Auth');  
   
 // Check if the Authorization header exists  
 const authorizationHeader = req.header("Authorization");  
 if (!authorizationHeader) {  
 throw new Error("Authorization header missing");  
 }  
  
 const token = authorizationHeader.replace("Bearer ", "");  
 const secretKey = 'myscrect'  
 const decoded = jwt.verify(token, secretKey);  
  
 const user = await User.findOne({  
 \_id: decoded.\_id,  
 "tokens.token": token,  
 });  
  
 if (!user) {  
 throw new Error("UnauthorizedUserDetected!");  
 }  
  
 req.user = user;   
 req.token = token;  
 next();  
 } catch (err) {  
 res.status(401).send({ error: "Please authenticate" });  
 }  
};  
  
  
module.exports = auth;

By integrating this JWT authentication middleware and the related methods into your Node.js and MongoDB application, you’ll enhance its security by controlling access to endpoints based on valid JWT tokens issued to authenticated users.

**Using the Auth Middleware for Endpoint Protection**

**Protecting a Secure Endpoint**

Consider a scenario where you want to create a secure endpoint that provides sensitive user data. Let’s say you have a route for fetching the user’s profile.

**Import the Router and auth Middleware**

//routes  
  
const express = require('express');  
const router = express.Router();  
const auth = require('../middleware/auth');

**Secure the Endpoint**

router.get("/users/me",auth,async(req,res)=>{  
 console.log(">myProfile")  
 try {  
 const user=await User.findById(req.user.\_id;)  
 console.log(user);  
 if(!user){  
 return res.status(404).send();  
 }  
 res.status(200).send(user);  
 } catch (error) {  
 res.status(400).send(error);  
 }  
})

In this example, the auth middleware is applied to the /users/me endpoint. The middleware ensures that only requests with a valid JWT token in the Authorization header can proceed to the route handler.

**Handling Authentication State. Logging Out and Reauthentication**

In our authentication system, logging out invalidates the current user’s token. Consequently, for subsequent secure operations, such as accessing secured endpoints or fetching user data, the user needs to log in again to obtain a new and valid JSON Web Token (JWT).

router.post('/users/logout',auth,async(req,res)=>{  
 try{  
 req.user.tokens=req.user.tokens.filter((token)=>{ //token object  
 return token.token!=req.token; //if return value is true ,token will not be removed else token will be removed  
 })  
 await req.user.save()  
 res.send();  
 }catch(err){  
 res.status(500).send();  
  
 }  
});

Login again to obatain valid Jason Web Token(JWT)

router.post('/users/login', async (req, res) => {  
 try {  
 console.log(">Login")  
 console.log(req.body)  
 const user = await User.findByCredentials(req.body.email, req.body.password);  
  
 const token=user.generateAuthToken();  
 res.status(200).send({user,token});   
 } catch (err) {  
 res.status(404).send({ error: 'Invalid credentials' });  
 }  
});

Understanding this flow is crucial for maintaining a secure authentication system, ensuring that only authorized users with valid tokens can access protected resources.

NRM : [www.linkedin.com/in/niman-ransindu-92b527239](http://www.linkedin.com/in/)