**JWT Authentication in Nodejs — Refresh JWT with Cookie-based Token**

[[](https://medium.com/@techsuneel99?source=post_page-----37348ff685bf--------------------------------)](https://medium.com/@techsuneel99?source=post_page-----37348ff685bf--------------------------------)

[Suneel Kumar](https://medium.com/@techsuneel99?source=post_page-----37348ff685bf--------------------------------)

·

Follow

6 min read

·

Feb 3

244

5



Photo by [Ferhat Deniz Fors](https://unsplash.com/@ferhat?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

JSON Web Tokens (JWTs) are a popular method of authentication that allow you to securely transmit information between parties as a JSON object. In this article, we’ll be diving into the details of JWT authentication in a Node.js application and exploring the use of refresh tokens to extend the life of our JWTs.

To start, let’s take a quick overview of JWT and its components.

**What is JWT?**

A JWT is a compact and self-contained JSON object that contains information about the authentication of a user. This information can be verified and trusted because it is digitally signed using a secret key. The main components of a JWT are:

1. Header: This contains information about how the JWT is encoded, such as the algorithm used for signing the token.
2. Payload: This contains the claims. Claims are statements about an entity (typically, the user) and additional metadata. There are three types of claims: registered, public, and private claims.
3. Signature: This is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn’t changed along the way. The signature is created by taking the encoded header, the encoded payload, and a secret key, then signing them using a specified algorithm.

Here’s an example of a JWT in its compact form:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c

**Setting up the environment**

Before we dive into the implementation, let’s set up our development environment. We’ll be using the following tools and packages:

1. Node.js: We’ll be using Node.js as our server-side language.
2. Express: This is a popular web framework for Node.js that we’ll be using to handle HTTP requests.
3. jsonwebtoken: This is a popular library for generating and verifying JWTs in Node.js.

Let’s start by creating a new Node.js project using the following command:

npm init

Next, let’s install the required packages by running the following command:

npm install express jsonwebtoken

**Implementing JWT Authentication**

Now that we’ve set up our environment, let’s start implementing JWT authentication in our Node.js application.

First, let’s create a new file called index.js and add the following code to set up our Express application:

const express = require('express');  
const app = express();  
const port = 3000;  
  
app.listen(port, () => {  
 console.log(`Server running at http://localhost:${port}`);  
});

Next, let’s create a simple route that will return a message if the user is authenticated:

app.get('/protected', (req, res) => {  
 res.send('Welcome to the protected route');  
});

Now, let’s implement the authentication process by adding middleware that will check for the presence of a valid JWT in the Authorization header of incoming requests:

const jwt = require('jsonwebtoken');  
const secretKey = 'secret\_key';  
  
const authenticate = (req, res, next) => {  
 const token = req.headers['authorization'];  
 if (!token) {  
 return res.status(401).send('Access Denied. No token provided.');  
 }  
  
 try {  
 const decoded = jwt.verify(token, secretKey);  
 req.user = decoded;  
 next();  
 } catch (error) {  
 return res.status(400).send('Invalid Token.');  
 }  
};  
  
app.get('/protected', authenticate, (req, res) => {  
 res.send('Welcome to the protected route');  
});

In the code above, we imported the jsonwebtoken library and created a secret key for signing and verifying our JWTs. Then, we created a middleware function called authenticate that will be used to check for a valid JWT in the Authorization header of incoming requests.

If a token is present, we use the verify method from the jsonwebtoken library to decode the token and check its validity. If the token is valid, we add the decoded information to the req object and call the next function to proceed to the next middleware or route handler.

If no token is provided or if the token is invalid, we return an error message with a 401 or 400 status code, respectively.

**Generating JWTs**

Now that we have our authentication process in place, let’s create a route for generating JWTs. This could be done when a user logs in, for example.

app.post('/login', (req, res) => {  
 const user = {  
 id: 1,  
 username: 'john.doe'  
 };  
  
 const token = jwt.sign({ user }, secretKey, { expiresIn: '1h' });  
  
 res.header('Authorization', token).send(user);  
});

In the code above, we created a /login route that will generate a JWT when hit with a POST request. We created a mock user object and used the sign method from the jsonwebtoken library to generate a JWT. We specified the secret key and set the expiresIn option to 1h, which means that the token will expire after one hour.

Finally, we added the generated JWT to the Authorization header of the response and sent the user information back to the client.

**Implementing Refresh Tokens**

One issue with JWTs is that they are short-lived and need to be refreshed regularly. To address this, we can use refresh tokens.

Refresh tokens are separate tokens from access tokens, and they can be used to generate new access tokens. This allows us to keep our access tokens short-lived for security reasons while still allowing the user to remain authenticated for longer periods of time.

In this section, we’ll explore the implementation of refresh tokens using cookies.

Let’s start by modifying the /login route to generate both an access token and a refresh token:

app.post('/login', (req, res) => {  
const user = {  
id: 1,  
username: 'john.doe'  
};  
  
const accessToken = jwt.sign({ user }, secretKey, { expiresIn: '1h' });  
const refreshToken = jwt.sign({ user }, secretKey, { expiresIn: '1d' });  
  
res  
.cookie('refreshToken', refreshToken, { httpOnly: true, sameSite: 'strict' })  
.header('Authorization', accessToken)  
.send(user);  
});

Next, we’ll create a new route for refreshing access tokens. This route will receive a refresh token from the client and use it to generate a new access token:

app.post('/refresh', (req, res) => {  
 const refreshToken = req.cookies['refreshToken'];  
 if (!refreshToken) {  
 return res.status(401).send('Access Denied. No refresh token provided.');  
 }  
  
 try {  
 const decoded = jwt.verify(refreshToken, secretKey);  
 const accessToken = jwt.sign({ user: decoded.user }, secretKey, { expiresIn: '1h' });  
  
 res  
 .header('Authorization', accessToken)  
 .send(decoded.user);  
 } catch (error) {  
 return res.status(400).send('Invalid refresh token.');  
 }  
});

In the code above, we created a new route for refreshing access tokens. This route checks for the presence of a refresh token in the cookies, and if a token is found, we use the verify method to check its validity. If the refresh token is valid, we generate a new access token with the same information as the original token and send it back to the client.

Finally, we can update the authenticate middleware to check for both the access token and refresh token and refresh the access token if necessary:

const authenticate = (req, res, next) => {  
 const accessToken = req.headers['authorization'];  
 const refreshToken = req.cookies['refreshToken'];  
  
 if (!accessToken && !refreshToken) {  
 return res.status(401).send('Access Denied. No token provided.');  
 }  
  
 try {  
 const decoded = jwt.verify(accessToken, secretKey);  
 req.user = decoded.user;  
 next();  
 } catch (error) {  
 if (!refreshToken) {  
 return res.status(401).send('Access Denied. No refresh token provided.');  
 }  
  
 try {  
 const decoded = jwt.verify(refreshToken, secretKey);  
 const accessToken = jwt.sign({ user: decoded.user }, secretKey, { expiresIn: '1h' });  
  
 res  
 .cookie('refreshToken', refreshToken, { httpOnly: true, sameSite: 'strict' })  
 .header('Authorization', accessToken)  
 .send(decoded.user);  
 } catch (error) {  
 return res.status(400).send('Invalid Token.');  
 }  
 }  
};

In the updated code, we first check for the presence of both access and refresh tokens. If only an access token is provided, we check its validity and continue the request if it’s valid. If the access token is invalid, we check for the presence of a refresh token. If a refresh token is found, we verify its validity and generate a new access token based on the information in the refresh token. If the refresh token is invalid, we send an error message to the client.

With these changes in place, we now have a complete JWT authentication system with a refresh token mechanism. To use this system, clients can first make a request to the /login route to receive an access token and refresh token. They can then include the access token in the Authorization header of all subsequent requests to access protected routes. If the access token expires, the client can make a request to the /refresh route to receive a new access token.

We hope this article has been helpful in demonstrating how to implement JWT authentication with a refresh token mechanism in Node.js. This is just one of the many ways to implement JWT authentication and there are many other ways to improve this system based on the specific needs of your application.

# How to implement JWT Authentication Using Node, Express, TypeScript 2023

[[](https://medium.com/@cristain.333?source=post_page-----6263b689950f--------------------------------)](https://medium.com/@cristain.333?source=post_page-----6263b689950f--------------------------------)

[Sukanta Das](https://medium.com/@cristain.333?source=post_page-----6263b689950f--------------------------------)

·

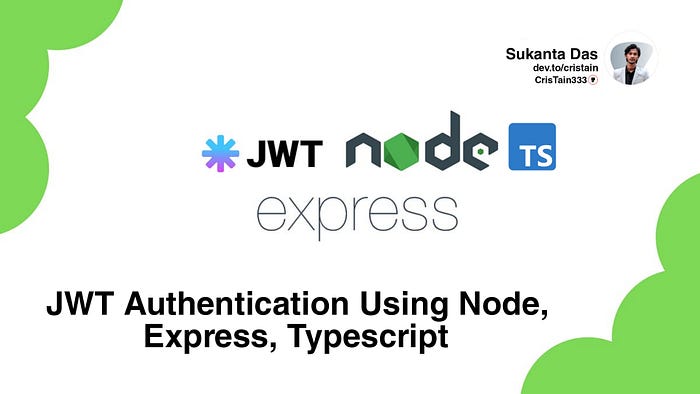
Follow

7 min read

·

Jul 25

2



Today We Will Learn How to Build NodeJs Authentication API using JWT, Express, and Typescript.

## Table of contents - What Is JSON Web Token (JWT) - Initialize Project - Install dependencies and devDependencies - Setup express server with typescript - Create a User Model - Create User Api To Register - Crate Login Api and Implement JWT Authentication

# 1. What Is JSON Web Token (JWT)

**JSON web token(JWT)** is JSON Object which is used to securely transfer information over the web(between two parties). It is generally used for authentication systems and can also be used for information exchange.

This is used to transfer data with encryption over the internet also these tokens can be more secured by using an additional signature.

# 2. Initialize Project

mkdir jwt-authentication  
cd jwt-authentication  
npm init - yes

# 3. Install dependencies and devDependencies

## 3.1 Install dependencies

npm install express mongoose cors jsonwebtoken dotenv

## 3.2 Install devDependencies

npm install -D typescript nodemon @types/express @types/cors @types/jsonwebtoken

## 3.3 Add a tsconfig.json for typescript configuration

tsc - init

add this config in tsconfig.json file

{  
 "compilerOptions": {  
 /\* Language and Environment \*/  
 "target": "es2016" /\* Set the JavaScript language version for emitted JavaScript and include compatible library declarations. \*/,  
 /\* Modules \*/  
 "module": "commonjs" /\* Specify what module code is generated. \*/,  
 "rootDir": "./",  
 "outDir": "./dist",  
 "esModuleInterop": true,  
 "forceConsistentCasingInFileNames": true /\* Ensure that casing is correct in imports. \*/,  
/\* Type Checking \*/  
 "strict": true,  
 "skipLibCheck": true /\* Skip type checking all .d.ts files. \*/  
 }  
}

# 4. Setup express server with typescript

In the root create a file name app.ts

import express from "express";  
import { Application } from "express";  
import mongoose from "mongoose";  
import cors from "cors";  
import dotenv from "dotenv";  
// Create the express app and import the type of app from express;  
const app: Application = express();  
// Cors  
app.use(cors());  
//configure env;  
dotenv.config();  
// Parser  
app.use(express.json());  
app.use(  
 express.urlencoded({  
 extended: true,  
 })  
);  
// Declare The PORT Like This  
const PORT: number = 8000;  
app.get("/", (req, res) => {  
 res.send("<h1>Welcome To JWT Authentication </h1>");  
});  
// Listen the server  
app.listen(PORT, async () => {  
 console.log(`🗄️ Server Fire on http:localhost//${PORT}`);  
// Connect To The Database  
 try {  
 await mongoose.connect(  
 process.env.DATABASE\_URL as string  
 );  
 console.log("🛢️ Connected To Database");  
 } catch (error) {  
 console.log("⚠️ Error to connect Database");  
 }  
});

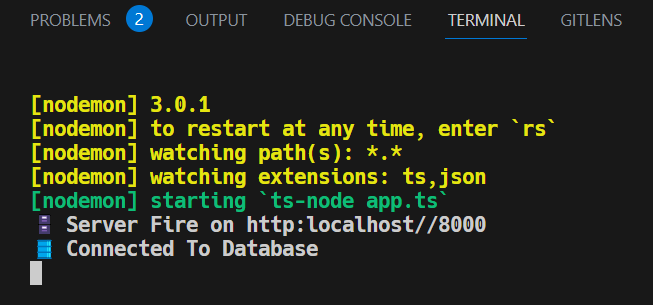
In the Package.json add this script

"scripts": {  
 "dev": "nodemon app.ts",  
 "test": "echo \"Error: no test specified\" && exit 1"  
 },

After Update the package.json file open the terminal and run the command

npm run dev

In the Terminal You will see



Open Your Browser and type the URL “[http://localhost:8000](http://localhost:8000/)"

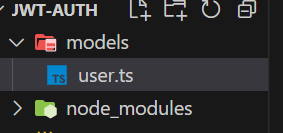


The Express server is now up and running!

# 5. Create a User Model

The User model is created using the Mongoose package. This model represents each user in the database.

In the root create a models folder, and create the file user.ts.



inside user.ts create the user schema;

import mongoose from "mongoose";  
const UserSchema = new mongoose.Schema(  
 {  
 name: {  
 type: String,  
 required: true,  
 },  
 email: {  
 type: String,  
 unique: true,  
 required: true,  
 },  
 password: {  
 type: String,  
 required: true,  
 },  
 },  
 { timestamps: true }  
);  
export const User = mongoose.model("Users", UserSchema);

After creating the user model we are ready to implement the jwt Authentication.

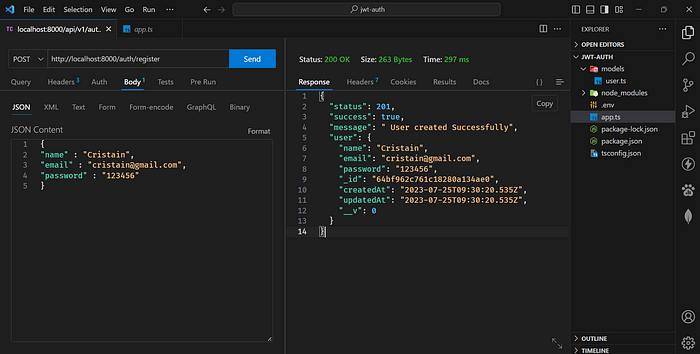
# 6. Create User Api To Register

in the “app.ts” create a register API to create an account.  
API = ‘/auth/register’

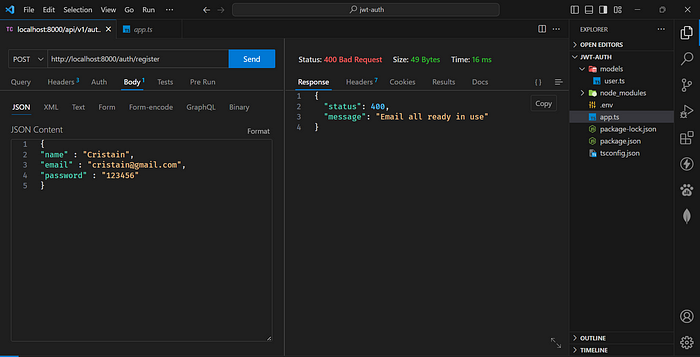
app.post("/auth/register", async (req, res) => {  
 try {  
 // \*\* Get The User Data From Body ;  
 const user = req.body;  
// \*\* destructure the information from user;  
 const { name, email, password } = user;  
// \*\* Check the email all ready exist in database or not ;  
 // \*\* Import the user model from "./models/user";  
const isEmailAllReadyExist = await User.findOne({  
 email: email,  
 });  
// \*\* Add a condition if the user exist we will send the response as email all ready exist  
 if (isEmailAllReadyExist) {  
 res.status(400).json({  
 status: 400,  
 message: "Email all ready in use",  
 });  
 return;  
 }  
// \*\* if not create a new user ;  
 // !! Don't save the password as plain text in db . I am saving just for demonstration.  
 // \*\* You can use bcrypt to hash the plain password.  
// now create the user;  
 const newUser = await User.create({  
 name,  
 email,  
 password,  
 });  
// Send the newUser as response;  
 res.status(200).json({  
 status: 201,  
 success: true,  
 message: " User created Successfully",  
 user: newUser,  
 });  
 } catch (error: any) {  
 // console the error to debug  
 console.log(error);  
// Send the error message to the client  
 res.status(400).json({  
 status: 400,  
 message: error.message.toString(),  
 });  
 }  
});

Now test the api. I am using ThunderClient Extension available in vs code Market Place.

if everything works well you will get a success message like this



now if I try to create the same user with the same email it will send me a error message.



after creating the user we can start jwt implementation.

# 7. Crate Login Api and Implement JWT Authentication

api = ‘/auth/login’

- Get The User Data From Body.  
- destructure the information from the user.  
- Check the (email/user) exist in the database or not.  
- if there is no user with the email we send user is not found.  
- if the (user) exists in the database we will check whether the password is valid or not.  
- compare the password in the database and the password in the request body.  
- if not matched send a response with the wrong password.  
- if the email and password are valid create a token.  
- To create a token JsonWebToken (JWT) receives 3 parameter  
1. Payload — This contains the claims or data you want to include in the token.  
2. Secret Key — A secure key known only to the server used for signing the token.  
3. expiration — Additional settings like token expiration or algorithm selection.

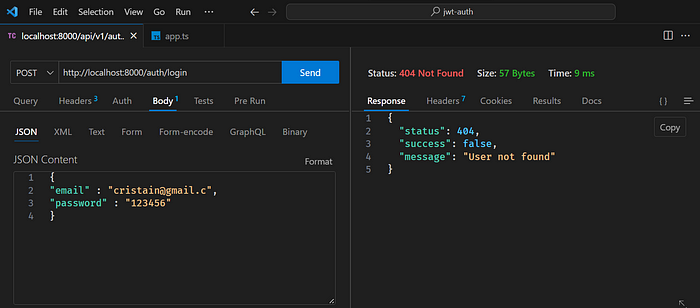
- Don’t Provide the secret openly, This secret is very sensitive for the server. please keep it in the .env file. I am Keeping it Open just for demonstration.

- After creating the token send the response.

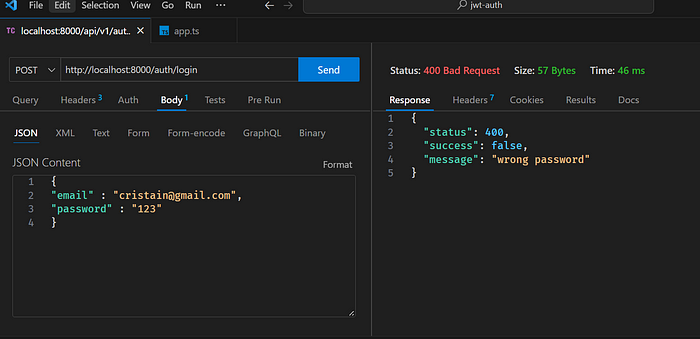
app.post("/auth/login", async (req, res) => {  
 try {  
 // \*\* Get The User Data From Body ;  
 const user = req.body;  
// \*\* destructure the information from user;  
 const { email, password } = user;  
// \*\* Check the (email/user) exist in database or not ;  
 const isUserExist = await User.findOne({  
 email: email,  
 });  
// \*\* if there is not any user we will send user not found;  
 if (!isUserExist) {  
 res.status(404).json({  
 status: 404,  
 success: false,  
 message: "User not found",  
 });  
return;  
 }  
// \*\* if the (user) exist in database we will check the password is valid or not ;  
 // \*\* compare the password in db and the password sended in the request body  
const isPasswordMatched = isUserExist?.password === password;  
// \*\* if not matched send response that wrong password;  
if (!isPasswordMatched) {  
 res.status(400).json({  
 status: 400,  
 success: false,  
 message: "wrong password",  
 });  
 return;  
 }  
// \*\* if the email and password is valid create a token  
/\*  
 To create a token JsonWebToken (JWT) receive's 3 parameter  
 1. Payload - This contains the claims or data you want to include in the token.  
 2. Secret Key - A secure key known only to the server used for signing the token.  
 3. expiration - Additional settings like token expiration or algorithm selection.  
 \*/  
// !! Don't Provide the secret openly, keep it in the .env file. I am Keeping Open just for demonstration  
// \*\* This is our JWT Token  
 const token = jwt.sign(  
 { \_id: isUserExist?.\_id, email: isUserExist?.email },  
 "YOUR\_SECRET",  
 {  
 expiresIn: "1d",  
 }  
 );  
// send the response  
 res.status(200).json({  
 status: 200,  
 success: true,  
 message: "login success",  
 token: token,  
 });  
 } catch (error: any) {  
 // Send the error message to the client  
 res.status(400).json({  
 status: 400,  
 message: error.message.toString(),  
 });  
 }  
});

Now test the API. I am using ThunderClient Extension available in vs code Market Place.

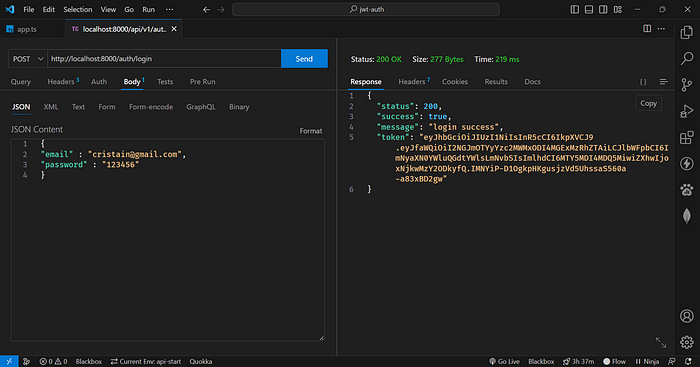
Now If Try To Login With Wrong Email that it will send the error User not found



Now If Try To Login With the right Email but with the wrong password it will send an error of the wrong password.

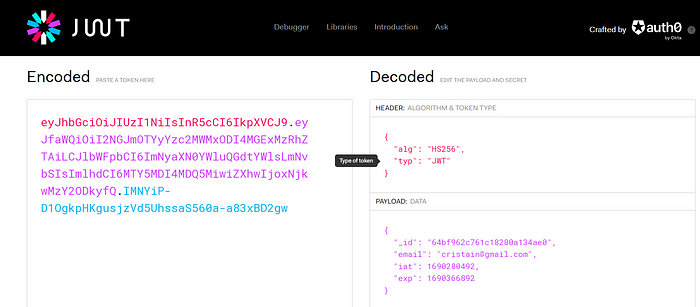


If I Provide The Correct Email and Password it will send me the token that we created.



Now if we copy the token that we got after the login and go to <https://jwt.io/> and past the token and press decode

we can see the information that we provided on creation time.



![json\_decode](<https://dev-to-uploads.s3.amazonaws.com/uploads/articles/bgn5eydygqqsydp3t6bd.png>)

If You Found This Help full, Give a star ⭐  
Share With Other’s as well.

Github Repo: <https://github.com/CrisTain333/jwt-authentication-node-express-typescript>

# That’s all.

**Sign up and Login Implementation with JWT in Node.js using Postgres and Sequelize**

[[](https://medium.com/@kizito917?source=post_page-----7093491b080f--------------------------------)](https://medium.com/@kizito917?source=post_page-----7093491b080f--------------------------------)

[amaechi kingsley](https://medium.com/@kizito917?source=post_page-----7093491b080f--------------------------------)

·

Follow

12 min read

·

Jun 23

11

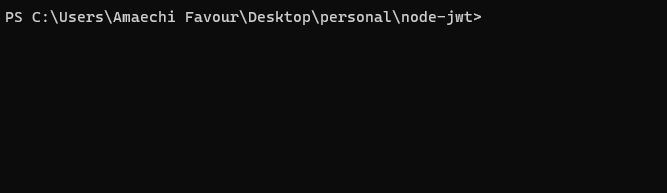


Software developers must create well-secured backend apps and APIs for whatever type of application they are working on because security concerns in backend/API development are growing constantly.  
This article will walk readers through each step of creating a Node.js **REST API** that uses **JWT**authentication. we will be implementing this feature using Sequelize ORM, Express, Node.js, and PostgreSQL. We’ll create a login and signup process utilizing JSON web token (jwt). To briefly depart from the subject, let’s define **token-based authentication**. It is a method for storing a token (such as a JSON web token) on the client side (which may be kept in the browser’s local storage, an iOS app’s keychain, or an Android app’s shared preference).

**Creating a Node.js app**

We will start by creating a Node.js application. It is expected that you are conversant with [Node.js](https://nodejs.org/) and [Npm](https://www.npmjs.com/" \t "_blank) and also have it installed on your computer.

The first step will be creating a directory in any area of your computer (always best to be in an area you can easily access through the terminal). We would create a folder called “**node-jwt**”. After creating the folder, open the folder in a terminal. You should have a similar output as the image below:



The next step will be initializing the project which will auto-generate a package.json file. This is done using the syntax “**npm init -y**”. This syntax generates a package.json file in your project directory and the package.json file is like a configuration file for your application. After the successful running of the command, you should get a similar result below:



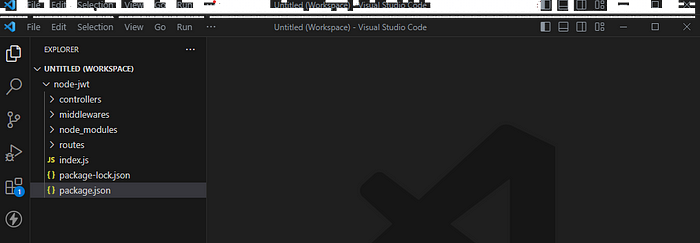
package.json generated content

Now the project structure of your application will contain a “**package.json**” file. The next step will be to create an entry file for your server/backend application. We will create a new file named **index.js** in the project root directory. Also, the following folders will be created in the root directory:

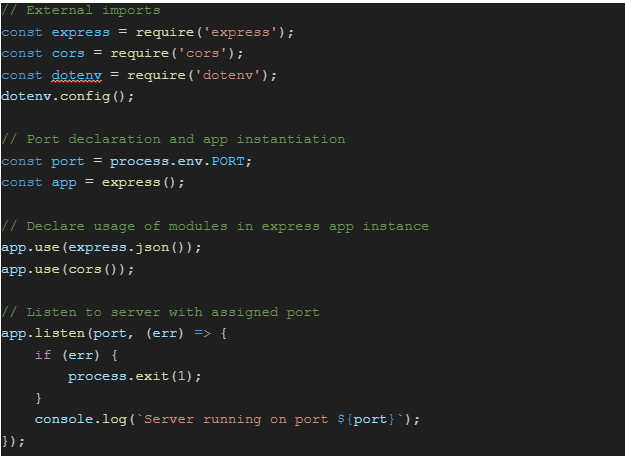
1. **Controllers**— The controller’s folder contains all logic and functions related to several routes. All database calls, custom functions, etc. are placed in files located inside the controller.
2. **Routes**— The route folder hosts several entry points to each specific function. For example, the endpoint for registration, login, profile fetch, etc. is all placed inside of the route.
3. **Middleware**- The middleware folder holds all functions that are to run before the controllers (immediately after the endpoint is triggered).

After creating the required files and folders, open up the project directory in a code editor like [Visual Studio](https://code.visualstudio.com/).

The next step will be installing packages required for the development of the Rest APIs. We will be installing the following packages for a start ( as dependencies on the project): **express**, **sequelize**, **cors**, **dotenv**, **bcryptjs**, and **jsonwebtoken**. To install the listed packages, we will run the command “**npm install express sequelize cors dotenv bcryptjs jsonwebtoken — save**”. After successful installation of the dependencies, a **node\_modules** folder will be created which contains all the installed dependencies and all other related dependencies. The project structure will now look like this:



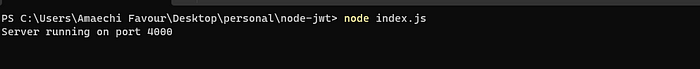
Now, Let’s get our hands dirty with some codes. As seen in the package.json, the entry file for our application will be the index.js file. This will be created in the root directory of the project and will have the following code in it for a start:



This code above requires the express package which was installed earlier and also requires other installed packages in the application like the cors and dotenv packages. Then we go further to specify the port in which the application will run and also create the express instance. We also make use of the **app.use()** method to tell our application to use express in json format and also allow the use of cors to prevent CORS errors from browsers. We finally use the **app.listen()** method to listen to our application on the specified port. Before going ahead to start our application, we need to create a “**.env”** file in the root directory of our application and paste the following code in it:

PORT=4000

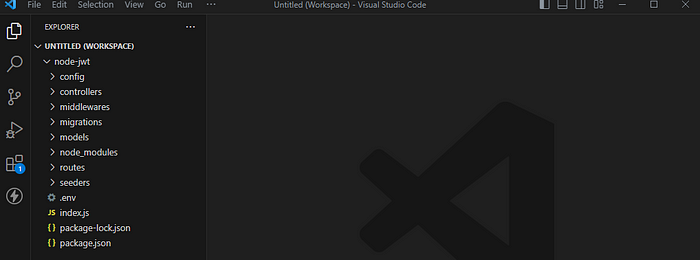
This is used to define the port number that will be used to serve the application once it has been launched. Now we can launch our application with the command “**node index.js**” and see the following message in our terminal:



You have one more step to complete before you can call yourself a pro (haha). If you remember, we installed the sequelize package. Now we need to set up the use of Postgres and sequelize in our application. For the sequelize setup, we also need to install two extra packages. Postgres will be installed as a dependency, along with the sequelize-cli (which will be deployed as a devDependency). We will then execute each of the commands below to install them:

* **npm install –save-dev sequelize-cli**
* **npm install –save pg**

If the aforementioned commands are installed successfully, the package will display them as installed packages in the package.json file and now our application/terminal will allow us to use sequelize commands. we can now run the command “**npx sequelize-cli init**” to set up sequelize in our application alongside Postgres. This will automatically create a few directories and files needed for Postgres & sequelize setup. Your project directory ought to now appear as follows:



Just a brief explanation, You will notice some new folders have been added to the project, Like the config, migrations, models and seeders.

1. **Config**— This folder holds the file that contains all the information needed to set up the database (Database host, password, name, username, etc.) in different environments like development, production and staging
2. **Migrations**— The migration folder contains all created migration files that are used to make changes to the database tables, columns, attributes, etc (e.g. a migration file can contain code used to create a new table in the database).
3. **Models** — The model’s folder consists of files that depict the schema for each table created in the database.

**Setup the Database**

Now we will start by setting up the database. Open the config folder and you will see the config.json file, rename it to config.js and paste the following code in it:

const dotenv = require(‘dotenv’);  
dotenv.config();  
  
  
module.exports = {  
 development: {  
 username: process.env.PGUSER,  
 password: process.env.PGPASSWORD,  
 database: process.env.PGDATABASE,  
 host: process.env.PGHOST,  
 dialect: "postgres",  
 dialectOptions: {  
 ssl: true,  
 },  
 },  
 production: {  
 username: process.env.PGUSER,  
 password: process.env.PGPASSWORD,  
 database: process.env.PGDATABASE,  
 host: process.env.PGHOST,  
 dialect: "postgres",  
 ssl: 'require'  
 }  
}

You will notice that the values are coming from the **.env** file which is the reason we are using **process.env** syntax. This is the best approach so as not to make our credentials visible to the public during deployment or commit to github or any other version control system. Now open the **.env** file and add the following credentials (replace the values with whatever values you’ve got in your Postgres Database on your computer or cloud):

PGHOST='localhost'  
PGDATABASE='node-jwt'  
PGUSER='postgres'  
PGPASSWORD='postgres'

Having done that, open the model’s folder and locate the index.js file inside the model’s folder. Look out for anywhere you have config.json and replace it with config.js (remember we had to rename our config.json to config.js to be able to make use of the “**process.env**” syntax).

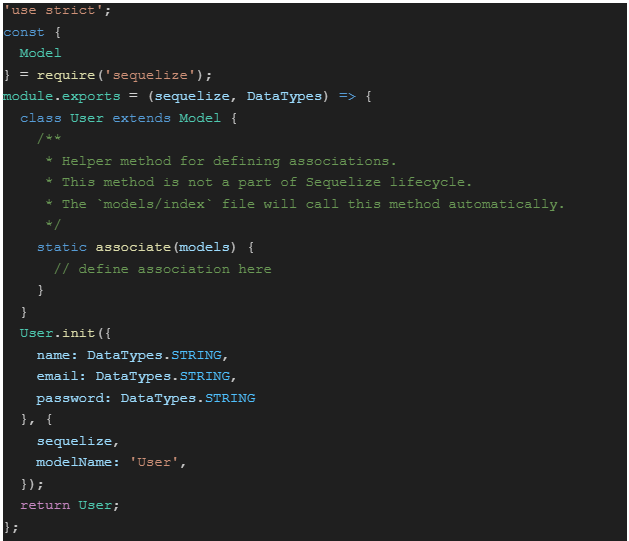
And yes!!! We have our database setup complete. We can start creating our database tables using sequelize and running the migrations. Let’s relax on the code writing and look into the next step we want to achieve. To bring in the use of jsonwebtoken (JWT), we need to implement a sign-up and login system which will allow us to implement the use of jwt. For the sake of this tutorial, we will be building two different endpoints. The first one is for user signup and the second one is for user login. We will require the user to sign up/register using the following information:

* Name
* Email
* Password

This means we will have to create a user table in our database and the table will contain the attributes (name, email, password). Once we are done creating the tables, we will then write the controller and route setup to trigger the API call for sign-up and login and then we can say: “**Here comes the end of our implementation.**” Now let’s dive into some codes. We will create a user table with the said information as explained above and to do that, we have to use the sequelize syntax for creating models and migrations. The table will be created by running the migration file and the migration file will then go into the Database and create the table. The syntax for creating the model and migration is:

“**npx sequelize-cli model:generate — name User — attributes name:string,email:string,password:string**”

This will successfully produce a migration file in the migrations folder as well as a model file called user.js (in the model’s folder). You should have code that is similar to or the same as the following in the model’s newly formed user.js file:



And in the created migrations file, you should have something similar to this code below:



This is not our primary concern, so we won’t go into detail about what each of these codes above performs. You can look at the [sequelize official documentation](https://sequelize.org/" \t "_blank) to have a better idea of what the models and migrations file content does. Now, we can go ahead and run the migration and we should have our table created in the just-created database. The syntax for running migrations in a Node.js sequelize/Postgres project is:

**npx sequelize-cli db:migrate**

This will run in the terminal and will create a table called **Users**with all the columns which are listed in the migration file. Now we have a table of Users and can now write the logic/route to be used in registering users and also for the login of users.

**Setting up the Controllers**

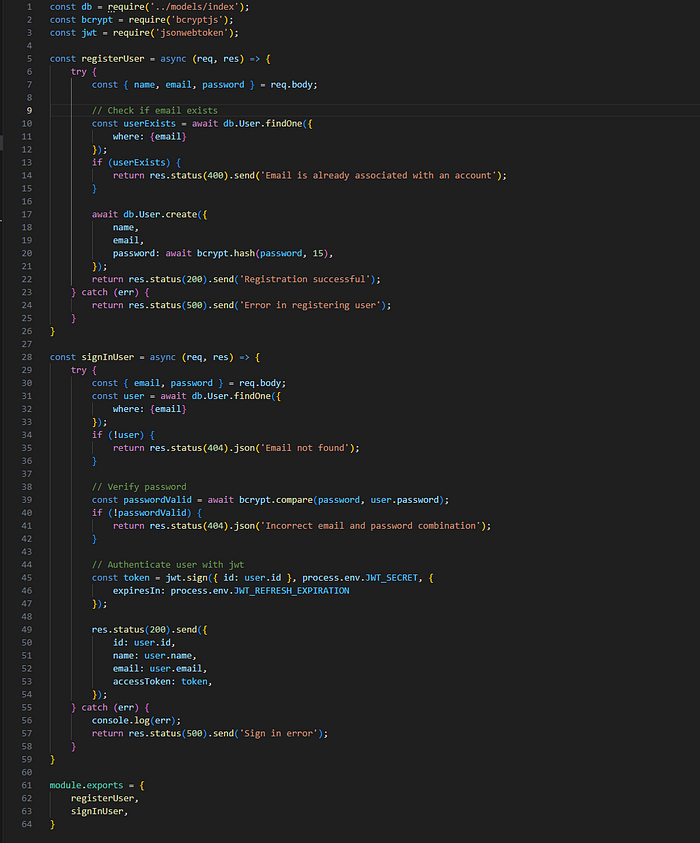
To set up the controllers as described earlier, we will create a file inside the controller’s folder called “**auth.controller.js**” by opening the folder for the controller. All the codes required for sign-up and sign-in will be in this file. The code below will be the file’s content:

Const db = require(‘../models/index’);  
const bcrypt = require('bcryptjs');  
const jwt = require('jsonwebtoken');  
  
  
const registerUser = async (req, res) => {  
 try {  
 const { name, email, password } = req.body;  
 // Check if the email exists  
 const userExists = await db.User.findOne({  
 where: {email}  
 });  
 if (userExists) {  
 return res.status(400).send('Email is already associated with an account');  
 }  
  
  
 await db.User.create({  
 name,  
 email,  
 password: await bcrypt.hash(password, 15),  
 });  
 return res.status(200).send('Registration successful');  
 } catch (err) {  
 return res.status(500).send('Error in registering user');  
 }  
}

A new user can be added to our database using the **registerUser**method we developed. To prevent several people from using the same email, we first check to see whether any other users have the same email. Also, since it’s unsafe to enter a plain password text into a database, we go ahead to hash our password using the **[bcrypt](https://www.npmjs.com/package/bcryptjs" \t "_blank)**package which was installed earlier. We then proceed to insert the new user’s information into the database using the sequelize **create()** method. Having taken care of that, the sign-in controller will be the next function we add to the controller:

const signInUser = async (req, res) => {  
 try {  
 const { email, password } = req.body;  
 const user = await db.User.findOne({  
 where: {email}  
 });  
 if (!user) {  
 return res.status(404).json('Email not found');  
 }  
  
  
 // Verify password  
 const passwordValid = await bcrypt.compare(password, user.password);  
 if (!passwordValid) {  
 return res.status(404).json('Incorrect email and password combination');  
 }  
  
  
 // Authenticate user with jwt  
 const token = jwt.sign({ id: user.id }, process.env.JWT\_SECRET, {  
 expiresIn: process.env.JWT\_REFRESH\_EXPIRATION  
 });  
   
 res.status(200).send({  
 id: user.id,  
 name: user.name,  
 email: user.email,  
 accessToken: token,  
 });  
 } catch (err) {  
 return res.status(500).send('Sign in error');  
 }  
}

In the **signInUser**function, we are checking if a user email exists and if it does, we further compare the password provided by the user with what exists in the database using the **bcrypt.hash()** method. After comparing and the password is valid, we then implement the jsonwebtoken signing for the user which will generate a token that the user will use in accessing other protected routes in the application. The token is sent to the client side and the application only understands the token and uses it to identify whichever user it is.  
Having written the controller for both signup and sign-in, your auth.controller.js file should be looking like this:



**Setting up the Routes**

Now, we have the controller written and ready to connect to the routes. The next step will be creating the routes and keying them to the just-created controllers. Open the routes folder in the project directory and create an auth.route.js file inside the folder. The created route file (**auth.route.js**) will be used to identify which controller gets called when a particular route is called. Go ahead and past the following code into the route:

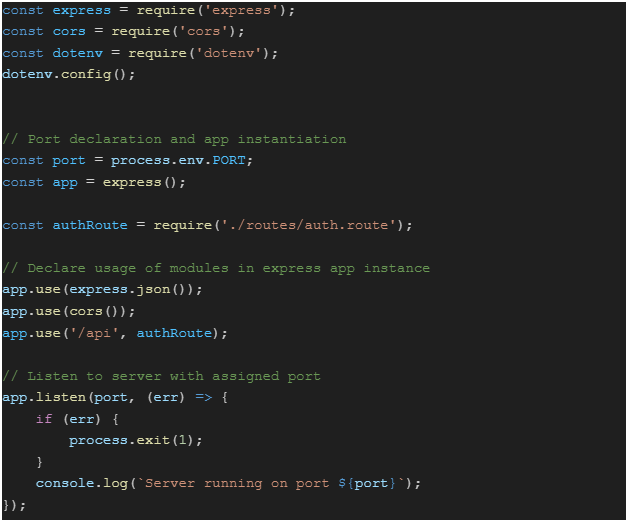
const express = require('express');  
const { registerUser, signInUser} = require('../controllers/auth.controller');  
const router = express.Router();  
  
// Registration route  
router.post('/sign-up', registerUser);  
  
  
// Signin route  
router.post('/sign-in', signInUser);  
  
module.exports = router;

The code above imports the express package and sets up a router instance. It also imports the controllers from the “**auth.controller.js**” file and we further declare the registration route as **router.post(‘/sign-up’)**. This means when our Backend applications make a POST request to <http://localhost:4000/api/sign-up>, the registerUser controller gets executed likewise that of the **router.post(‘sign-in’)** route also. Now we can go ahead and import this route file into our application entry file (index.js)

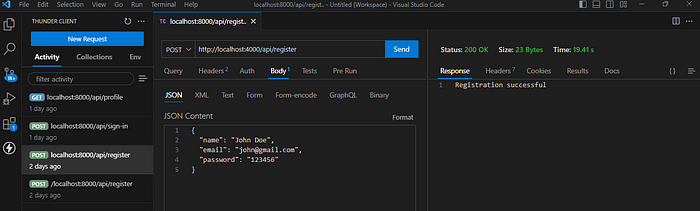
If you came this far, Kudos to you…you are a legend!!!

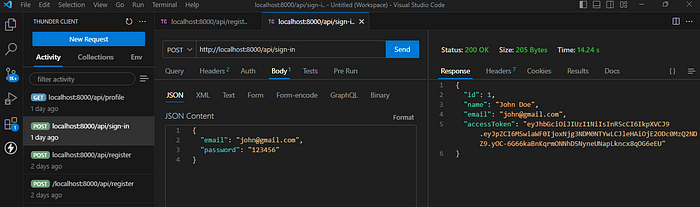
**Final Step**

To couple our application together, we will need to import the auth.route.js file into the entry file (index.js). Now your entry file should look like this:



Everything is in place at this moment, so we can launch our application and test the endpoints by running the “**node index.js**” command again to launch our application. Now that our application is properly connected, we can test our endpoints using any third-party programs, such as Postman, Thunder Client, insomnia, etc. We will test our endpoints using the Thunder client extension in Visual Studio Code for purposes of learning. If tested carefully using Thunder Client, you should have a similar result to the images below:





You would notice the **jwt token** generated in the sign-in endpoint. The generated token is used to make requests to other endpoints in the application where the token will be passed in the request header while making the requests. This token is what makes the server understand that **John Doe** with **user id 1** is making a request.

**Conclusion**

With our implementation, we can say we’ve successfully implemented a Backend API with Node.js, Postgres with Sequelize ORM and JSON WEB TOKEN for authentication. It doesn’t end here in a real-life application because the token generated is being used to make request to other protected endpoints in the server and it also has a validity time. Once its validity expires and the user is still in the application, the user should not be logged out of the application and this is where the implementation of a refresh token comes in.

Did you enjoy the detailed article? Kindly visit the [Part 2 of Node.js API implementation with Postgres, Sequelize and JWT which covers the middleware implementation and also implement the use of a refresh token in our application](https://medium.com/@kizito917/jwt-refresh-token-implementation-with-node-js-postgres-and-sequelize-106ef6b3de68). Cheers…

**Protect your REST APi’s in Node js**

[[](https://medium.com/@developerom?source=post_page-----253c11a2dfd7--------------------------------)](https://medium.com/@developerom?source=post_page-----253c11a2dfd7--------------------------------)

[Android & Node js](https://medium.com/@developerom?source=post_page-----253c11a2dfd7--------------------------------)

·

Follow

3 min read

·

Jun 27

31

To protect an API from hackers in a Node.js Express application, you can implement several security measures. Here’s an example of how you can secure your API using common security practices:

**Use HTTPS**: Enable secure communication by using HTTPS instead of HTTP. This ensures that the data transmitted between the client and server is encrypted. You can use the https module in Node.js or set up a reverse proxy like Nginx to handle SSL/TLS termination.

**Input validation and sanitization:** Validate and sanitize all user input to prevent common security vulnerabilities such as SQL injection, cross-site scripting (XSS), and command injection. You can use libraries like Joi or express-validator for input validation.

const { body, validationResult } = require('express-validator');  
  
app.post('/api/users', [  
 body('username').isAlphanumeric().trim().escape(),  
 body('password').isLength({ min: 8 }).escape(),  
], (req, res) => {  
 const errors = validationResult(req);  
 if (!errors.isEmpty()) {  
 return res.status(400).json({ errors: errors.array() });  
 }  
 // Process the request  
});

**Authentication:** Implement a robust authentication mechanism to verify the identity of the clients accessing your API. You can use strategies like JSON Web Tokens (JWT) or session-based authentication. Libraries like Passport.js can simplify the authentication process.

const passport = require('passport');  
const jwt = require('jsonwebtoken');  
  
// Login route  
app.post('/api/login', (req, res, next) => {  
 passport.authenticate('local', { session: false }, (err, user, info) => {  
 if (err || !user) {  
 return res.status(401).json({ message: 'Authentication failed' });  
 }  
 // Generate and send JWT token  
 const token = jwt.sign({ id: user.id }, 'secret-key');  
 res.json({ token });  
 })(req, res, next);  
});  
  
// Protected route  
app.get('/api/profile', passport.authenticate('jwt', { session: false }), (req, res) => {  
 // Access only allowed for authenticated users  
 res.json({ message: 'Authenticated user' });  
});

**Authorization:** Control access to different parts of your API based on user roles and permissions. Ensure that each request to protected endpoints is authorized to perform the requested action. You can implement role-based access control (RBAC) or create custom authorization middleware.

// Authorization middleware  
const authorize = (roles) => {  
 return (req, res, next) => {  
 if (roles.includes(req.user.role)) {  
 next();  
 } else {  
 res.status(403).json({ message: 'Forbidden' });  
 }  
 };  
};  
  
// Protected route with authorization  
app.get('/api/admin', passport.authenticate('jwt', { session: false }), authorize(['admin']), (req, res) => {  
 // Access only allowed for users with 'admin' role  
 res.json({ message: 'Admin access granted' });  
});

**Rate limiting:** Implement rate limiting to prevent abuse and brute force attacks on your API. Libraries like express-rate-limit can help you set limits on the number of requests per IP address or user.

const rateLimit = require('express-rate-limit');  
  
// Apply rate limiting middleware  
const apiLimiter = rateLimit({  
 windowMs: 15 \* 60 \* 1000, // 15 minutes  
 max: 100, // Limit each IP to 100 requests per windowMs  
});  
app.use('/api/', apiLimiter);

**Error handling:** Implement proper error handling to prevent sensitive information from being exposed in error responses. Use generic error messages and log errors without revealing sensitive details.

app.use((err, req, res, next) => {  
 console.error(err);  
 res.status(500).json({ error: 'Internal server error' });  
});

hese are some of the essential security measures you can take to protect your Node.js Express API from hackers. It’s important to note that security is an ongoing process, and you should regularly update your dependencies and stay informed about new vulnerabilities to ensure the continued security of your application.

Happy Coding!