**NodeJs authentication using jwt and refresh token and sequelize and Postgres from scratch**

Image for post



Introduction

Authentication is very important in building a secure system. To a beginner, authentication and authorization are two confusing words. Authentication is simply a verification of user’s claims while Authorization is simply access control(i.e who should access a given resource). JSON web token (aka jwt) is a spec that provides a way that token must be represented. A token is a piece of random strings that contain decodable information about a user.

Note: jwt is not encrypted but it’s base64 encoded what is the implementation of that, anybody that has access to jwt can easily decode it, hence we should not store sensitive information inside jwt. Common examples of a token are access token and refresh token.  
**Access token**carries the information required to access a protected resource directly. they usually have an expiration and are also short-lived. the resource server uses the access token to determine if the client is authorized or not.

**Refresh token** carries the information necessary to get a new access token. the client uses the refresh token to get a new token issued by the auth server maybe the access token has expired. Refresh token can also expire but they are rather long-lived. it’s worth noting that refresh token must be stored securely, they can also be blacklisted by the authorization server. if a refresh token is leaked it maybe be used to obtain a new access token(and therefore access protected resources) until it’s either blacklisted or it expires.

***Why are do we need refresh token in our application?***Imagine asking a user to enter login credentials every time their access token get expired.

Image for post



what the fuck! login again

to give a user a better experience, we auto-generate new access token using their refresh token that hasn’t expired without requiring them to log in them.

let get straight into the code.

**prerequisite:  
1.**Basic javascript knowledge  
2. node installed. <https://nodejs.org/en/download/>  
3. PostgreSQL database. <https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>

4. Basic knowledge of Sequelize.

To get started, create a folder and run

npm init -y

The command generates a package.json for you. the package.json file contains all the necessary information about your project and commands to run the project.

I will go ahead to create a folder structure for my project. Inside your root directory type the command below

mkdir src (this create a source folder for our application)  
cd src  
touch index.js ( this create the entry point for our application)

still on the src directory type

mkdir controllers services routes libs routes middlewares

this folder structure will help write an organized code.  
Next, let install the necessary dependencies.

we are going to be writing our code using ES6 and ESNext syntax, so we will use babel to transpile our to code to ES5 syntax. How do we do that?  
inside your root directory,

touch .babelrc  
npm install @babel/core @babel/node @babel/cli @babel/preset-env -D

the -D flag at the end of the snippet above means dev dependency

paste the snippet below into the .babelrc file created above

{  
 "presets": ["@babel/preset-env"]  
}

let spinner up our server, we going to use express as our framework. To get your hand dirty on express your need to install express

npm install express

I like logging requests made to and endpoints, to do that will install morgan and nodemon (keep eye on our file and restart the server if is any change in our file)

npm install morgan  
npm install nodemon -D

copy the snippet below into the index.js you created previously

import express from "express";  
import morgan from "morgan";const app = express();   
app.use(morgan("dev")); // line 5const PORT = process.env.PORT || 7000; //line 7app.listen(PORT, () => {  
 console.log(`server listen at port ${PORT}`);  
});

At line 5 we are passing morgan as middleware. A middleware is a function that has access to the request and response object. In express, all middleware starts with ***app.use()***, and the order of the placement is very important to the request-response lifecycle. we will see more of this later.  
At line 7 we telling express to use default port 7000(it could be any number) if it does not see any in port in the environment. From the snippet above express would use Port as 7000 because of process.env. port is undefined.  
The question is how do make express to get the value of process.env.port?  
let’s do that.

on your root directory run

touch .env

the .env file is a place your store your environment variable, variable that is sensitive to your application, and wouldn’t like it to get to wrong hands.  
create a gitignore file and add the .env and node\_modulus inside it, doing this allows you not to commit those file into your git repository

touch .gitignore

paste the code into the gitignore

.env  
node\_modulus/

Replace the index.js with the code below

import express from "express";  
import morgan from "morgan";  
import dotenv from "dotenv";const app = express();  
dotenv.config();app.use(morgan("dev"));  
const PORT = process.env.PORT || 7000;app.listen(PORT, () => {  
console.log(`server listen at port ${PORT}`);  
});

To start our server we need to make some changes to our packages.json file. Replace the test command with the snippet below:

"start:dev": "nodemon --exec babel-node ./src/index.js"

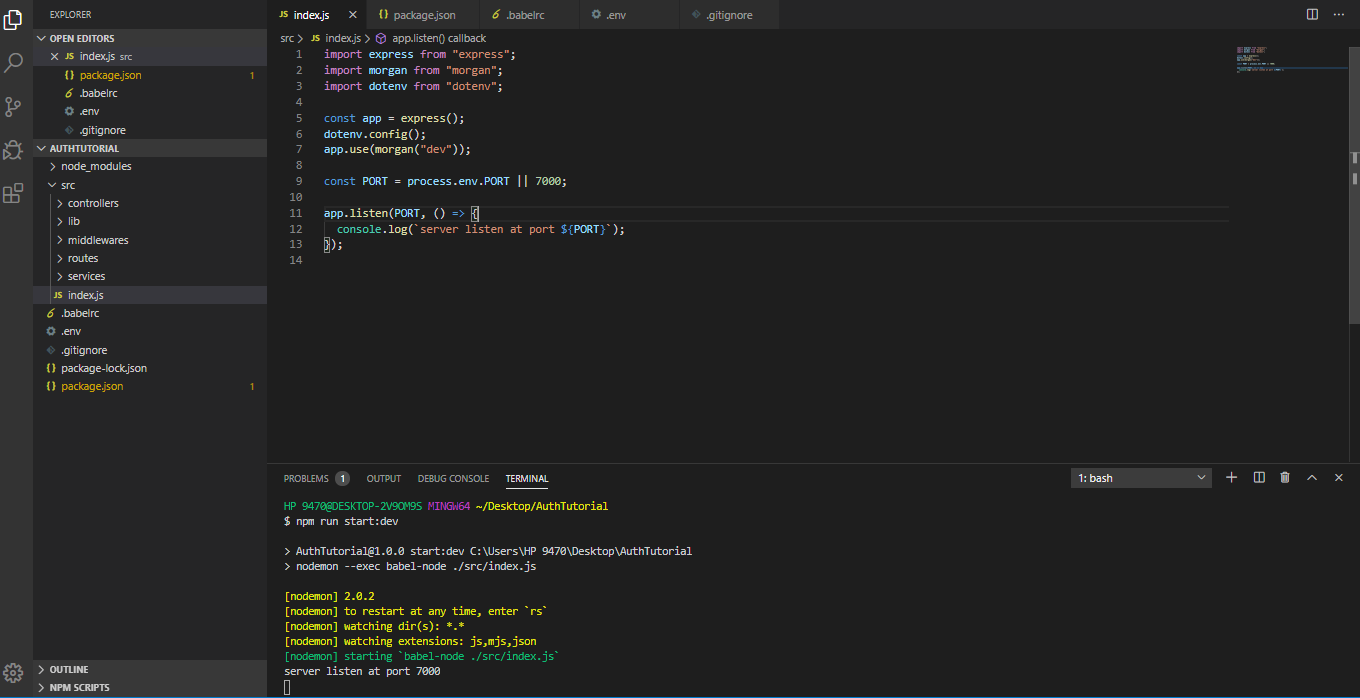
the snippet above tells babel to transpile our code down to ES5.

finally, spin up the server by running the command on your command line

npm run start:dev

if you made it to this point you should see this on your editor

Image for post



**Setting up Sequelize and Postgres:**

Sequelize is a promise-based Node.js ORM for Postgres, MySQL, MariaDB, SQLite, and Microsoft SQL Server. let install the necessary libraries. ORM allows us to interact with our database in a more friendly way, by using object representation. that means we don't necessarily need to write SQL, instead we can directly use object methods.

npm install sequelize-cli --globalnpm install sequelize pg pg-hstore

we are installing Postgres(pg), pg-hstore( is a node package for serializing and deserializing JSON data to hstore format), Sequelize(main library), sequelize-cli( is a package that enables us to interact with the database through sequelize from the CLI).

create a .sequelizerc file at the root directory

touch .sequelizerc  
npm install @babel/register

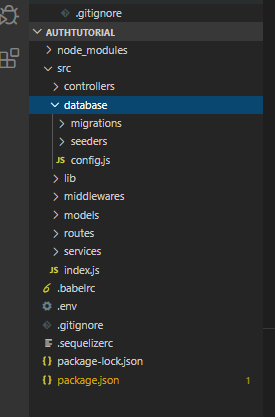
paste the code snippet below inside it. the snippet is telling sequelize to scaffold the necessary files required to interact with a database using the specified path.

require('@babel/register')  
const path = require('path');  
module.exports = {  
"config": path.resolve('./src/database', 'config.js'),  
"models-path": path.resolve('./src/models'),  
"seeders-path": path.resolve('./src/database/seeders'),  
"migrations-path": path.resolve('./src/database/migrations')  
};

To implement the scaffolding run

sequelize init

Image for post



Replace the content of the database/config.js with the snippet below

import dotenv from "dotenv";dotenv.config();  
module.exports = {  
 development: {  
 use\_env\_variable: true,  
 url: process.env.DEV\_DATABASE\_URL,  
 dialect: 'postgres'  
},test: {  
use\_env\_variable: true,  
url: process.env.TEST\_DATABASE\_URL,  
dialect: 'postgres'  
},production: {  
use\_env\_variable: true,  
url: process.env.PROD\_DATABASE\_URL,  
dialect: 'postgres'  
}  
};

the dialect represent the database driver(vendor) we should. here we are using Postgres.

Replace the content of the model/index.js with the snippet below

import fs from 'fs';  
import path from 'path';  
import Sequelize from 'sequelize';import { NODE\_ENV } from '../config/constant';  
import configs from '../database/config';const basename = path.basename(\_\_filename);  
const env = NODE\_ENV || 'development';  
const config = configs[env];  
const db = {};  
const sequelize = config.use\_env\_variable  
? new Sequelize(config.url, config)  
: new Sequelize(config.database, config.username, config.password, config);fs.readdirSync(\_\_dirname).filter(  
file =>// eslint-disable-next-line implicit-arrow-linebreakfile.indexOf('.') !== 0 && file !== basename && file.slice(-3) === '.js').forEach(file => {const model = sequelize.import(path.join(\_\_dirname, file));db[model.name] = model;});Object.keys(db).forEach(modelName => {if (db[modelName].associate) {  
db[modelName].associate(db);}});db.sequelize = sequelize;  
db.Sequelize = Sequelize;  
db.Op = Sequelize.Op;export default db;

the snippet above is dynamically configuring our model, and allow us to interact with a database using the specified connection string.

create a connection string using the format below

postgres://<dbUsername>:<dbPassword>@127.0.0.1:5432/<dbName>

You can create a database using the Postgres GUI or from the command line.

paste connection string inside your .env file

DEV\_DATABASE\_URL=postgres://<dbUsername>:<dbPassword>@127.0.0.1:5432/<dbName\_dev>TEST\_DATABASE\_URL=postgres://<dbUsername>:<dbPassword>@127.0.0.1:5432/<dbName\_test>PROD\_DATABASE\_URL=postgres://<dbUsername>:<dbPassword>@127.0.0.1:5432/<dbName\_prod>NODE\_ENV=development

lets creates our file migration and model file.  
A migration file is a schema that allows us to create our database tables while the model allows us to interact with our database.

let create our first model:

sequelize model:generate --name Account --attributes firstName:string,email:string,lastName:string

Note: ensure you don’t leave out space between attributes to avoid throwing up error.

Replace the content of the created migration with the snippet below:

import uuid from 'uuid/v4';module.exports = {  
up: (queryInterface, Sequelize) =>queryInterface.createTable('Accounts', {id: {allowNull: false,primaryKey: true,type: Sequelize.UUID,defaultValue: uuid()},firstName: {type: Sequelize.STRING},password: {type: Sequelize.STRING},email: {type: Sequelize.STRING,unique: true},lastName: {type: Sequelize.STRING},verified: {type: Sequelize.BOOLEAN,defaultValue: false},blocked: {type: Sequelize.BOOLEAN,defaultValue: false},changedPassword: {type: Sequelize.STRING,allowNull: true},createdAt: {allowNull: false,type: Sequelize.DATE,defaultValue: new Date()},updatedAt: {allowNull: false,type: Sequelize.DATE,defaultValue: new Date()}}),down: queryInterface => queryInterface.dropTable('Accounts')};

and also the content of the model/account with the code below:

import uuid from 'uuid/v4';module.exports = (sequelize, DataTypes) => {const Account = sequelize.define(  
'Account',{id: {allowNull: false,primaryKey: true,type: DataTypes.UUID,defaultValue: uuid()},firstName: {type: DataTypes.STRING,allowNull: false},lastName: {type: DataTypes.STRING,allowNull: false},email: {type: DataTypes.STRING,allowNull: false},password: {type: DataTypes.STRING,allowNull: false},verified: {type: DataTypes.BOOLEAN,allowNull: false,defaultValue: false},blocked: {type: DataTypes.BOOLEAN,allowNull: false,defaultValue: false},changedPassword: {type: DataTypes.STRING,allowNull: true}},{});Account.associate = () => {// associations can be defined here};return Account;};

Observation: the migration and model generated are virtually the same. you cannot interact with any field omitted in the model file. Example omitting firstName field whilst it appears in the corresponding migration file.

Note: generating a migration file does not automatically create the table in the database, to create the table run

sequelize db:migrate

kudos to you for making it to this point.

lets now create our endpoint, starting with the registration endpoint,

Firstly we need to install the outstanding library and also create some helper functions.

npm install bcryptjs jsonwebtoken cookie-parse lodash

* bcryptjs for encrypt our password
* jsonwebtoken for generate token for authorization and authentication
* cookie-parser for reading and setting cookie
* lodash for objects manipulations

Navigate to the lib folder and create passwordOp.js

touch passwordOp.js //to create the file

paste the code snippet below:

import bcrypt from 'bcryptjs';  
const SALT\_HASH\_KEY = 11;  
export const hashPassword = password => bcrypt.hash(password, SALT\_HASH\_KEY);

The SALT\_HASH\_KEY is the CPU intensive factor require to hash a password. The snippet above accepts a plain user password and returns hash password.

inside the routes folder, create authRoute.js and index.js and paste the snippet below:

// authRoute.js  
import { Router } from 'express';import { signupController } from '../controllers/authController';const authRouter = Router();authRouter.post('/signup',signupController);export default authRouter;

Navigate to your controller's folder and create authController.js and paste the snippet below::

export const signupController = async (req, res, next) => {await signupService(req, res, next);};

Navigate to services folder and create authService.js and index.js  
services/authService.js

import Model from "../models"; // this is the Account model we generated  
import { findOrCreate } from "./index";  
import { hashPassword } from "../lib/passwordOp";const { Account } = Model;export const signupService = async (req, res, next) => {  
try {  
const password = await hashPassword(req.body.password);  
const email = req.body.email.toLowerCase();  
const [account, created] = await findOrCreate(Account, {  
...req.body,  
password,  
email  
});  
if (!created) {  
return res.status(400).json({  
status: "fail,  
message: "user already exist"  
});  
}  
return res.status(201).json({  
status: "success",  
message: "user successfully created",  
payload: account.toJSON()  
});  
} catch (err) {  
return res.status(500).json({  
status: "error",  
message: "something went wrong"  
});  
}  
};

paste the snippet inside services/index.js

export const findOrCreate = (model, payload) =>  
model.findOrCreate({  
where: { email: payload.email },  
defaults: {...payload }  
});

this service structure allows separation of concerns, if tomorrow we want to change to database driver from Postgres to another vendor, we just change it in a single file. findorCreate is Postgres API that finds a record if it does not exist it will create it.  
Looking at code in the services/authService.js you will observe that use try-catch block if we have multiple services files, it means we will have multiple try-catch block in each file. I think it will make sense to have one generic file that handles the all the try-catch block without repeating ourself(remember DRY principle)  
how do we do that?  
**Implementing Global try-catch block**

Navigate to the lib directory and createcatchAsync.json the terminal paste the code snippet below:

const catchAsync = fn => (req, res, next) => {  
fn(req, res, next).catch(next);  
};  
export default catchAsync;

this is HOC(higher-order function) which returns a function that gets called when a request is made to an endpoint.  
Replace authservice.js with the snippet below:

import { Account } from "../models";  
import { findOrCreate } from "./index";  
import { hashPassword } from "../lib/passwordOps";  
import catchAsync from "../lib/catchAsync";export const signupService = catchAsync(async (req, res, next) => {  
const password = await hashPassword(req.body.password);const email = req.body.email.toLowerCase();  
const [account, created] = await findOrCreate(Account, {  
...req.body,  
password,  
email  
});  
if (!created) {  
return res.status(400).json({  
status: "fail",  
message: "user already exist"  
});  
}  
return res.status(201).json({  
status: "success",  
message: "user successfully created",  
payload: account.toJSON()  
});  
});

At the code above the catchAsync function accept the request-response function as a parameter, return request-response with its specific catch block.

we need to import our apiRouter from routes/index into src/index.js. let

replace src/index.js content with the code snippet below:

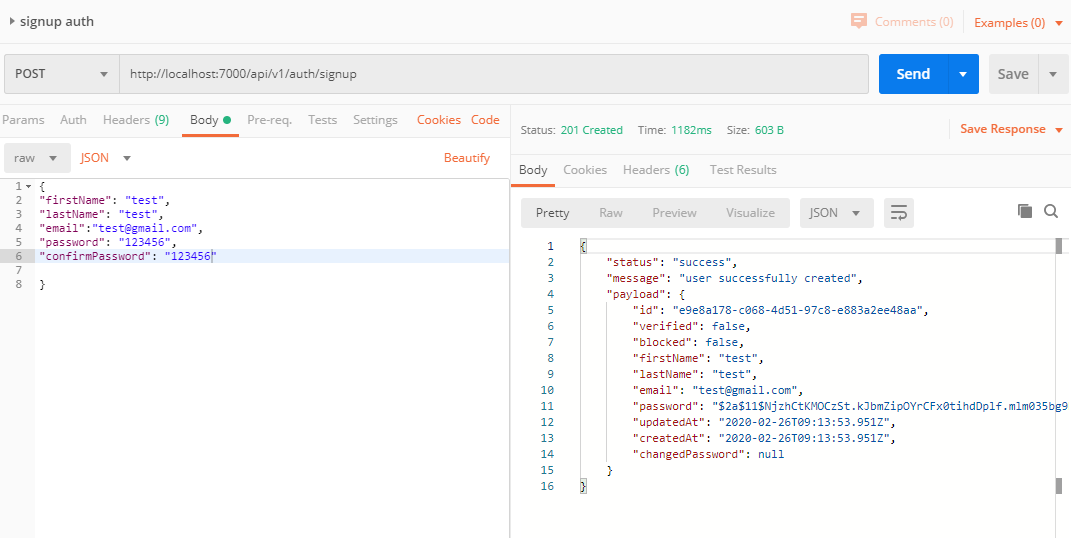
import express from "express";  
import morgan from "morgan";  
import dotenv from "dotenv";  
import apiRouter from "./routes";const app = express();  
dotenv.config();app.use(morgan("dev"));app.use(express.json()); // \*app.use(apiRouter); // \*\*  
const PORT = process.env.PORT || 7000;  
app.listen(PORT, () => {  
console.log(`server listen at port ${PORT}`);});

in order to read the request body, we pass express.json() as a middleware and also the entire route as also a middleware.

we are good to go, let start our and test out our registration endpoint using postman

npm run start:dev

Image for post

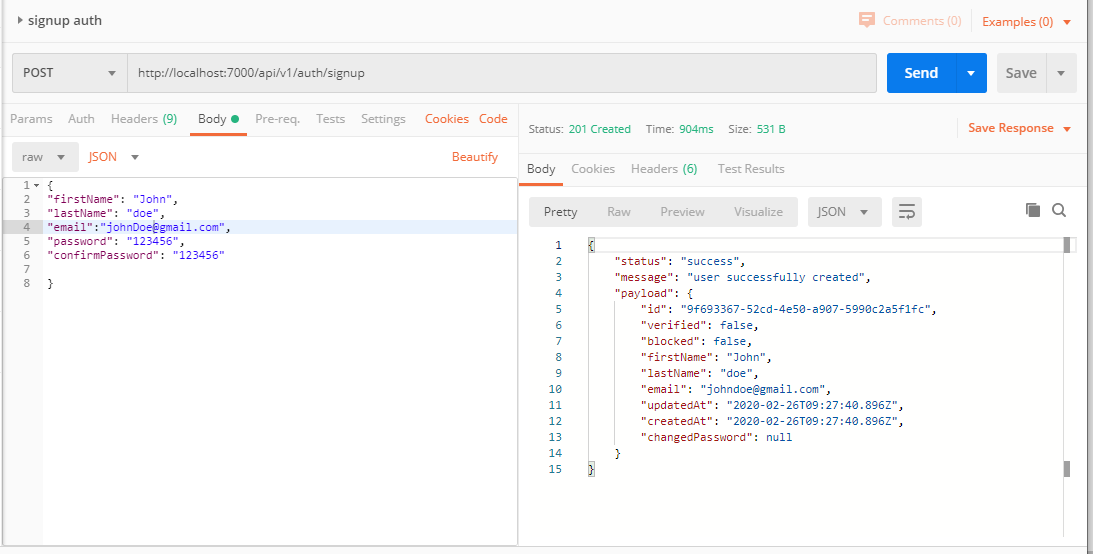


Note: we wouldn’t want to return the password as part of the response, despite it’s encrypted, we can omit the password using the lodash helper function.

Edit services/authservice.js

import Model from "../models";  
import \_ from "lodash"; // lodash import  
import { findOrCreate } from "./index";  
import { hashPassword } from "../lib/passwordOp";  
import catchAsync from "../lib/catchAsync";const { Account } = Model;  
export const signupService = catchAsync(async (req, res, next) => {const password = await hashPassword(req.body.password);  
const email = req.body.email.toLowerCase();  
const [account, created] = await findOrCreate(Account, {  
...req.body,password,email});if (!created) {  
return res.status(400).json({  
status: "fail",  
message: "user already exist"  
});  
}  
return res.status(201).json({  
status: "success",  
message: "user successfully created",  
payload: \_.omit(account.toJSON(), ["password"])  
});  
});

Image for post



Before moving forward, we should be able to have a single place where we can handle errors like user requesting for an endpoint that does not exist, user token expired, theses are operational errors, and we need to handle it at the global level.  
How do we do that?

**Global Error Handler in Express**

Express comes with error handler out of the box, and in order to make express understand request handler as error handler we need to pass four arguments to the request handler

(err, req, res, next) => {  
//handler necessary errors}

Navigate to the lib and create globalErrorHandler.js and paste the code below:

const errorHandler = (err, req, res, next) => {  
err.statusCode = err.statusCode || 500; // \*  
err.status = err.status || "error"; // \*\*  
if (process.env.NODE\_ENV === "development") {  
res.status(err.statusCode).json({  
status: err.status,  
message: err.message,  
err: err.stack  
});}  
if (process.env.NODE\_ENV === "production") {  
return res.status(500).json({  
message: "something went wrong!",  
status: "error"});}};export default errorHandler;

Looking at line \* and \*\* you would likely ask where statusCode and status is coming from, whether is part of our err properties. Hold let see what’s was going on inside err

create a file globalError.js inside the lib directory,

class GlobalError extends Error {constructor(message, statusCode) {super(message);this.statusCode = statusCode;this.status = `${statusCode}`.startsWith('4') ? 'fail' : 'error'; // \*}  
}export default GlobalError;

Looking at the snippet above the GlobalError is our custom error class which is inheriting from the built-in javascript Error constructor. The constructor of the parent error class expects a single argument that is the reason we pass message as the only argument to super on line \* where are checking for user prompted errors and developer/server prompt error. with that said, we can raise an error by instantiating the GlobalError passing in the necessary argument.

new GlobalError('User already exist', 400)

Note: in express js whenever we pass an argument to the next function, express handles it as an error, so the next function will be the good place to pass our generic GlobalError handler.

next(new GlobalError('User already exist', 400))

last we need to import errorHandler into our src/index and pass it as a middleware.

Note: the error handler middleware should be the last middleware in the middleware stack.

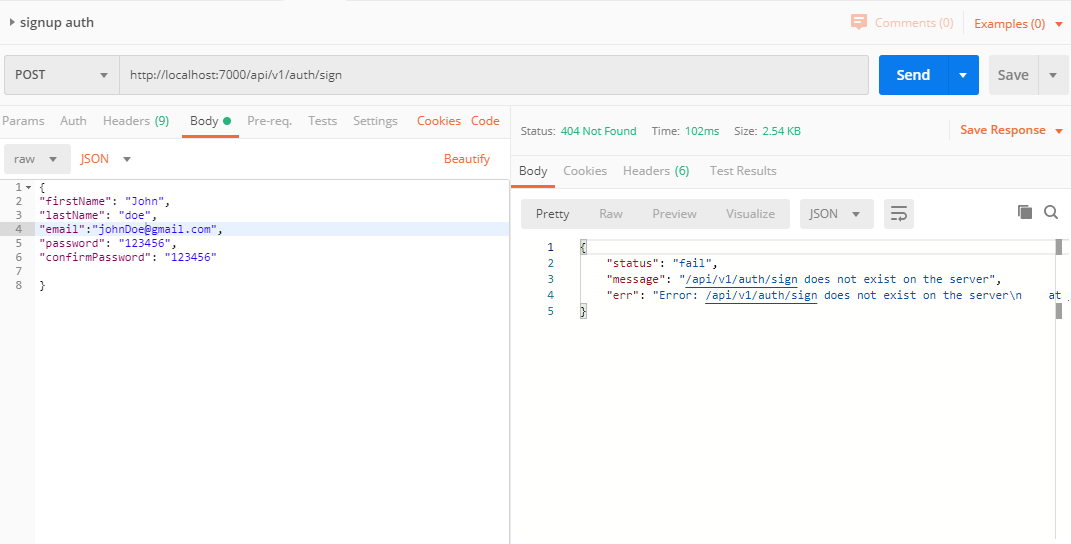
Replace the content of the src/index.js with the snippet below

import express from "express";  
import morgan from "morgan";  
import dotenv from "dotenv";import apiRouter from "./routes";  
import errorHandler from "./lib/globalErrorHandler";  
import GlobalError from "./lib/globalError";const app = express();  
dotenv.config();  
app.use(morgan("dev"));  
app.use(express.json());  
app.get("/", (req, res) => {  
console.log("hello world");});app.use(apiRouter);app.all("\*", async (req, res, next) => { // \*  
const err = new GlobalError( //\*\*  
`${req.originalUrl} does not exist on the server`,  
404  
);  
next(err); // \*\*\*  
});app.use(errorHandler); // \*\*\*\*  
const PORT = process.env.PORT || 7000;  
app.listen(PORT, () => {  
console.log(`server listen at port ${PORT}`);});

At line \* we handling when a user visits an endpoint that does not exist  
at line \*\* we are handling the error with our customer global error handler  
at line \*\*\* we pass the instance of the GlobalError to the next() function, we cause express to handler the error for us.

let test our global error, using postman

Image for post



**Let create our login endpoint:**

paste the snippet into the routes/authRoute.js

import { Router } from "express";import {signupController} from "../controllers/authController";import { signinAuth } from "../middlewares/authMiddleware";  
const authRouter = Router();authRouter.post("/signup", signupController);authRouter.post("/signin", signinAuth);export default authRouter;

create authMiddlware.js inside middlewares directory and paste the code snippet below:

import \_ from 'lodash';  
import Model from '../models';  
import GlobalError from '../lib/globalError';  
import { findUser } from '../services/index';  
import { comparePassord } from '../lib/passwordOp';const { Account } = Model;  
export const signinAuth = async (req, res, next) => {  
const { email, password } = req.body;  
const user = await findUser(Account, email); //\*  
if (!user) {  
return next(new GlobalError('Invalid credential', 400));  
}if (!(await comparePassord(password, user.password))) { //\*\*return next(new GlobalError('Invalid credential', 400));}if (user && user.toJSON().blocked) {  
return next(new GlobalError('Account is blocked, please contact the system administrator',401));}req.user = user.toJSON(); //\*\*\*  
next();};

line \* is an abstracted services function from sequelize API. for the implementation navigate to the services/index inside it.

export const findUser = (model, payload) =>  
model.findOne({  
where: {  
email: payload},logging: false})

and line \*\* navigate to passwordOp.js and paste this implementation

export const comparePassord = (password, dbPassword) =>bcrypt.compare(password, dbPassword);

since the encryption is one-way flow, the bcrypt encrypt the new password and compare it with already store database, it returns true if the two passwords matches.

if all the conditions are met we pass the user object into the request(req) object and calling the next middleware function, every middleware has access to the request object.

Next middleware will be our signinController, I abstracted the code above to make our controller as slim as possible.

paste the code snippet inside the authController

import \_ from 'lodash';  
import { signupService } from '../services/authService';  
import { createTokens } from '../lib/generateToken';  
import catchAsync from '../lib/catchAsync';  
import { createCookie } from '../lib/createCookie';export const signupController = catchAsync(async (req, res, next) => {  
await signupService(req, res, next);});export const signinController = async (req, res) => {const refreshSecret = process.env.JWT\_REFRESH\_KEY + req.user.password; // 1  
const [token, refreshToken] = createTokens( //2  
{id: req.user.id,verified: req.user.verified,blocked: req.user.blocked,role: req.user.role},refreshSecret);const payload = { ...req.user, token, refreshToken };createCookie(res, token, '\_\_act', process.env.JWT\_ACCESS\_TOKEN\_EXPIRES);  
createCookie( // 3res,refreshToken,'\_\_rt',process.env.JWT\_REFRESH\_TOKEN\_EXPIRES);return res.status(200).json({status: "success",message: "user successfully created",payload: \_.omit(payload, ["password"])});  
};

Let analysis the code above,

At comment 1, it usually best practice to bind user refresh key secret to password, why? if a user refresh token gets to the wrong hand if the changes his password it automatically invalidates the refresh token because the refresh secret becomes different.

At comment 2, we are generating access token and refresh token. it go over to the implementation for the createTokens functions.

Navigate to the lib directory and add generateToken.js to the folder.

touch generateToken.js

paste the code below in it.

import jwt from "jsonwebtoken";import \_ from "lodash";  
export const createToken = (payload, secretKey, expiresIn) =>jwt.sign(payload, secretKey, {expiresIn,audience: process.env.JWT\_AUDIENCE,issuer: process.env.JWT\_ISSUER});export const createTokens = (payload, refreshSecret) => {const token = createToken(payload,process.env.JWT\_SECRET\_KEY,`${process.env.JWT\_ACCESS\_TOKEN\_EXPIRES}`);const refreshToken = createToken(payload,refreshSecret,`${process.env.JWT\_REFRESH\_TOKEN\_EXPIRES}`);return [token, refreshToken];};

in our .env file we have the following, remember never commit your something your .env file

JWT\_AUDIENCE=example.com,JWT\_ISSUER=example.comJWT\_SECRET\_KEY=example-is-my-secret-keyNODE\_ENV=developmentJWT\_REFRESH\_KEY=example-is-my-refresh-key-tokenREFRESH\_TOKEN\_COOKIE\_EXPIRES=30ACCESS\_TOKEN\_COOKIE\_EXPIRES=60JWT\_ACCESS\_TOKEN\_EXPIRES=30sJWT\_REFRESH\_TOKEN\_EXPIRES=7d

at comment 3 in authController.js we are storing our token and refresh token in the cookie. let see the implementation.

inside in the lib directory create createCookie.js file and paste the snippet below:

export const createCookie = (res, token, name, expires) => {const cookieOption = {httpOnly: true,expires: new Date(Date.now() + expires \* 24 \* 60 \* 60 \* 1000),secure: process.env.NODE\_ENV === "production" ? true : false};return res.cookie(name, token, cookieOption);};

if we are using cookie to store our tokens it must be done securely, how?

* httpOnly: true force the browser to only send it with changing its value,
* secure: true means cookie can only be sent in a secure connection,
* expires is the time it takes for the cookie to get expires in millisecond

Our cookie will not work without cookie middleware, to get the cookie middleware we need to install cookie-parser

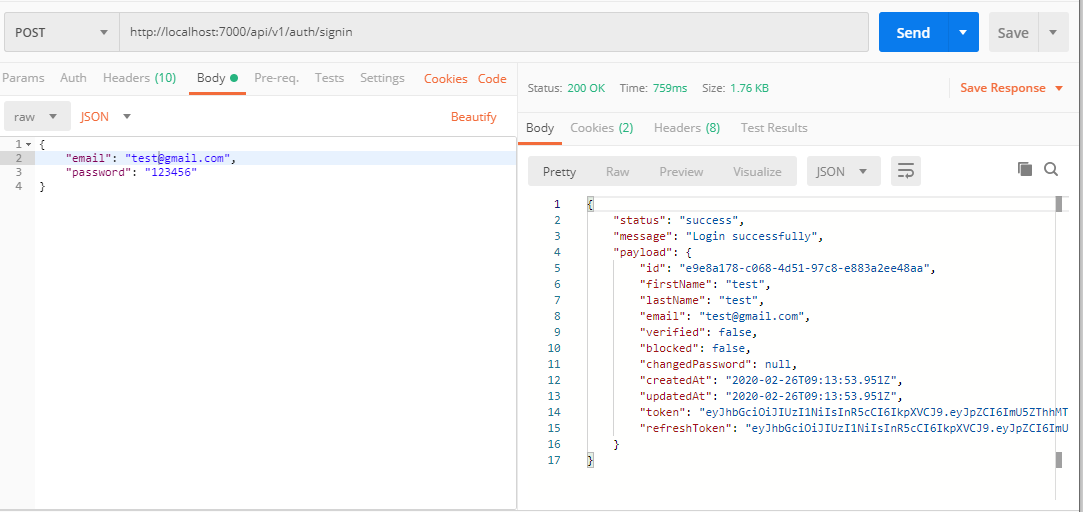
npm install cookie-parser

and add the code snippet to the src/index.js

import express from "express";  
import morgan from "morgan";  
import dotenv from "dotenv";  
import cookieParser from "cookie-parser";  
import apiRouter from "./routes";  
import errorHandler from "./lib/globalErrorHandler";  
import GlobalError from "./lib/globalError";const app = express();dotenv.config();app.use(morgan("dev"));app.use(express.json());app.use(cookieParser()); // cookie parse middleware to access the cookie objectapp.use(apiRouter);app.all("\*", async (req, res, next) => {const err = new GlobalError(`${req.originalUrl} does not exist on the server`,404);next(err);});app.use(errorHandler);const PORT = process.env.PORT || 7000;app.listen(PORT, () => {console.log(`server listen at port ${PORT}`);});

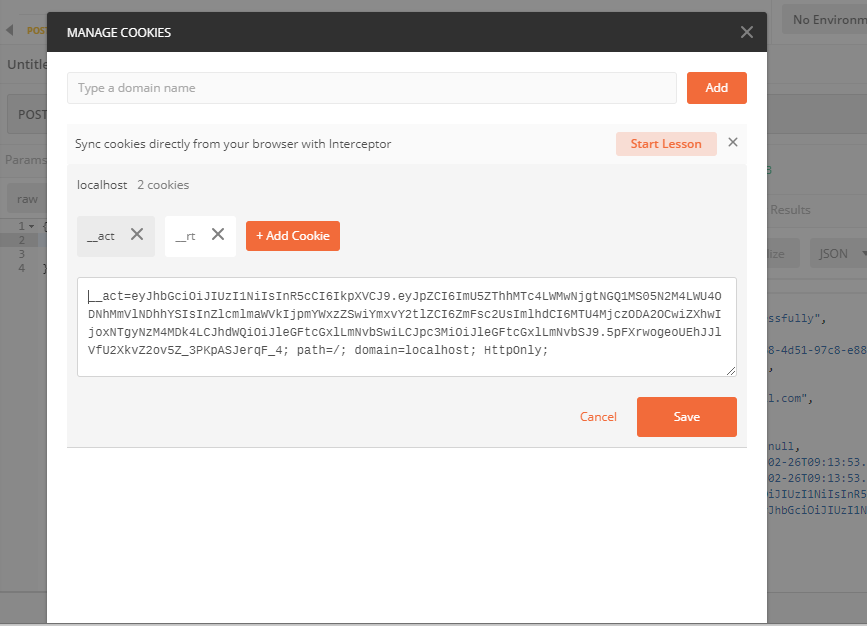
let test our login route using postman

Image for post



login request response

Image for post



congratulation for making it to this point

Image for post



I made it

**Getting started with a protected route.**

To protect certain resources from unauthenticated users to active them access just resources. middleware pipeline helps us to build just an authorisation system.

How do we achieve that?

Navigate to routes directory and create userRoute.js

paste the snippet below

import { Router } from 'express';import { jwtProtect } from '../middlewares/jwtAuthMiddleware';const router = Router();  
router.get('/', jwtProtect); // will soon add middleware pipelineexport default router;

Go to middlewares directory and create jwtAuthMiddleware.js and paste the snippet below:

import GlobalError from "../lib/globalError";  
import { jwtVerifyToken, refreshToken } from "../lib/generateToken";  
import { isPasswordChanged } from "../lib/passwordOp";  
import catchAsync from "../lib/catchAsync";  
import Model from "../models";  
import { findByPk } from "../services/index";  
import { createCookie } from "../lib/createCookie";  
const { Account } = Model;export const jwtProtect = catchAsync(async (req, res, next) => {const token = req.cookies.\_\_act; // 1if (!token) {  
return next(new GlobalError("You are not logged in", 401));}try {  
const decoded = await jwtVerifyToken(token); //2  
if (decoded.blocked) {  
return next(new GlobalError("Your account has been blocked, contact system administrator ",401));}  
const freshUser = await findByPk(Account, decoded.id); // 3if (!freshUser) {return next(new GlobalError("user from does not exist", 401));}const passwordChangeAt = Math.round(new Date(`${freshUser.toJSON().changedPassword}`).getTime() / 1000);if (isPasswordChanged(decoded.iat, passwordChangeAt)) { //4return next(new GlobalError("You are not logged in, please login with correct details",401));}req.user = freshUser.toJSON();next();} catch (err) {const { \_\_rt } = req.cookies;const { accessToken, newRefreshToken } = await refreshToken(\_\_rt, next);if (accessToken && newRefreshToken) {createCookie(res,accessToken,"\_\_act",process.env.ACCESS\_TOKEN\_COOKIE\_EXPIRES);createCookie(res,newRefreshToken,"\_\_rt",process.env.REFRESH\_TOKEN\_COOKIE\_EXPIRES);  
next();}}});

lets what is the code is actually doing. At comment #1 we are retrieving the token from the cookie, the cookie object what available to us because of the cookie middleware at src/index .

At line comment #2 we verify if the token is still valid, a token can become invalid if it has expired if the token has been changed.

let see the implementation of the jwtVerifyToken ,

paste the code into the generateToken.js

export const jwtVerifyToken = token => jwt.verify(token, process.env.JWT\_SECRET\_KEY);

this function returns the decoded object if the token is still valid otherwise it throw up error.

At comment #3 and #4 we are checking if the user still and if the password has been changed. if the password has been changed after the issuance of the token, the already existing token automatically becomes invalid.

Going to their implementation:

export const findByPk = (model, id) => model.findByPk(id);

paste the snippet above inside services/index.js and

export const isPasswordChanged = (jwtExpiresTime, passwordChangedAt) => passwordChangedAt > jwtExpiresTime;

inside passwordOp.js

if the token has expired and the refresh token is stil valid we will use it to get a new token and also a new refresh token. let look out the implementation of refreshToken from the generateToken.js add the code below:

import jwt from "jsonwebtoken";import { promisify } from "util";import \_ from "lodash";import { findByPk } from "../services";import Model from "../models";import GlobalError from "../lib/catchAsync";const { Account } = Model;export const createToken = (payload, secretKey, expiresIn) =>jwt.sign(payload, secretKey, {expiresIn,audience: process.env.JWT\_AUDIENCE,issuer: process.env.JWT\_ISSUER});export const createTokens = (payload, refreshSecret) => {const token = createToken(payload,process.env.JWT\_SECRET\_KEY,`${process.env.JWT\_ACCESS\_TOKEN\_EXPIRES}`);const refreshToken = createToken(payload,refreshSecret,`${process.env.JWT\_REFRESH\_TOKEN\_EXPIRES}`);return [token, refreshToken];};export const jwtVerifyToken = token =>jwt.verify(token, process.env.JWT\_SECRET\_KEY);export const refreshToken = async (\_\_rt, next) => {const decoded = jwt.decode(\_\_rt);if (!decoded.id) {return next(new GlobalError("unAuthorize, please login", 401));}  
const freshUser = await findByPk(Account, decoded.id);if (!freshUser) {return next(new GlobalError("user does not exist", 401));}const refreshSecret =process.env.JWT\_REFRESH\_KEY + freshUser.toJSON().password;try {await promisify(jwt.verify)(\_\_rt, refreshSecret);} catch (err) {return next(new GlobalError("you are not logged in from refreshtoken", 401));}const [token, refreshToken] = createTokens(\_.pick(freshUser.toJSON(), ["id", "verified", "blocked", "role"]),refreshSecret);if (token && refreshToken) {return {accessToken: token,newRefreshToken: refreshToken};}};

we started out with decoding the token without the jwt verification if the user actually exists we need to verify the token, but remember the refresh token secret is bind to their password. if old refresh token is valid we can go ahead to to create a new token and new refresh token.

Navigate to routes/userRoute.js paste the code below:

import { Router } from "express";  
import { jwtProtect } from "../middlewares/jwtAuthMiddleware";  
import { getAllUsers } from "../controllers/userController";const router = Router();router.get("/", jwtProtect, getAllUsers);export default router;

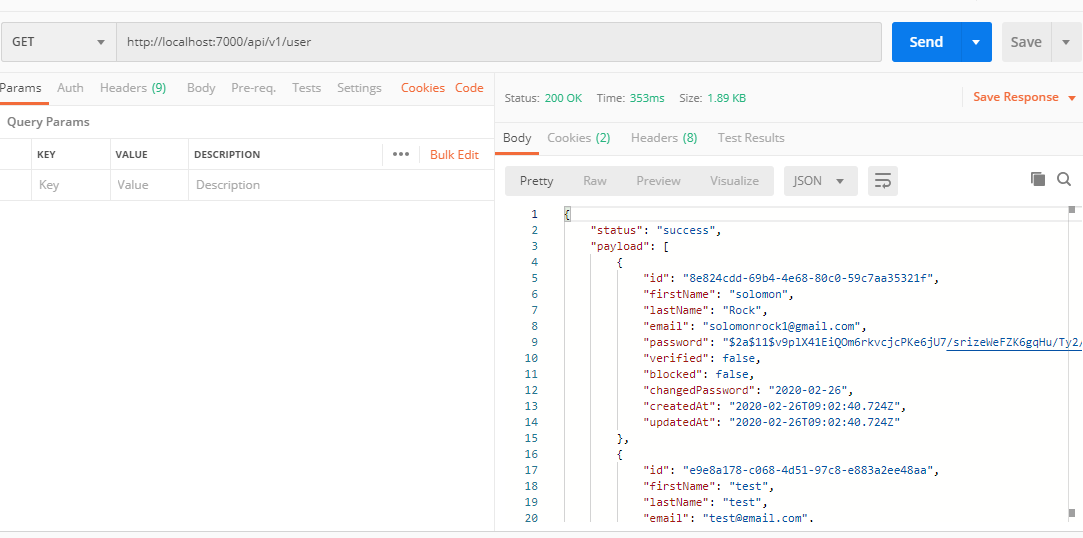
if the request passes the jwtProtect middleware the getAllUsers get called ;

Navigate to the controllers and create userController.js paste the snippet below:

import \_ from "lodash";  
import { findAll } from "../services";  
import Model from "../models";  
import catchAsync from "../lib/catchAsync";const { Account } = Model;export const getAllUsers = catchAsync(async (req, res, next) => {const users = await findAll(Account);const body = users.map(user => \_.omit(user, ["password"]));return res.status(200).json({status: "success",payload: users});});

lets test the endpoint:

Image for post



if the user token get expired, the refresh token auto-generate a new token for the user.

Download code by here <https://github.com/solomonfrank/jwt-refresh-token>