Simple Login and Registration with ExpressJS, Sequelize, bcrypt and JWT

In this guide, we will see how a NodeJS and PostgreSQL web application can incorporate user registration and login feature. We will use bcrypt to hash the password and JWT for token generation.

In this post, we will not create any HTML page or template. Instead, We should consider it as a process to add login and registration functionality in REST API.

We will go through this post in following steps:

1. Setting up ExpressJS application with Sequelize
2. Installing NPM dependencies .i.e. Sequelize, Sequelize-cli, bcrypt and jsonwebtoken
3. Initializing project with sequelize-cli and configuring PostgreSQL db
4. Creating Users table using Sequelize migration
5. Writing the User registration logic
6. Writing login function
7. Securing a route using middleware and JWT verification

1. Setting up ExpressJS application with Sequelize

To setup expressjs application with Sequelize, we will first create an empty project directory and then open terminal. Now run `npx express-generator .`. This will generate project structure in the directory. Now run `npm install` to install all the dependencies listed in package.json file.

2. Installing NPM dependencies .i.e. Sequelize, Sequelize-cli, bcrypt and jsonwebtoken

Let's quickly summarize which dependencies are needed and why ?

1. *PostgreSQL2* is the package utilized by Sequelize ORM to interact with PostgreSQL DB.

2. *sequelize* offers great support in order to deal with CRUD operations and relationship management between tables in PostgreSQL.

3. *sequelize-cli* makes it super easy to create migrations, seeders and models with CLI commands.

4. *bcrypt* will be used to hash the plain password.

5. *jsonwebtoken* generates JWT token which can be used to authenticate the user.

6. *dotenv* allows us to use .env file and access it's content with `process.env.MY_ENV_VARIABLE`

To install all above packages, run `npm install --save PostgreSQL2 sequelize sequelize-cli bcrypt jsonwebtoken`. We now have the structure ready to start coding.

3. Initializing project with sequelize-cli and configuring PostgreSQL db

`npx sequelize-cli init` command will create config, migrations, models and seeders directories. These directories will contain their respective files generated from various sequelize-cli commands.

Go ahead and open config.json file from config directory. Modify the content of this file according to your DB settings.

```
{
  "development": {
    "username": "admin",
    "password": "admin12345",
    "database": "blog_post_db",
```

```
    "host": "127.0.0.1",
    "dialect": "postgres"
  },
  "test": {
    "username": "admin",
    "password": "admin12345",
    "database": "blog_post_db",
    "host": "127.0.0.1",
    "dialect": "postgres"
  },
  "production": {
    "username": "admin",
    "password": "admin12345",
    "database": "blog_post_db",
    "host": "127.0.0.1",
    "dialect": "postgres"
  }
}
```

4. Creating Users table using Sequelize migration

The sequelize's way of creating a table is generating a migration and run it. Type `npx sequelize-cli model:generate --name User --attributes first_name:string,last_name:string,email:string,password:string` . This will create a User model and it's migration as well. You can verify the presence of migration file in the migrations directory.

```
'use strict';
module.exports = {
  up: async (queryInterface, Sequelize)=>{
    await queryInterface.createTable('Users', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
```

```
      type: Sequelize.INTEGER
    },
    first_name: {
      type: Sequelize.STRING
    },
    last_name: {
      type: Sequelize.STRING
    },
    email: {
      type: Sequelize.STRING
    },
    password: {
      type: Sequelize.STRING
    },
    createdAt: {
      allowNull: false,
      type: Sequelize.DATE
    },
    updatedAt: {
      allowNull: false,
      type: Sequelize.DATE
    }
```

```
    });
  },
  down: async (queryInterface, Sequelize)=>{
    await queryInterface.dropTable('Users');
  }
};
```

Now run `npx sequelize-cli db:migrate` command. If you don't already had the database created, You need to run `npx sequelize-cli db:create` before running the db:migrate. Once migrated, you should see the Users table in your PostgreSQL database.

5. Writing the User registration logic

As we created the project structure with `express-generator`, *user.js* file inside routes directory should already be available with following content:

```
var express = require('express');
var router = express.Router();
```

```
3
4/* GET users listing. */
5router.get('/', function(req, res, next) {
6  res.send('respond with a resource');
7});
8
9module.exports = router;
```

This router.get method will be called whenever a get request is received on http://127.0.0.1:3000/users.

In any REST API, if we need to add a resource, we should be posting to the resource URL. In this case, we will create a POST route for the users such that posting to http://127.0.0.1/users should generate a User.

Let's add a router.post method in user.js file. This method will pick input from post data and create a user.

```
1var express = require('express');
2var router = express.Router();
3const Models = require('./../models');
4const bcrypt = require("bcrypt");
5const jwt = require('jsonwebtoken');
```

```
const dotenv = require('dotenv');
const User = Models.User;
dotenv.config();

/* GET users listing. */
router.get('/', function(req, res, next) {
  res.send('respond with a resource');
});


router.post('/', async(req, res, next)=>{
  //res.status(201).json(req.body);
  //add new user and return 201
  const salt = await bcrypt.genSalt(10);
  var usr = {
    first_name : req.body.first_name,
    last_name : req.body.last_name,
    email : req.body.email,
    password : await bcrypt.hash(req.body.password, salt)
  };
  created_user = await User.create(usr);
  res.status(201).json(created_user);
```

```
});

module.exports = router;
```

Here, we are using bcrypt as our password hashing mechanism. Follow this link for a detailed explanation of bcrypt.

`created_user = await User.create(usr)` is the code responsible for adding the user record in the PostgreSQL database.

6. Writing the login function

Login function will expect user's email and password. On successful password match, it will return a JWT token.

```
router.post('/login',async(req,res,next)=>{
const user = await User.findOne({ where : {email : req.body.email }});
if(user){
    const password_valid = await bcrypt.compare(req.body.password,user.password);
    if(password_valid){
```

```
        token = jwt.sign({ "id" : user.id,"email" : user.email,"first_name":user.first_name },process.env.SECRET);
        res.status(200).json({ token : token });
    } else {
      res.status(400).json({ error : "Password Incorrect" });
    }

  }else{
    res.status(404).json({ error : "User does not exist" });
  }

  });

```

First off, We are retrieving the user based on the provided email id. `.findOne()` is a sequelize model function which will fetch the first record matching the criteria. In our case, we are finding the record against user's email address.

Once the user is retrieved, we will then compare the provided password against the stored hash using `bcrypt.compare` function. It is important to note that `bcrypt.compare` is not taking salt as a parameter while comparing the password against the stored hash. The reason for this is the fact that salt already exists in the hash.

As the last step, we will generate the JWT using payload and secret with `jwt.sign` method. The payload part .i.e. `{"id": user.id,"email" : user.email,"first_name":user.first_name }` must not contain any sensitive information. The secret should be stored in .env file and should never be disclosed.

At the time of token validation, the decoded payload will help us identify the logged in user. We usually validate the token in a middleware. Token validation is beyond the scope of this post, I will cover this in my upcoming posts.

Upon successful login, the function respond with a JWT token. The client like ReactJS, AngularJS or a mobile application is responsible to save the JWT token in localStorage or cookies. The client is also responsible to add

the JWT token in subsequent HTTP requests using *Authorization* header field.

7. Securing a route using middleware and JWT verification

Once the JWT is generated and sent to the client, it's time to see how we can protect a route and make sure that a valid JWT is available in the request header.

We will create a simple GET route which will respond with the logged in user data. At the time of login, we set the payload in JWT containing the user id, We will retrieve JWT from request header and then obtain user id from the decoded JWT.

`router.get()` method takes first argument as the name of the route, after that we have the option to add as many callbacks as needed. Each of these callbacks would have access to *request*, *response* and *next*. When the response is sent, it is considered as an end of request lifecycle, any of the callbacks in `router.get` could send the response and complete the request lifecycle.

Ideally, each callback should either send the response or call the next function to pass the request to the next callback. From my definition, any callback which performs some kind of checks and pass the request to next callback is called a middleware.

Go ahead and create a */users/me* route in *routes/users.js*.

```
router.get('/me',
async(req,res,next)=>{
  try {
    let token = req.headers['authorization'].split(" ")[1];
    let decoded = jwt.verify(token,process.env.SECRET);
    req.user = decoded;
    next();
  } catch(err){
    res.status(401).json({"msg":"Couldnt Authenticate"});
  }
  },
  async(req,res,next)=>{
    let user = await User.findOne({where:{id : req.user.i
```

```
d},attributes:{exclude:["password"]}});
    if(user === null){
      res.status(404).json({'msg':"User not found"});
    }
    res.status(200).json(user);
});
```

In the first callback function .i.e. middleware, we are checking if the *authorization* header is available in the request. The authorization header sent in following format:

```
Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwi
ZW1haWwiOiJmaWhsYXR2QGdtYWlsLmNvbSIsImZpcnN0X25hbWUiOiJU
b21teVRWIiwiaWF0IjoxNjU2NDA4MzEzfQ.W_sVCTUASPOa8CNnvuDcr
R59kv88fFAskknhs8kL6JA
```

The actual JWT is after `Bearer` , so we used `.split()` function and get it's last index.

`jwt.verify` function returns the decoded payload which in our case, is a JSON object containing the user id and few other fields. Once the decoding done, the payload gets added in request. The request passed to next callback using `next()` function call.

Conclusion:

I skipped validation, email uniqueness etc to keep things simple. I hope this guide will make it easy to understand the concept of simple login and registration in expressjs using sequelize and PostgreSQL db.