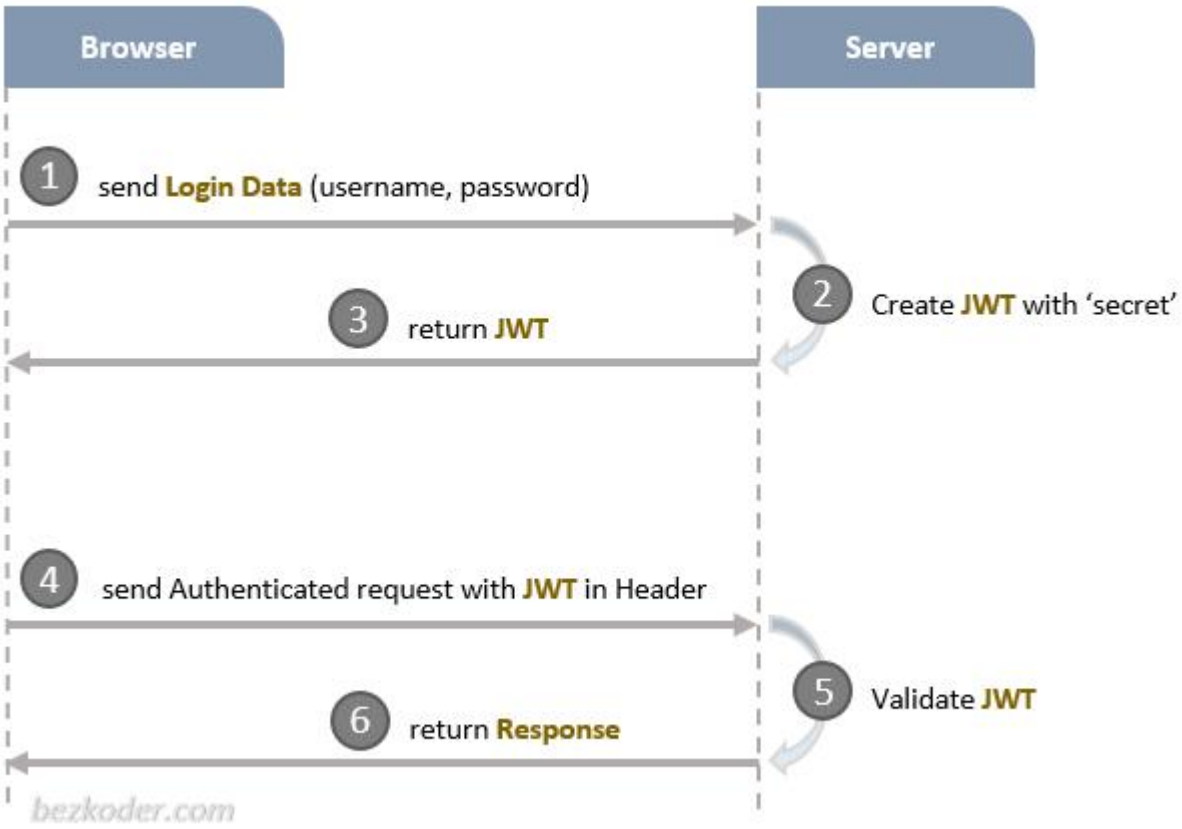
In this tutorial, we're gonna build a Node.js Express Rest API example that supports Token Based Authentication with JWT (JSONWebToken) and PostgreSQL. You'll know:

- Appropriate Flow for User Registration & Login with JWT Authentication
- Node.js Express Architecture with CORS, Authenticaton & Authorization middlewares & Sequelize
- How to configure Express routes to work with JWT
- How to define Data Models and association for Authentication and Authorization
- Way to use Sequelize to interact with PostgreSQL Database

Token Based Authentication

Comparing with Session-based Authentication that need to store Session on Cookie, the big advantage of Token-based Authentication is that we store the JSON Web Token (JWT) on Client side: Local Storage for Browser, Keychain for IOS and SharedPreferences for Android… So we don't need to

build another backend project that supports Native Apps or an additional Authentication module for Native App users.



There are three important parts of a JWT: Header, Payload, Signature. Together they are combined to a standard structure: `header.payload.signature`.

The Client typically attaches JWT in **Authorization** header with Bearer prefix:

```
Authorization: Bearer [header].[payload].[signature]
```

Or only in **x-access-token** header:

```
x-access-token: [header].[payload].[signature]
```

Overview of Node.js Express JWT Authentication with PostgreSQL example

We will build a Node.js Express application in that:

- User can signup new account, or login with username & password.
- User information will be stored in PostgreSQL database
- By User's role (admin, moderator, user), we authorize the User to access resources

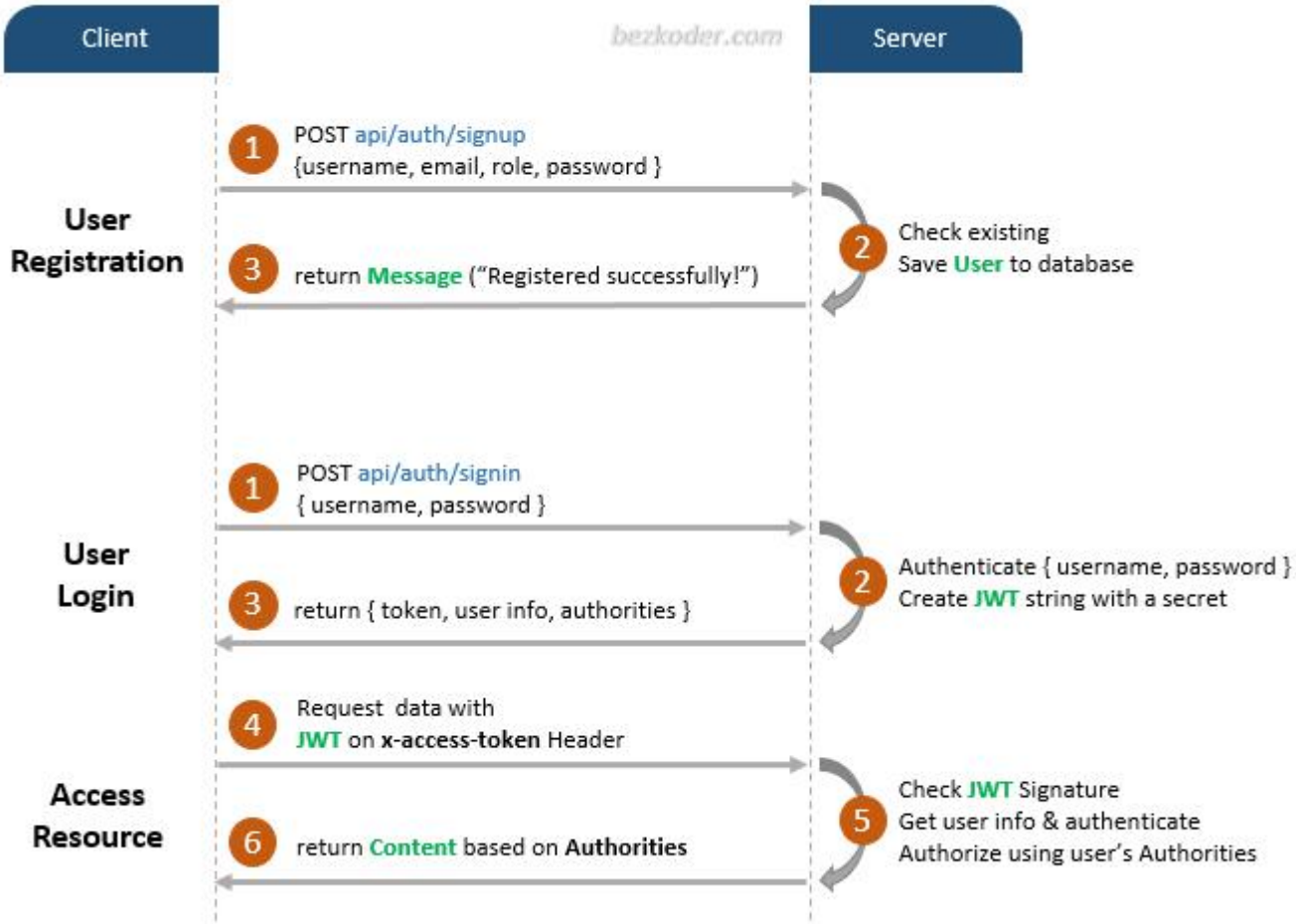
These are APIs that we need to provide:

Methods	Urls	Actions
POST	/api/auth/signup	signup new account
POST	/api/auth/signin	login an account
GET	/api/test/all	retrieve public content
GET	/api/test/user	access User's content

Methods	Urls	Actions
GET	/api/test/mod	access Moderator's content
GET	/api/test/admin	access Admin's content

Flow for Signup & Login with JWT Authentication

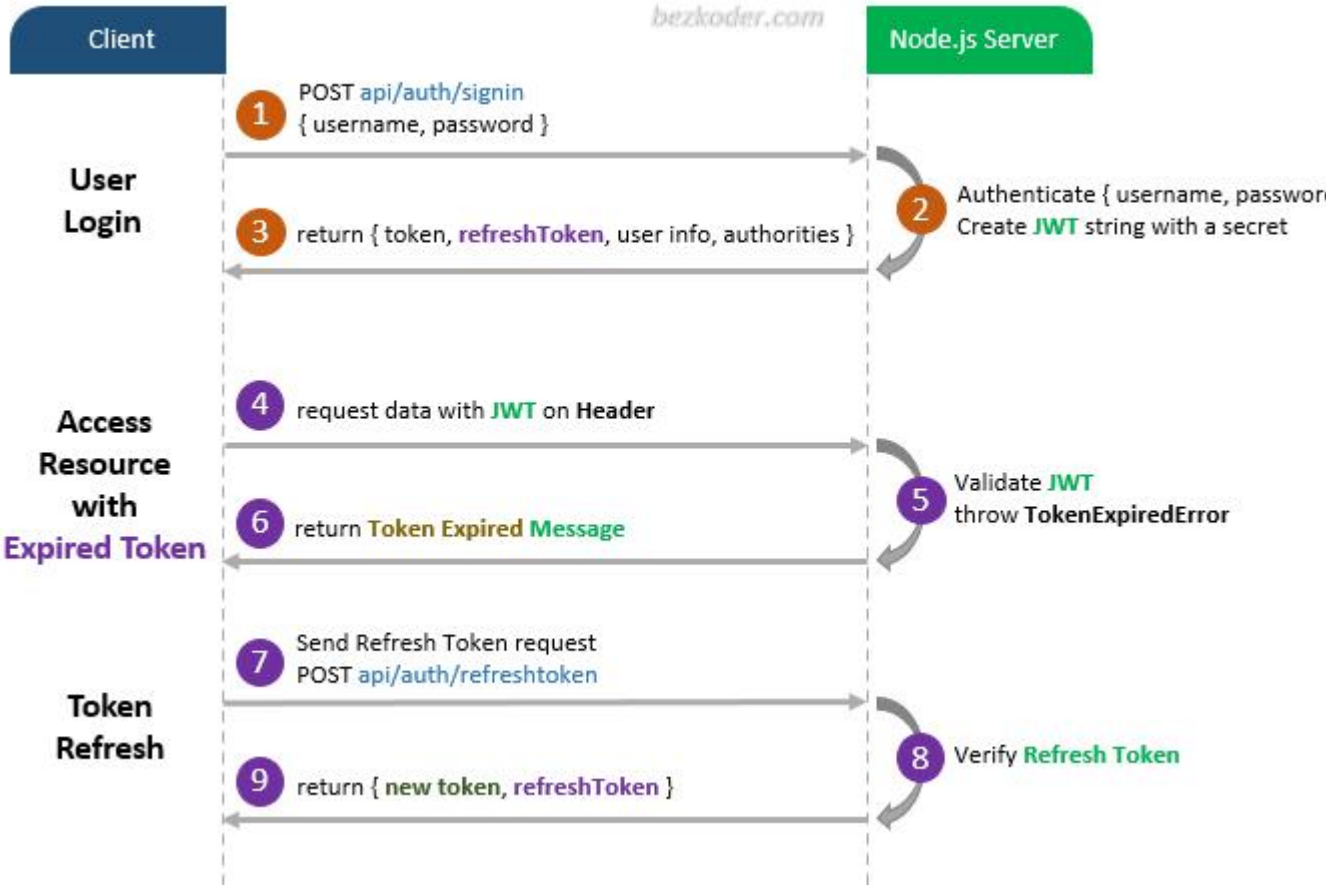
The diagram shows flow of User Registration, User Login and Authorization process.



A legal JWT must be added to HTTP **x-access-token** Header

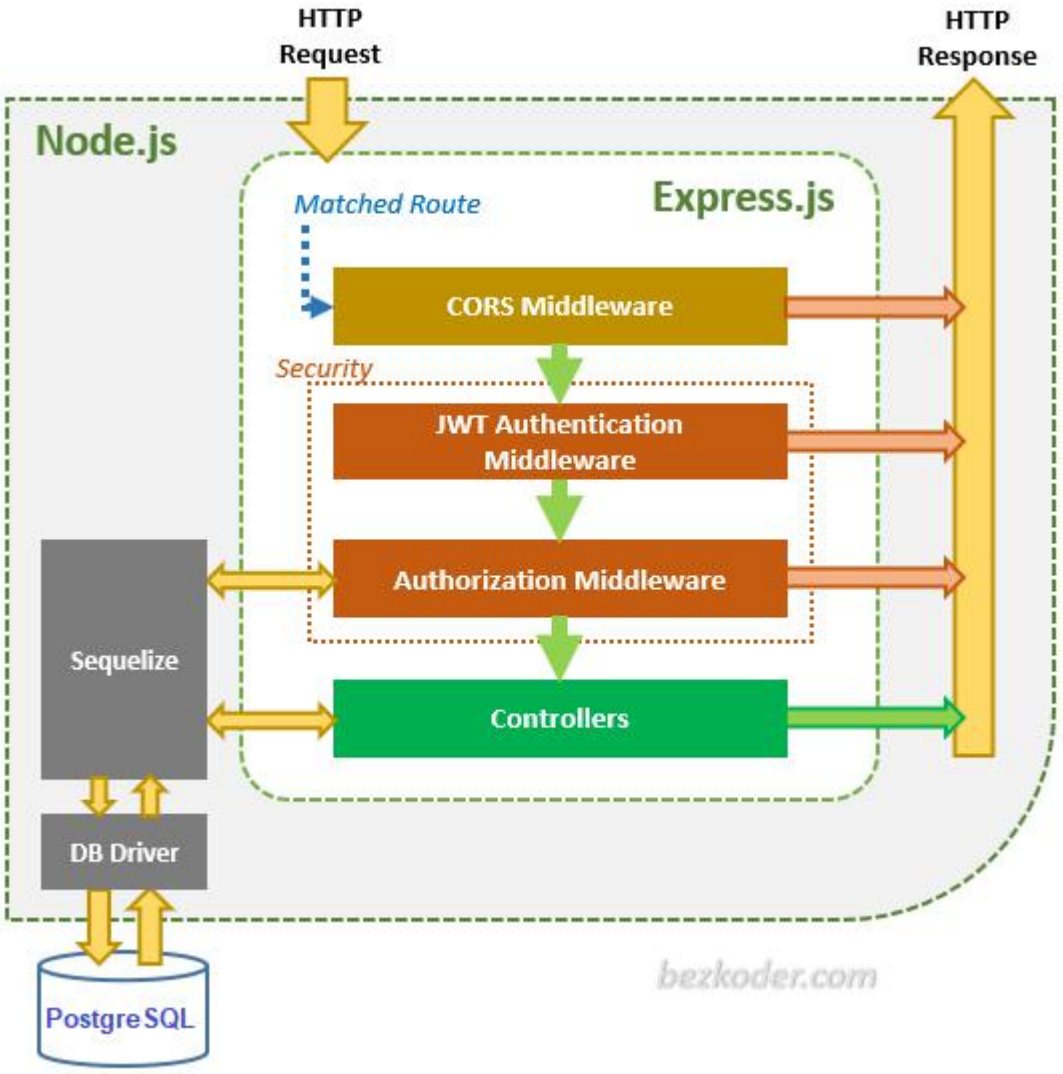
if Client accesses protected resources.

You will need to implement Refresh Token:



Node.js Express Architecture with Authentication & Authorization

You can have an overview of our Node.js Express JWT Auth App with the diagram below:



Via Express routes, **HTTP request** that matches a route will be checked by **CORS Middleware** before coming to **Security** layer.

Security layer includes:

- JWT Authentication Middleware: verify SignUp, verify token

- Authorization Middleware: check User's roles with record in database

If these middlewares throw any error, a message will be sent as HTTP response.

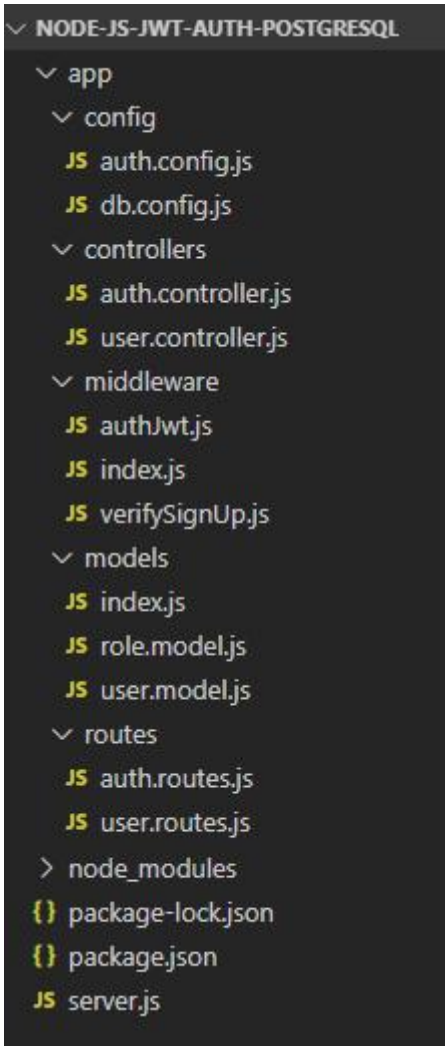
Controllers interact with PostgreSQL Database via Sequelize and send **HTTP response** (token, user information, data based on roles…) to client.

Technology

```
 body-parser@1.20.0
+ pg@8.7.3
+ cors@2.8.5
+ express@4.18.1
+ jsonwebtoken@8.5.1
+ pg-hstore@2.3.4
+ bcryptjs@2.4.3
+ sequelize@6.21.0
```

Project Structure

This is directory structure for our Node.js JWT Authentication with PostgreSQL application:



Let me explain it briefly.

– **config**

- configure PostgreSQL database & Sequelize
- configure Auth Key

– **routes**

- auth.routes.js: POST signup & signin
- user.routes.js: GET public & protected resources

– **middlewares**

- verifySignUp.js: check duplicate Username or Email
- authJwt.js: verify Token, check User roles in database

– **controllers**

- auth.controller.js: handle signup & signin actions
- user.controller.js: return public & protected content

– **models** for Sequelize Models

- user.model.js
- role.model.js

– server.js: import and initialize neccesary modules and routes, listen for connections.

Create Node.js App

First, we create a folder for our project:

```
$ mkdir node-js-jwt-auth-postgresql
```

```
$ cd node-js-jwt-auth-postgresql
```

Then we initialize the Node.js App with a package.json file:

```
npm init

name: (node-js-jwt-auth-postgresql)

version: (1.0.0)

description: Node.js Demo for JWT Authentication with
PostgreSQL database

entry point: (index.js) server.js

test command:

git repository:

keywords: node js, express, jwt, authentication, postg
resql

author:TommyTV

license: (ISC)

Is this ok? (yes) yes
```

We need to install necessary

modules: `express`, `cors`, `body-parser`, `sequelize`, `pg`, `p`

`g-hstore`, `jsonwebtoken` and `bcryptjs`.

Run the command:

```
npm install express sequelize pg pg-hstore body-parser
 cors jsonwebtoken bcryptjs --save
```

*`pg` for PostgreSQL and `pg-hstore` for converting data into the PostgreSQL hstore format.

The 聽 package.json 聽 file now looks like this:

```
{
  "name": "node-js-jwt-auth-postgresql",
  "version": "1.0.0",
  "description": "Node.js Demo for JWT Authentication with PostgreSQL database",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "node js",
    "jwt",
    "authentication",
    "express",
    "postgresql"
```

```
],

"author": "fulstackdev",

"license": "ISC",

"dependencies": {

  "bcryptjs": "^2.4.3",

  "body-parser": "^1.19.0",

  "cors": "^2.8.5",

  "express": "^4.17.1",

  "jsonwebtoken": "^8.5.1",

  "pg": "^7.17.1",

  "pg-hstore": "^2.3.3",

  "sequelize": "^5.21.3"

}}
```

Setup Express web server

In the root folder, let's create a new server.js file:

```
const express = require("express");const bodyParser = require("body-parser");const

cors = require("cors");const app = express();var corsOptions = {
```

```
  origin: "http://localhost:8081"};

app.use(cors(corsOptions));// parse requests of content-type - application/json

app.use(bodyParser.json());// parse requests of content-type -

application/x-www-form-urlencoded

app.use(bodyParser.urlencoded({ extended: true }));// simple route

app.get("/", (req, res) => {

  res.json({ message: "Welcome to fulstackdev application." });});// set port, listen for

requestsconst PORT = process.env.PORT || 8080;

app.listen(PORT, () => {

  console.log(`Server is running on port ${PORT}.`);});
```

Let me explain what we've just done:

- import `express`, `body-parser` and `cors` modules:

  - Express is for building the Rest apis
  - body-parser helps to parse the request and create the `req.body` object
  - cors provides Express middleware to enable CORS

- create an Express app, then add `body-parser` and `cors` middlewares using `app.use()` method. Notice that we set origin: `http://localhost:8081`.

– define a GET route which is simple for test.

– listen on port 8080 for incoming requests.

Now let's run the app with command: `node server.js`.

Open your browser with url http://localhost:8080/, you will see:

Configure PostgreSQL database & Sequelize

In the **app** folder, create **config** folder for configuration with db.config.js file like this:

```
module.exports = {
  HOST: "localhost",
  USER: "postgres",
  PASSWORD: "123",
  DB: "testdb",
  dialect: "postgres",
```

```
pool: {
  max: 5,
  min: 0,
  acquire: 30000,
  idle: 10000
}};
```

First five parameters are for PostgreSQL connection.

`pool` is optional, it will be used for Sequelize connection pool configuration:

- `max`: maximum number of connection in pool
- `min`: minimum number of connection in pool
- `idle`: maximum time, in milliseconds, that a connection can be idle before being released
- `acquire`: maximum time, in milliseconds, that pool will try to get connection before throwing error

For more details, please visit API Reference for the Sequelize constructor.

Define the Sequelize Model

In models folder, create `User` and `Role` data model as following code:

models/user.model.js

```
module.exports = (sequelize, Sequelize) => {

  const User = sequelize.define("users", {

    username: {

      type: Sequelize.STRING

    },

    email: {

      type: Sequelize.STRING

    },

    password: {

      type: Sequelize.STRING

    }

  });

  return User;};
```

models/role.model.js

```
module.exports = (sequelize, Sequelize) => {
```

```
const Role = sequelize.define("roles", {
  id: {
    type: Sequelize.INTEGER,
    primaryKey: true
  },
  name: {
    type: Sequelize.STRING
  }
});
return Role;};
```

These Sequelize Models represents **users** & **roles** table in PostgreSQL database.

After initializing Sequelize, we don't need to write CRUD functions, Sequelize supports all of them:

- create a new User: `create(object)`
- find a User by id: `findByPk(id)`
- find a User by email: `findOne({ where: { email: ... } })`

- get all Users: `findAll()`
- find all Users by username: `findAll({ where: { username: ... } })`

These functions will be used in our Controllers and Middlewares.

Initialize Sequelize

Now create **app/models**/index.js with content like this:

```
const config = require("../config/db.config.js");const Sequelize =
require("sequelize");const sequelize = new Sequelize(
  config.DB,
  config.USER,
  config.PASSWORD,
  {
    host: config.HOST,
    dialect: config.dialect,
    operatorsAliases: false,
    pool: {
      max: config.pool.max,
```

```
        min: config.pool.min,
        acquire: config.pool.acquire,
        idle: config.pool.idle
      }
  });const db = {};
db.Sequelize = Sequelize;
db.sequelize = sequelize;
db.user = require("../models/user.model.js")(sequelize, Sequelize);
db.role = require("../models/role.model.js")(sequelize, Sequelize);
db.role.belongsToMany(db.user, {
  through: "user_roles",
  foreignKey: "roleId",
  otherKey: "userId"});
db.user.belongsToMany(db.role, {
  through: "user_roles",
  foreignKey: "userId",
  otherKey: "roleId"});
db.ROLES = ["user", "admin", "moderator"];
module.exports = db;
```

The association between Users and Roles is Many-to-Many relationship:

– One User can have several Roles.

– One Role can be taken on by many Users.

We use `User.belongsToMany(Role)` to indicate that the user model can belong to many Roles and vice versa.

With `through, foreignKey, otherKey`, we're gonna have a new table **user_roles** as connection between **users** and **roles** table via their primary key as foreign keys.

If you want to know more details about how to make Many-to-Many Association with Sequelize and Node.js, please visit:

Sequelize Many-to-Many Association example

Don't forget to call `sync()` method in server.js.

```
...const app = express();

app.use(...);const db = require("./app/models");const Role = db.role;

db.sequelize.sync({force: true}).then(() => {

  console.log('Drop and Resync Db');

  initial();});...function initial() {
```

```
Role.create({
  id: 1,
  name: "user"
});


Role.create({
  id: 2,
  name: "moderator"
});


Role.create({
  id: 3,
  name: "admin"
});}
```

`initial()` function helps us to create 3 rows in database. In development, you may need to drop existing tables and re-sync database. So you can use `force: true` as code above.

For production, just insert these rows manually and use `sync()` without parameters to avoid dropping data:

For production, just insert these rows manually and use `sync()` without parameters to avoid dropping data:

```
...const app = express();

app.use(...);const db = require("./app/models");

db.sequelize.sync();...
```

Learn how to implement Sequelize One-to-Many Relationship at:

Sequelize Associations: One-to-Many example

Configure Auth Key

jsonwebtoken functions such as `verify()` or `sign()` use algorithm that needs a secret key (as String) to encode and decode token.

In the **app/config** folder, create auth.config.js file with following code:

```
module.exports = {

  secret: "fulstackdev-secret-key"};
```

You can create your own `secret` String.

Create Middleware functions

To verify a Signup action, we need 2 functions:

- check if `username` or `email` is duplicate or not

- check if `roles` in the request is existed or not

middleware/verifySignUp.js

```
const db = require("../models");const ROLES = db.ROLES;const User =

db.user;checkDuplicateUsernameOrEmail = (req, res, next) => {

  // Username

  User.findOne({

    where: {

      username: req.body.username

    }

  }).then(user => {

    if (user) {

      res.status(400).send({

        message: "Failed! Username is already in use!"

      });

      return;

    }
```

```
  // Email
  User.findOne({
    where: {
      email: req.body.email
    }
  }).then(user => {
    if (user) {
      res.status(400).send({
        message: "Failed! Email is already in use!"
      });
      return;
    }
    next();
  });
});};checkRolesExisted = (req, res, next) => {
if (req.body.roles) {
  for (let i = 0; i < req.body.roles.length; i++) {
    if (!ROLES.includes(req.body.roles[i])) {
      res.status(400).send({
```

```
            message: "Failed! Role does not exist = " + req.body.roles[i]

          });

          return;

        }

      }

    }



  next();};const verifySignUp = {

  checkDuplicateUsernameOrEmail: checkDuplicateUsernameOrEmail,

  checkRolesExisted: checkRolesExisted};

module.exports = verifySignUp;
```

To process Authentication & Authorization, we have these functions:

- check if `token` is provided, legal or not. We get token from **x-access-token** of HTTP headers, then use **jsonwebtoken**'s `verify()` function.
- check if `roles` of the user contains required role or not.

middleware/authJwt.js

middleware/authJwt.js

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

require("../config/auth.config.js");const db = require("../models");const User =

db.user;verifyToken = (req, res, next) => {

  let token = req.headers["x-access-token"];

  if (!token) {

    return res.status(403).send({

      message: "No token provided!"

    });

  }

  jwt.verify(token, config.secret, (err, decoded) => {

    if (err) {

      return res.status(401).send({

        message: "Unauthorized!"

      });

    }

    req.userId = decoded.id;

    next();

  });};isAdmin = (req, res, next) => {

  User.findByPk(req.userId).then(user => {
```

```
  user.getRoles().then(roles => {
    for (let i = 0; i < roles.length; i++) {
      if (roles[i].name === "admin") {
        next();
        return;
      }
    }
    res.status(403).send({
      message: "Require Admin Role!"
    });
    return;
  });
});};isModerator = (req, res, next) => {
User.findByPk(req.userId).then(user => {
  user.getRoles().then(roles => {
    for (let i = 0; i < roles.length; i++) {
      if (roles[i].name === "moderator") {
        next();
        return;
```

```
        }

      }

      res.status(403).send({

        message: "Require Moderator Role!"

      });

    });

});};isModeratorOrAdmin = (req, res, next) => {

User.findByPk(req.userId).then(user => {

    user.getRoles().then(roles => {

      for (let i = 0; i < roles.length; i++) {

        if (roles[i].name === "moderator") {

          next();

          return;

        }

        if (roles[i].name === "admin") {

          next();

          return;

        }

      }
```

```
        res.status(403).send({
            message: "Require Moderator or Admin Role!"
        });
    });
  });};const authJwt = {
  verifyToken: verifyToken,
  isAdmin: isAdmin,
  isModerator: isModerator,
  isModeratorOrAdmin: isModeratorOrAdmin};
module.exports = authJwt;
```

middleware/index.js

```
const authJwt = require("./authJwt");const verifySignUp = require("./verifySignUp");
module.exports = {
  authJwt,
  verifySignUp};
```

Create Controllers

Controller for Authentication

There are 2 main functions for Authentication:

- `signup`: create new User in database (role is **user** if not specifying role)
- `signin`:

  - find `username` of the request in database, if it exists
  - compare `password` with `password` in database using **bcrypt**, if it is correct
  - generate a token using **jsonwebtoken**
  - return user information & access Token

controllers/auth.controller.js

```
const db = require("../models");const config = require("../config/auth.config");const User = db.user;const Role = db.role;const Op = db.Sequelize.Op;var jwt = require("jsonwebtoken");var bcrypt = require("bcryptjs");

exports.signup = (req, res) => {
  // Save User to Database
  User.create({
    username: req.body.username,
    email: req.body.email,
    password: bcrypt.hashSync(req.body.password, 8)
  })
```

```
.then(user => {
  if (req.body.roles) {
    Role.findAll({
      where: {
        name: {
          [Op.or]: req.body.roles
        }
      }
    }).then(roles => {
      user.setRoles(roles).then(() => {
        res.send({ message: "User was registered successfully!" });
      });
    });
  } else {
    // user role = 1
    user.setRoles([1]).then(() => {
      res.send({ message: "User was registered successfully!" });
    });
  }
```

```
    })
    .catch(err => {
      res.status(500).send({ message: err.message });
    });};
exports.signin = (req, res) => {
  User.findOne({
    where: {
      username: req.body.username
    }
  })
    .then(user => {
      if (!user) {
        return res.status(404).send({ message: "User Not found." });
      }
      var passwordIsValid = bcrypt.compareSync(
        req.body.password,
        user.password
      );
      if (!passwordIsValid) {
```

```
    return res.status(401).send({
      accessToken: null,
      message: "Invalid Password!"
    });
  }
  var token = jwt.sign({ id: user.id }, config.secret, {
    expiresIn: 86400 // 24 hours
  });
  var authorities = [];
  user.getRoles().then(roles => {
    for (let i = 0; i < roles.length; i++) {
      authorities.push("ROLE_" + roles[i].name.toUpperCase());
    }
    res.status(200).send({
      id: user.id,
      username: user.username,
      email: user.email,
      roles: authorities,
      accessToken: token
```

```
        });

      });

    })

    .catch(err => {

      res.status(500).send({ message: err.message });

    });};
```

Controller for testing Authorization

There are 4 functions:

- `/api/test/all` for public access
- `/api/test/user` for loggedin users

(role: **user**/**moderator**/**admin**)

- `/api/test/mod` for users having **moderator** role
- `/api/test/admin` for users having **admin** role

controllers/user.controller.js

```
exports.allAccess = (req, res) => {

  res.status(200).send("Public Content.");};

exports.userBoard = (req, res) => {

  res.status(200).send("User Content.");};
```

```
exports.adminBoard = (req, res) => {

  res.status(200).send("Admin Content.");};

exports.moderatorBoard = (req, res) => {

  res.status(200).send("Moderator Content.");};
```

Now, do you have any question? Would you like to know how we can combine middlewares with controller functions? Let's do it in the next section.

Define Routes

When a client sends request for an endpoint using HTTP request (GET, POST, PUT, DELETE), we need to determine how the server will response by setting up the routes.

We can separate our routes into 2 part: for Authentication and for Authorization (accessing protected resources).

Authentication:

- POST `/api/auth/signup`
- POST `/api/auth/signin`

routes/auth.routes.js

```
const { verifySignUp } = require("../middleware");const controller =

require("../controllers/auth.controller");

module.exports = function(app) {

  app.use(function(req, res, next) {

    res.header(

      "Access-Control-Allow-Headers",

      "x-access-token, Origin, Content-Type, Accept"

    );

    next();

  });

  app.post(

    "/api/auth/signup",

    [

      verifySignUp.checkDuplicateUsernameOrEmail,

      verifySignUp.checkRolesExisted

    ],

    controller.signup

  );

  app.post("/api/auth/signin", controller.signin);};
```

Authorization:

- GET `/api/test/all`
- GET `/api/test/user` for loggedin users (user/moderator/admin)
- GET `/api/test/mod` for moderator
- GET `/api/test/admin` for admin

routes/user.routes.js

```
const { authJwt } = require("../middleware");const controller =
require("../controllers/user.controller");
module.exports = function(app) {
  app.use(function(req, res, next) {
    res.header(
      "Access-Control-Allow-Headers",
      "x-access-token, Origin, Content-Type, Accept"
    );
    next();
  });
  app.get("/api/test/all", controller.allAccess);
  app.get(
```

```
    "/api/test/user",
    [authJwt.verifyToken],
    controller.userBoard
  );
  app.get(
    "/api/test/mod",
    [authJwt.verifyToken, authJwt.isModerator],
    controller.moderatorBoard
  );
  app.get(
    "/api/test/admin",
    [authJwt.verifyToken, authJwt.isAdmin],
    controller.adminBoard
  );};
```

Don't forget to add these routes in 聽server.js:

```
...// routes
require('./app/routes/auth.routes')(app);
require('./app/routes/user.routes')(app);// set port, listen for requests...
```

Run & Test with Results

Run Node.js application with command: `node server.js`

Tables that we define in models package will be automatically generated in PostgreSQL Database.

Run & Test with Results

Run Node.js application with command: `node server.js`

Tables that we define in models package will be automatically generated in PostgreSQL Database.
If you check the database, you can see things like this:

```
testdb=# \d users

                                    Table "public.users"

  Column    |            Type             |
Modifiers-----------+-------------------------+------------------
--------------------------------

 id          | integer                     | not null default
nextval('users_id_seq'::regclass)

 username   | character varying(255)    |

 email       | character varying(255)    |
```

```
 password   | character varying(255)     |
 createdAt | timestamp with time zone | not null
 updatedAt | timestamp with time zone | not null
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "user_roles" CONSTRAINT "user_roles_userId_fkey" FOREIGN KEY
("userId") REFERENCES users(id) ON UPDATE CASCADE ON DELETE CASCADE


testdb=# \d roles
                    Table "public.roles"
  Column    |              Type              |
Modifiers-----------+------------------------+-----------
 id          | integer                        | not null
 name        | character varying(255)     |
 createdAt | timestamp with time zone | not null
 updatedAt | timestamp with time zone | not null
Indexes:
    "roles_pkey" PRIMARY KEY, btree (id)
```

```
Referenced by:
    TABLE "user_roles" CONSTRAINT "user_roles_roleId_fkey" FOREIGN KEY
("roleId") REFERENCES roles(id) ON UPDATE CASCADE ON DELETE CASCADE



testdb=# \d user_roles
                Table "public.user_roles"
  Column    |              Type              |
Modifiers-----------+-------------------------+-----------
 createdAt | timestamp with time zone | not null
 updatedAt | timestamp with time zone | not null
 roleId    | integer                  | not null
 userId    | integer                  | not null
Indexes:
    "user_roles_pkey" PRIMARY KEY, btree ("roleId", "userId")Foreign-key
constraints:
    "user_roles_roleId_fkey" FOREIGN KEY ("roleId") REFERENCES roles(id) ON
UPDATE CASCADE ON DELETE CASCADE
    "user_roles_userId_fkey" FOREIGN KEY ("userId") REFERENCES users(id) ON
UPDATE CASCADE ON DELETE CASCADE
```

```
testdb=# select * from roles;

 id |     name      |            createdAt            |
updatedAt----+-----------+----------------------------+-----------
----------------

  1 | user          | 2020-11-19 21:09:51.826+07 | 2020-11-19 21:09:51.826+07

  2 | moderator | 2020-11-19 21:09:51.828+07 | 2020-11-19 21:09:51.828+07

  3 | admin         | 2020-11-19 21:09:51.828+07 | 2020-11-19 21:09:51.828+07(

3 rows)
```

Register some users with `/signup` API:

- **admin** with `admin` role
- **mod** with `moderator` and `user` roles
- **zkoder** with `user` role

POST http://localhost:8080/api/auth/signup Send

Params Auth Headers (10) Body Pre-req. Tests Settings

raw JSON

```
{
    "username": "mod",
    "email": "mod@bezkoder.com",
    "password": "12345678",
    "roles": ["user", "moderator"]
}
```

Body 200 OK 1667 ms 397 B

Pretty Raw Preview Visualize JSON

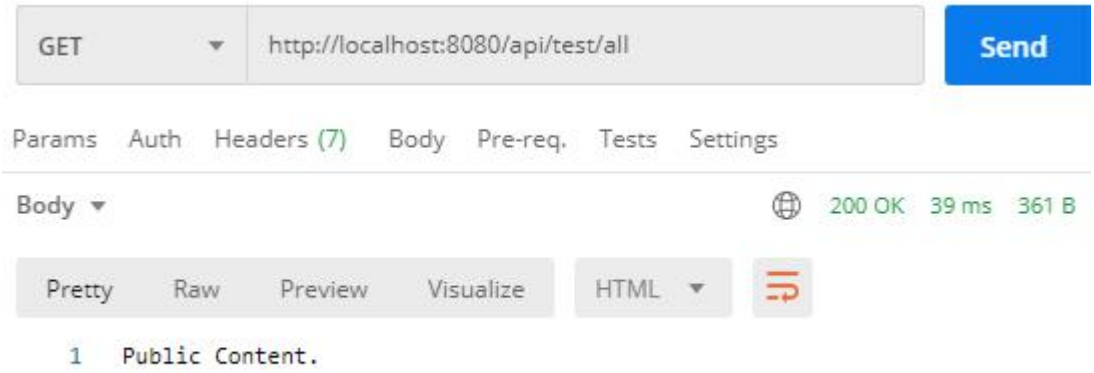
```
{
    "message": "User registered successfully!"
}
```

Our tables after registration could look like this.

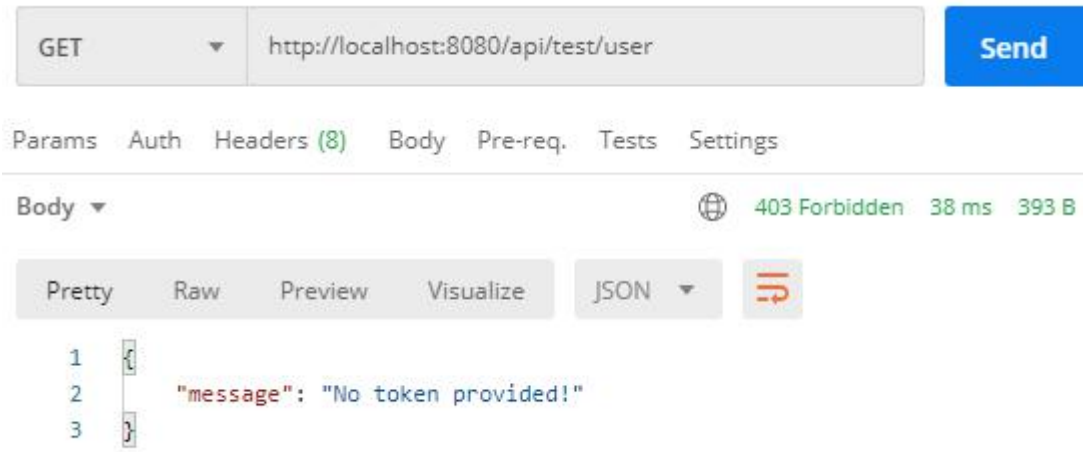
```
testdb=# select * from users;

 id | username |          email           |                              password
|            createdAt              |
updatedAt----+----------+--------------------+--------------------
----------------------------------------+---------------------
------+----------------------------

  1 | admin    | admin@bezkoder.com |
$2a$08$T0B0i/96KE90jAYPOhpsN.vJGVPMfFw.FbxljzuQkkN4ZK3YauRLq |
2020-11-19 21:20:49.305+07 | 2020-11-19 21:20:49.305+07
```

```
  2 | mod         | mod@bezkoder.com     |
$2a$08$CmCiT5Y/9OTUM0ofSP2r2eQSHVlcqhjp1wH.GYA5oPcRlJ7Hr2C66 |
2020-11-19 21:21:13.67+07   | 2020-11-19 21:21:13.67+07
  3 | user        | user@bezkoder.com    |
$2a$08$f.exOM3efA4DF4BtohzhAOzcv2.iCppJlbdSHFLRmka569sCNXfSe |
2020-11-19 21:23:00.978+07 | 2020-11-19 21:23:00.978+07(3 rows)


testdb=# select * from user_roles;
            createdAt              |              updatedAt              | roleId |
userId----------------------------+----------------------------+--
------+--------
 2020-11-19 21:20:50.045+07 | 2020-11-19 21:20:50.045+07 |        3 |        1
 2020-11-19 21:21:14.604+07 | 2020-11-19 21:21:14.604+07 |        1 |        2
 2020-11-19 21:21:14.604+07 | 2020-11-19 21:21:14.604+07 |        2 |        2
 2020-11-19 21:23:02.1+07     | 2020-11-19 21:23:02.1+07     |        1 |        3(
4 rows)
```

Access public resource: GET `/api/test/all`



Access protected resource: GET `/api/test/user`



Login an account (with wrong password): POST `/api/auth/signin`

POST http://localhost:8080/api/auth/signin Send

Params Auth Headers (10) Body Pre-req. Tests Settings

raw JSON

```
1 {
2     "username": "mod",
3     "password": "123456789"
4 }
```

Body 401 Unauthorized 189 ms 414 B

Pretty Raw Preview Visualize JSON

```
1 {
2     "accessToken": null,
3     "message": "Invalid Password!"
4 }
```

Login a correct account: POST `/api/auth/signin`

POST http://localhost:8080/api/auth/signin Send

Params Auth Headers (10) Body Pre-req. Tests Settings

raw JSON

```
1 {
2     "username": "mod",
3     "password": "12345678"
4 }
```

Body 200 OK 713 ms 600 B

Pretty Raw Preview Visualize JSON


```
1 {
2     "id": 2,
3     "username": "mod",
4     "email": "mod@bezkoder.com",
5     "roles": [
6         "ROLE_USER",
7         "ROLE_MODERATOR"
8     ],
9     "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
          eyJpZCI6MiwiaWF0IjoxNjA1Nzk2MTE3LCJleHAiOjE2MDU4ODI1MTd9.
          8snpzSVc3Zh6pgvsICg4sq5cpd6ITMVRGh5l6-JNw8I"
10 }
```


Access protected resources:

- GET `/api/test/user`
- GET `/api/test/mod`
- GET `/api/test/admin`

GET http://localhost:8080/api/test/user Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

Body 200 OK 39 ms 359 B

Pretty Raw Preview Visualize HTML

```
1  User Content.
```

GET http://localhost:8080/api/test/mod Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

Body 200 OK 570 ms 365 B

Pretty Raw Preview Visualize HTML

```
1  Moderator Content.
```

GET http://localhost:8080/api/test/admin Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

GET http://localhost:8080/api/test/user Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

Body 200 OK 39 ms 359 B

Pretty Raw Preview Visualize HTML

```
1  User Content.
```

GET http://localhost:8080/api/test/mod Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

Body 200 OK 570 ms 365 B

Pretty Raw Preview Visualize HTML

```
1  Moderator Content.
```

GET http://localhost:8080/api/test/admin Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

Congratulation!

Today we've learned so many interesting things about Node.js JWT (JSONWebToken) Authentication & Authorization example with PostgreSQL database.
Despite we wrote a lot of code, I hope you will understand the overall architecture of the application, and apply it in your project at ease.

----------Section B----------

Overview of JWT Refresh Token with Node.js example

We already have a Node.js Express JWT Authentication and Authorization application with MySQL/PostgreSQL in that:

- User can signup new account, or login with username & password.
- By User's role (admin, moderator, user), we authorize the User to access resources

With APIs:

Methods	Urls	Actions
POST	/api/auth/signup	signup new account
POST	/api/auth/signin	login an account
GET	/api/test/all	retrieve public content
GET	/api/test/user	access User's content
GET	/api/test/mod	access Moderator's content
GET	/api/test/admin	access Admin's content

GET http://localhost:8080/api/test/user Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

Body 200 OK 39 ms 359 B

Pretty Raw Preview Visualize HTML

```
1  User Content.
```

GET http://localhost:8080/api/test/mod Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

Body 200 OK 570 ms 365 B

Pretty Raw Preview Visualize HTML

```
1  Moderator Content.
```

GET http://localhost:8080/api/test/admin Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

GET http://localhost:8080/api/test/user Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

Body 200 OK 39 ms 359 B

Pretty Raw Preview Visualize HTML

```
1  User Content.
```

GET http://localhost:8080/api/test/mod Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

Body 200 OK 570 ms 365 B

Pretty Raw Preview Visualize HTML

```
1  Moderator Content.
```

GET http://localhost:8080/api/test/admin Send

Params Auth Headers (8) Body Pre-req. Tests Settings

Headers 7 hidden

	KEY	VALUE	DE ••• Bul
☑	x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX...	
	Key	Value	Description

We’re gonna add Token Refresh to this Node.js & JWT Project.

The final result can be described with following requests/responses:

– Send `/signin` request, return response with `refreshToken`.

POST http://localhost:8080/api/auth/signin ...

Params Auth Headers (10) Body Pre-req. Tests Settings

raw JSON

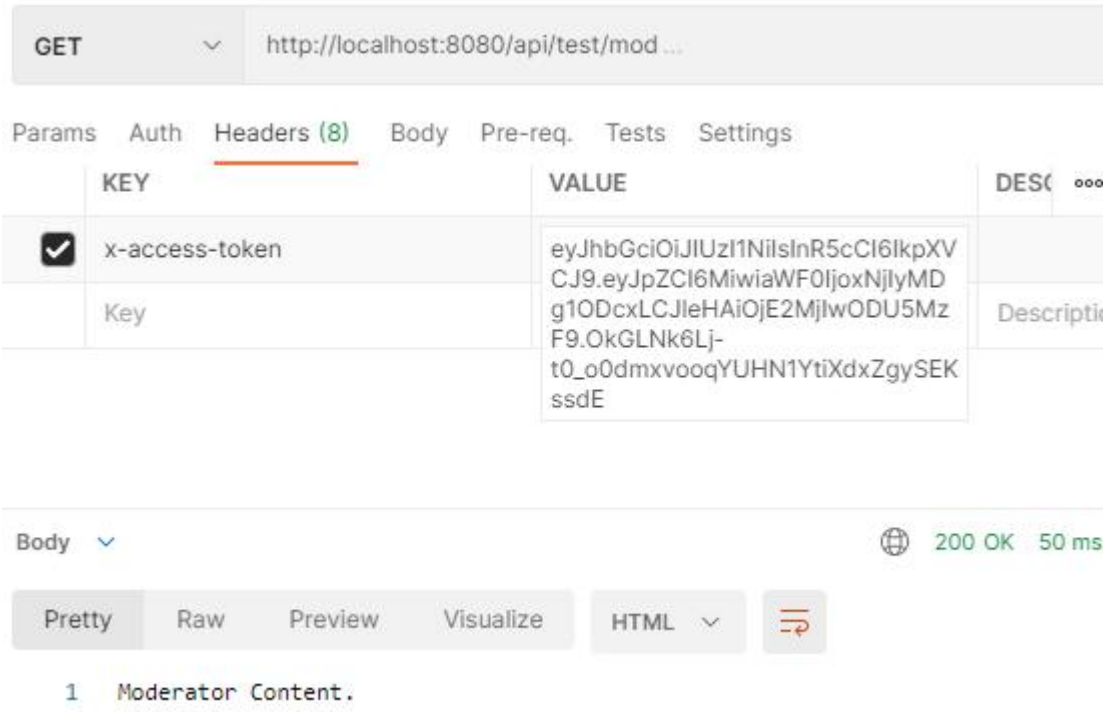
```
1 {
2     "username": "mod",
3     "password": "12345678"
4 }
```

Body 200 OK 96 ms

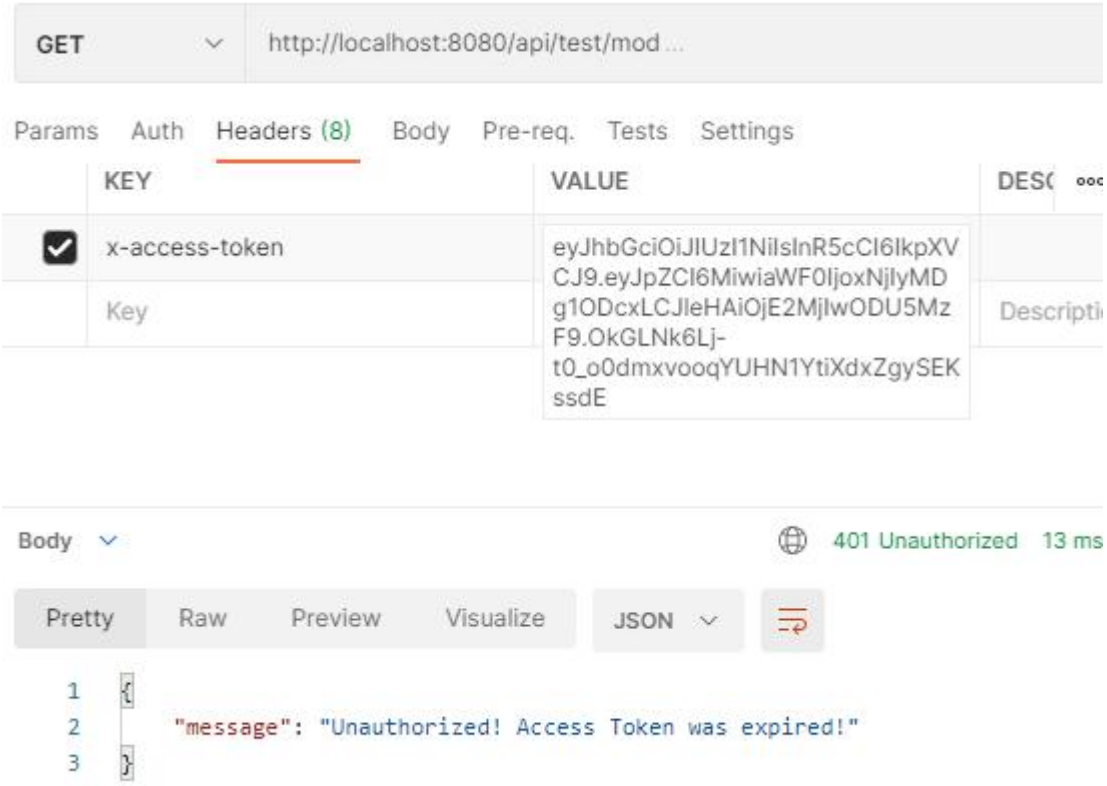
Pretty Raw Preview Visualize JSON

```
1 {
2     "id": 2,
3     "username": "mod",
4     "email": "mod@bezkoder.com",
5     "roles": [
6         "ROLE_USER",
7         "ROLE_MODERATOR"
8     ],
9     "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
          eyJpZCI6MiwiaWF0IjoxNjIyMDg1ODcxLCJleHAiOjE2MjIwODU5MzF9.
          OkGLNk6Lj-t0_o0dmxvooqYUHN1YtiXdxZgySEKssdE",
10    "refreshToken": "f5ec4710-9450-47e0-a202-7a1e22cd2c3c"
11 }
```

– Access resource successfully with `accessToken`



– When the `accessToken` is expired, user cannot use it anymore.



– Send `/refreshtoken` request, return response with new `accessToken`.

POST http://localhost:8080/api/auth/refreshtoken

Params Auth Headers (9) Body Pre-req. Tests Settings

raw JSON

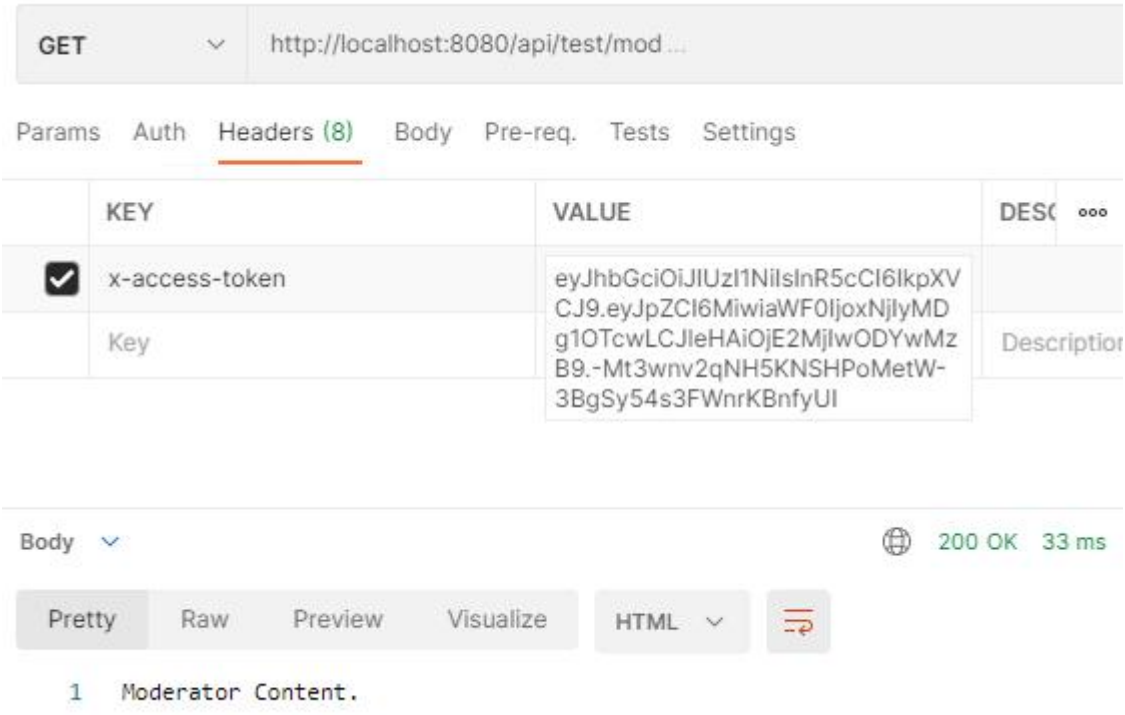
```
1 {
2     "refreshToken": "f5ec4710-9450-47e0-a202-7a1e22cd2c3c"
3 }
```

Body 200 OK 80 ms

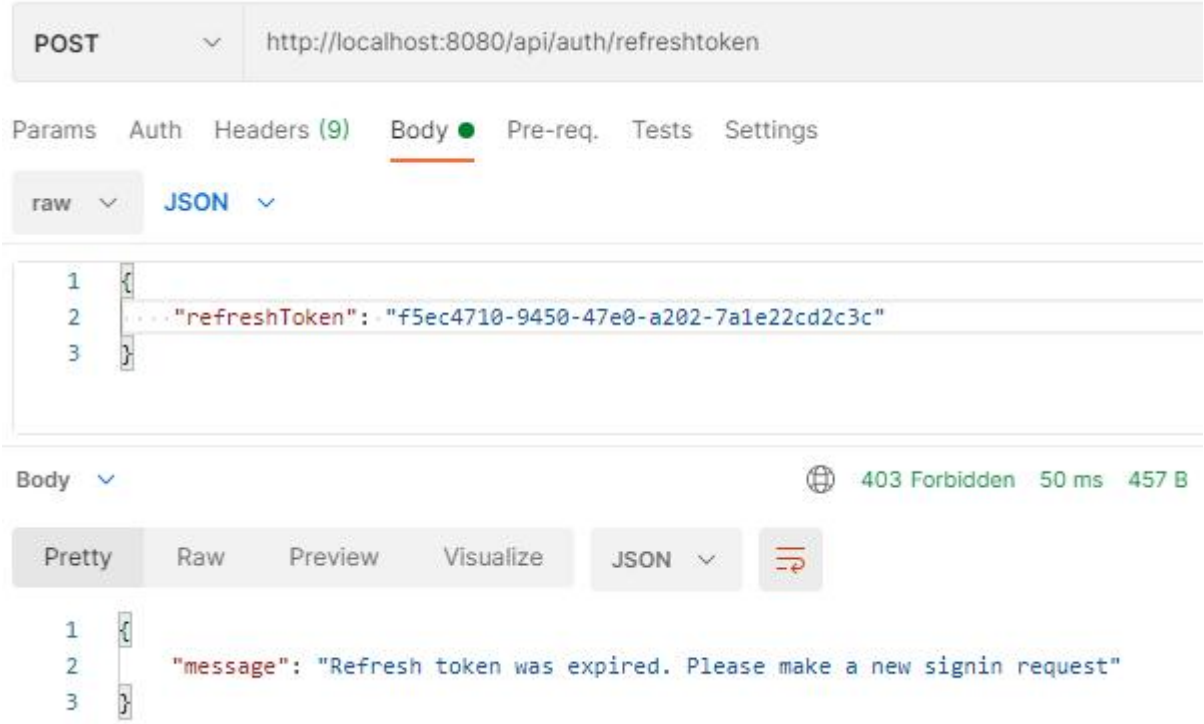
Pretty Raw Preview Visualize JSON


```
1 {
2     "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
          eyJpZCI6MiwiaWF0IjoxNjIyMDg1OTcwLCJleHAiOjE2MjIwODYwMzB9.
          -Mt3wnv2qNH5KNSHPoMetW-3BgSy54s3FWnrKBnfyUI",
3     "refreshToken": "f5ec4710-9450-47e0-a202-7a1e22cd2c3c"
4 }
```

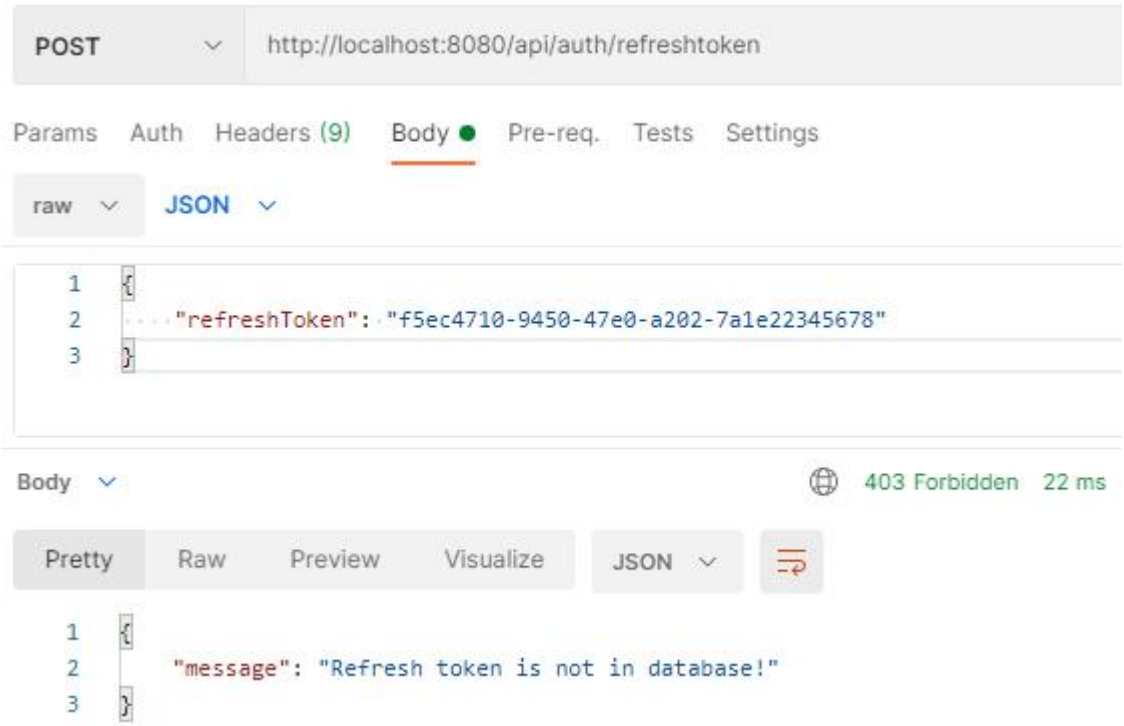
– Access resource successfully with new `accessToken`.



– Send an **expired** Refresh Token.

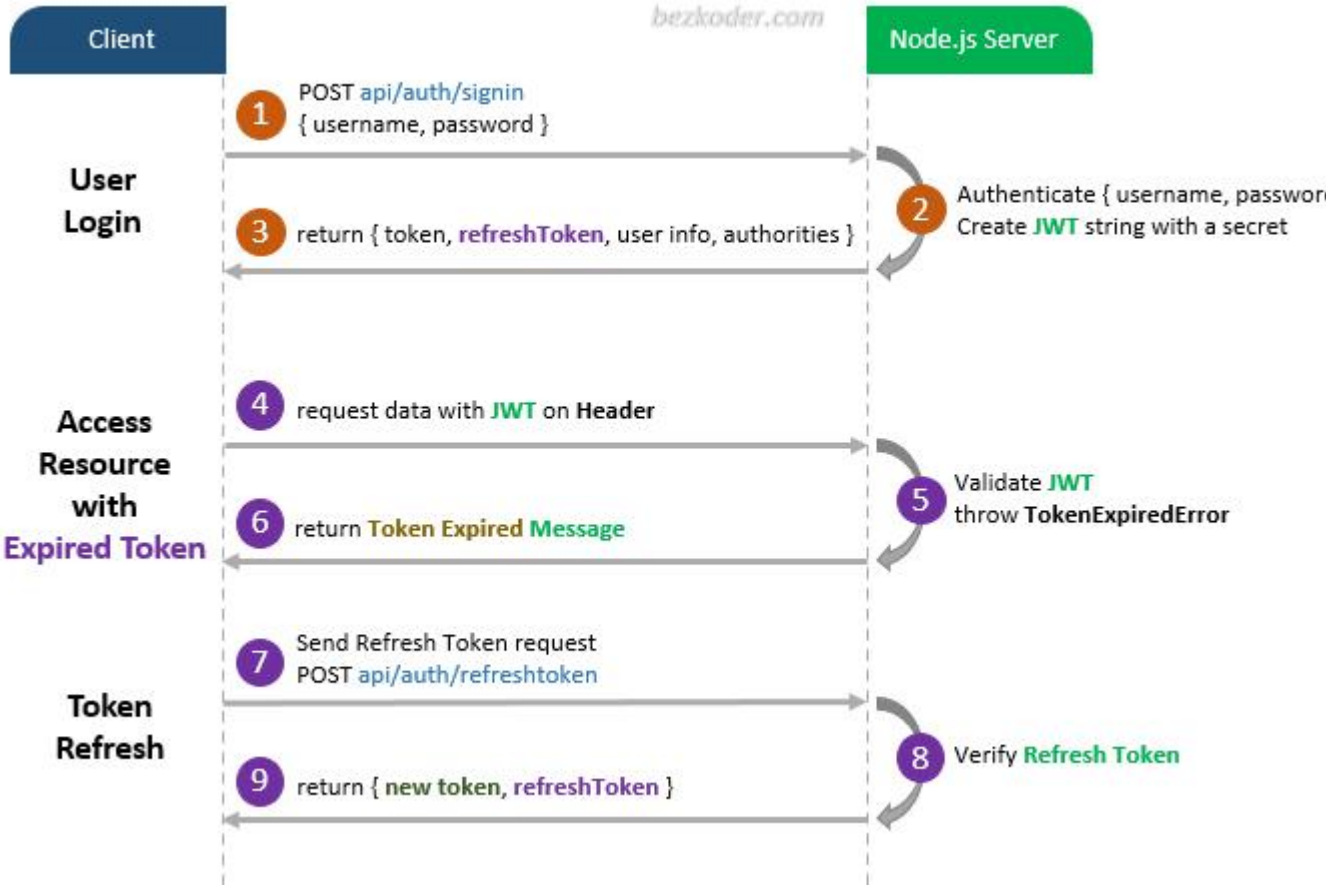


– Send an inexistent Refresh Token.



Flow for JWT Refresh Token implementation

The diagram shows flow of how we implement Authentication process with Access Token and Refresh Token.



– A legal `JWT` must be added to HTTP Header if Client accesses protected resources.

– A `refreshToken` will be provided at the time user signs in.

How to Expire JWT Token in Node.js

The Refresh Token has different value and expiration time to the Access Token.

Regularly we configure the expiration time of Refresh Token longer than Access Token's.

Open **config**/auth.config.js:

```
module.exports = {

  secret: "bezkoder-secret-key",

  jwtExpiration: 3600,                // 1 hour

  jwtRefreshExpiration: 86400,     // 24 hours

  /* for test */

  // jwtExpiration: 60,               // 1 minute

  // jwtRefreshExpiration: 120,   // 2 minutes};
```

Update **middlewares**/authJwt.js file to catch `TokenExpiredError` in `verifyToken()` function.

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

require("../config/auth.config.js");...const { TokenExpiredError } = jwt;const

catchError = (err, res) => {

  if (err instanceof TokenExpiredError) {

    return res.status(401).send({ message: "Unauthorized! Access Token was

expired!" });

  }

  return res.sendStatus(401).send({ message: "Unauthorized!" });}const verifyToken =

(req, res, next) => {
```

```
let token = req.headers["x-access-token"];

if (!token) {

  return res.status(403).send({ message: "No token provided!" });

}

jwt.verify(token, config.secret, (err, decoded) => {

  if (err) {

    return catchError(err, res);

  }

  req.userId = decoded.id;

  next();

});};
```

Create Refresh Token Model

This Sequelize model has one-to-one relationship with `User` model. It contains `expiryDate` field which value is set by adding `config.jwtRefreshExpiration` value above.

There are 2 static methods:

- `createToken`: use `uuid` library for creating a random token and save new object into PostgreSQL database
- `verifyExpiration`: compare `expiryDate` with current Date time to check the expiration

models/refreshToken.model.js

```
const config = require("../config/auth.config");const { v4: uuidv4 } = require("uuid");

module.exports = (sequelize, Sequelize) => {

  const RefreshToken = sequelize.define("refreshToken", {

    token: {

      type: Sequelize.STRING,

    },

    expiryDate: {

      type: Sequelize.DATE,

    },

  });

  RefreshToken.createToken = async function (user) {

    let expiredAt = new Date();

    expiredAt.setSeconds(expiredAt.getSeconds() + config.jwtRefreshExpiration);

    let _token = uuidv4();
```

```
    let refreshToken = await this.create({

      token: _token,

      userId: user.id,

      expiryDate: expiredAt.getTime(),

    });

    return refreshToken.token;

  };

  RefreshToken.verifyExpiration = (token) => {

    return token.expiryDate.getTime() < new Date().getTime();

  };

  return RefreshToken;};
```

Don’t forget to use `belongsTo()` and `hasOne()` for configure association with `User` model.

Then export `RefreshToken` model in **models**/index.js:

```
const config = require("../config/db.config.js");const Sequelize =

require("sequelize");const sequelize = new Sequelize( ... );const db = {};

db.Sequelize = Sequelize;

db.sequelize = sequelize;
```

```
db.user = require("../models/user.model.js")(sequelize, Sequelize);

db.role = require("../models/role.model.js")(sequelize, Sequelize);

db.refreshToken = require("../models/refreshToken.model.js")(sequelize, Sequelize);

db.role.belongsToMany(db.user, {

  through: "user_roles",

  foreignKey: "roleId",

  otherKey: "userId"});

db.user.belongsToMany(db.role, {

  through: "user_roles",

  foreignKey: "userId",

  otherKey: "roleId"});

db.refreshToken.belongsTo(db.user, {

  foreignKey: 'userId', targetKey: 'id'});

db.user.hasOne(db.refreshToken, {

  foreignKey: 'userId', targetKey: 'id'});

db.ROLES = ["user", "admin", "moderator"];

module.exports = db;
```

Node.js Express Rest API for JWT Refresh Token

Let's update the payloads for our Rest APIs:

– Requests:

- { **refreshToken** }

– Responses:

- Signin Response: { accessToken, **refreshToken**, id, username, email, roles }
- Message Response: { message }
- RefreshToken Response: { new **accessToken**, **refreshToken** }

In the `Auth` Controller, we:

- update the method for `/signin` endpoint with Refresh Token
- expose the POST API for creating new Access Token from received Refresh Token

controllers/auth.controller.js

```
const db = require("../models");const config = require("../config/auth.config");const
{ user: User, role: Role, refreshToken: RefreshToken } = db;const jwt =
require("jsonwebtoken");const bcrypt = require("bcryptjs");...
exports.signin = (req, res) => {
  User.findOne({
    where: {
      username: req.body.username
    }
  })
    .then(async (user) => {
      if (!user) {
        return res.status(404).send({ message: "User Not found." });
      }
      const passwordIsValid = bcrypt.compareSync(
        req.body.password,
        user.password
      );
      if (!passwordIsValid) {
        return res.status(401).send({
```

```
        accessToken: null,
        message: "Invalid Password!"
    });
}
const token = jwt.sign({ id: user.id }, config.secret, {
    expiresIn: config.jwtExpiration
});
let refreshToken = await RefreshToken.createToken(user);
let authorities = [];
user.getRoles().then(roles => {
    for (let i = 0; i < roles.length; i++) {
        authorities.push("ROLE_" + roles[i].name.toUpperCase());
    }
    res.status(200).send({
        id: user.id,
        username: user.username,
        email: user.email,
        roles: authorities,
        accessToken: token,
```

```
            refreshToken: refreshToken,

          });

        });

      })

      .catch(err => {

        res.status(500).send({ message: err.message });

      });};

exports.refreshToken = async (req, res) => {

  const { refreshToken: requestToken } = req.body;

  if (requestToken == null) {

    return res.status(403).json({ message: "Refresh Token is required!" });

  }

  try {

    let refreshToken = await RefreshToken.findOne({ where: { token:
requestToken } });

    console.log(refreshToken)

    if (!refreshToken) {

      res.status(403).json({ message: "Refresh token is not in database!" });

      return;
```

```
    }

    if (RefreshToken.verifyExpiration(refreshToken)) {

      RefreshToken.destroy({ where: { id: refreshToken.id } });



      res.status(403).json({

        message: "Refresh token was expired. Please make a new signin request",

      });

      return;

    }

    const user = await refreshToken.getUser();

    let newAccessToken = jwt.sign({ id: user.id }, config.secret, {

      expiresIn: config.jwtExpiration,

    });

    return res.status(200).json({

      accessToken: newAccessToken,

      refreshToken: refreshToken.token,

    });

  } catch (err) {

    return res.status(500).send({ message: err });
```

```
}};
```

In `refreshToken()` function:

- Firstly, we get the Refresh Token from request data
- Next, get the `RefreshToken` object {`id`, `user`, `token`, `expiryDate`} from raw Token using `RefreshToken` model static method
- We verify the token (expired or not) basing on `expiryDate` field. If the Refresh Token was expired, remove it from database and return message
- Continue to use user `id` field of `RefreshToken` object as parameter to generate new Access Token using `jsonwebtoken` library
- Return { new `accessToken`, `refreshToken` } if everything is done
- Or else, send error message

Define Route for JWT Refresh Token API

Finally, we need to determine how the server with an endpoint will response by setting up the routes.

In **routes**/auth.routes.js, add one line of code:

```
...const controller = require("../controllers/auth.controller");

module.exports = function(app) {

  ...

  app.post("/api/auth/refreshtoken", controller.refreshToken);};
```

Conclusion

Today we've learned JWT Refresh Token implementation in Node.js Rest Api example using Express, Sequelize and MySQL or PostgreSQL. You also know how to expire the JWT Token and renew the Access Token.