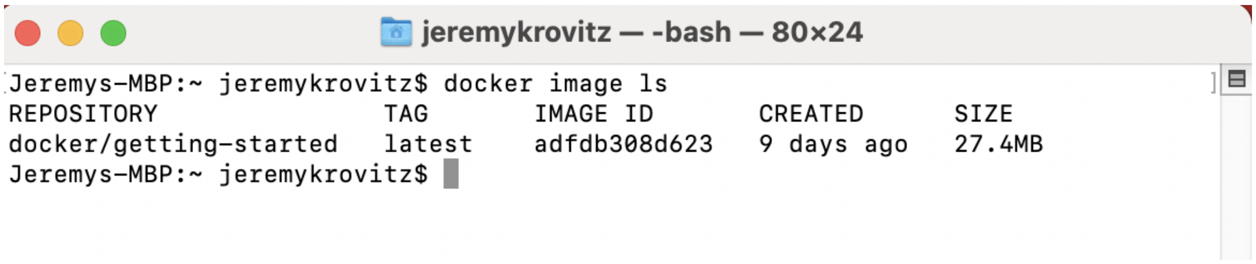
# Introduction to Sequelize ORM (w/ Express.js + Postgres)

# Section 1: Introduction

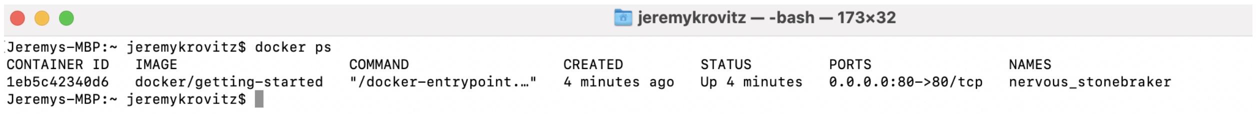
## 6. Necessary tools - Docker and Docker Compose

Type `docker image ls` in the terminal to see all docker images.



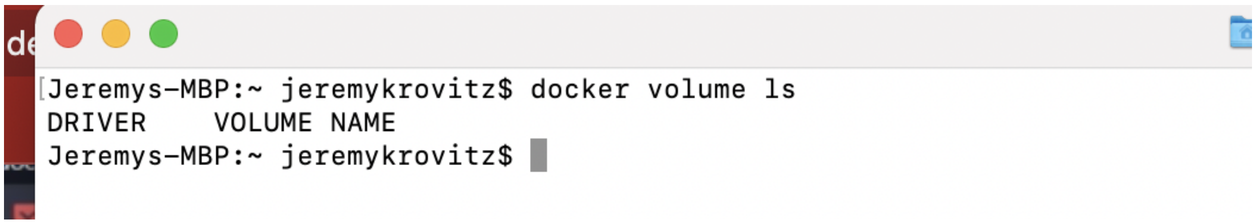
```
Jeremys-MBP:~ jeremykrovitz$ docker image ls
REPOSITORY               TAG       IMAGE ID       CREATED       SIZE
docker/getting-started   latest    adfdb308d623   9 days ago    27.4MB
Jeremys-MBP:~ jeremykrovitz$
```

Type `docker ps` in the terminal to see the containers that are running.



```
Jeremys-MBP:~ jeremykrovitz$ docker ps
CONTAINER ID   IMAGE                     COMMAND                  CREATED         STATUS         PORTS                NAMES
1eb5c42340d6   docker/getting-started    "/docker-entrypoint.…"   4 minutes ago   Up 4 minutes   0.0.0.0:80->80/tcp   nervous_stonebraker
Jeremys-MBP:~ jeremykrovitz$
```

Type `docker volume ls` in the terminal to see all of the volumes.

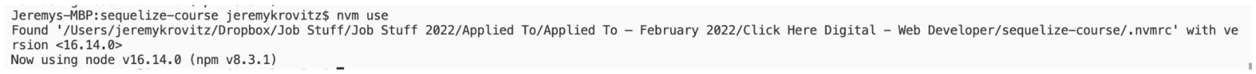


```
Jeremys-MBP:~ jeremykrovitz$ docker volume ls
DRIVER    VOLUME NAME
Jeremys-MBP:~ jeremykrovitz$
```

# Section 2

## 10. Initial setup

Add the version of `Node` that you want to use to the `.nvmrc` file.
Each time you open the folder, you then type in the terminal `nvm use`, which will then set the `Node` version from the `.nvmrc` file.



```
Jeremys-MBP:sequelize-course jeremykrovitz$ nvm use
Found '/Users/jeremykrovitz/Dropbox/Job Stuff/Job Stuff 2022/Applied To/Applied To - February 2022/Click Here Digital - Web Developer/sequelize-course/.nvmrc' with ve
rsion <16.14.0>
Now using node v16.14.0 (npm v8.3.1)
```

You can add the following to the `.zshrc` file to automatically load the `Node` version but this only works with the .zsh shell:

```
# AUTOMATIC NVM USE
autoload -U add-zsh-hook
load-nvmrc() {
 local node_version="$(nvm version)"
 local nvmrc_path="$(nvm_find_nvmrc)"
 if [ -n "$nvmrc_path" ]; then
   local nvmrc_node_version=$(nvm version "$(cat "${nvmrc_path}")")

   if [ "$nvmrc_node_version" = "N/A" ]; then
     nvm install
   elif [ "$nvmrc_node_version" != "$node_version" ]; then
     nvm use
   fi
 elif [ "$node_version" != "$(nvm version default)" ]; then
   echo "Reverting to nvm default version"
   nvm use default
 fi
}
add-zsh-hook chpwd load-nvmrc
load-nvmrc
```

## To create `package.json` file

To create `package.json` file, run the following command in the terminal:
`npm init -y`

Running this command for this project produced the following `package.json` file:

```
{
 "name": "sequelize-course",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
   "test": "echo \"Error: no test specified\" && exit 1"
 },
 "keywords": [],
 "author": "",
 "license": "ISC"
}
```

## 11. Adding all the dependencies

### Installing the Dev dependencies

In the command `npm install -D`, the `-D` indicates that it is a Dev dependency. We need this command `npm install -D @babel/cli @babel/core @babel/node @babel/preset-env` in order to use ES6 modules.

- We also need to install `jest`, which is our testing framework as well as `@types/jest`, which is needed because you may be testing and `toEqual` may not be available and `@types/jest` makes it available.
- We need to install `supertest` in order to make fake requests to our Express App.
- We also need to install `nodemon`, so we can start our app automatically when we are developing.

We install the four dependencies in the following manner:

```
npm install -D jest @types/jest supertest nodemon
```

### Installing the "real" dependencies

We are going to following real dependencies:

- `bcrypt` - is an algorithm, and this algorithm is going to help us to hash unsolved our passwords (this will be explained later)
- `sequelize`
- `sequelize-cli` - the cli for sequelize
- `jsonwebtoken` - the library to generate and verify json web tokens; we are going to use access tokens and refresh tokens and both of them our going to be json web tokens
- `pg` - the driver for PostgreSQL
- `express` - that's the framework we're going to use
- `dotenv` - in order to be able to use environment variables
- `morgan` - this is for logging requests; very useful in production
- `cls-hooked` - helps us automatically manage transactions for relational dbs

We use the following command to install all of our "real" dependencies, the dependencies used in production:

```
npm install bcrypt sequelize sequelize-cli jsonwebtoken pg
express dotenv morgan cls-hooked
```

## 12. Configuring Babel, Jest and Nodemon

To configure babel we will create a file called `babel.config.js`:
```
touch babel.config.js
```

To configure jest, we will create a file called `jest.config.js`:
```
touch jest.config.js
```

We start configuring `babel.config.js`. We type `module.exports` and assign it to an object with a key called `presets`, that is going to have a value that is an array of arrays, and the first element is going to be `'@babel/preset-env'`, and the second element will be an object with a `targets` key that has a value or property that is an object with a `node` property that has a value of `'current'`.
```
module.exports = {
   presets: [['@babel/preset-env', { targets: { node: 'current' } }]],
};
```

We then configure `jest.config.js` by writing `module.exports` and assigning it to an object with a key called `testEnvironment` with a value of `'node'`.
```
module.exports = {
   testEnvironment: 'node',
};
```

That's all of the configuration we need to do to use ES modules and to start testing our code.

### Writing our Scripts for Testing

Now we need to write our scripts in package.json. In package.json we currently have the following:
```
{
 "name": "sequelize-course",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
   "test": "echo \"Error: no test specified\" && exit 1"
```

```
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/cli": "^7.17.3",
    "@babel/core": "^7.17.5",
    "@babel/node": "^7.16.8",
    "@babel/preset-env": "^7.16.11",
    "@types/jest": "^27.4.0",
    "jest": "^27.5.1",
    "nodemon": "^2.0.15",
    "supertest": "^6.2.2"
  },
  "dependencies": {
    "bcrypt": "^5.0.1",
    "cls-hooked": "^4.2.2",
    "dotenv": "^16.0.0",
    "express": "^4.17.3",
    "jsonwebtoken": "^8.5.1",
    "morgan": "^1.10.0",
    "pg": "^8.7.3",
    "sequelize": "^6.16.2",
    "sequelize-cli": "^6.4.1"
  }
}
```

The `"scripts"` key has a value that is an object with a key `"test".` The key called `"test"` has a default value of the following: `"echo \"Error: no test specified\" && exit 1"`. We remove this and instead for the test we are going to execute `jest`, so we write `jest` and then we add `-runInBand` command:

```
  "scripts": {
    "test": "jest --runInBand"
  },
```

On the next line we add `"test:watch":"npm test -- --watch"`:

```
    "test": "jest --runInBand",
    "test:watch":"npm test -- --watch"
```

Everything that is to the right of the `--` is going to be appended to the original command. Essentially, this line `"npm test -- --watch"` is equivalent to `"jest --runInBand --watch"`. It is equivalent, but it would be a repetition of code to do:

```
"scripts": {
  "test": "jest --runInBand",
  "test:watch":"jest --runInBand --watch"
},
```

So that is why we do this instead:

```
"scripts": {
  "test": "jest --runInBand",
  "test:watch":"npm test -- --watch"
},
```

The other test script that we need to write is coverage to test column coverage, so we do `"test:coverage":"npm test -- coverage"`:

```
"scripts": {
  "test": "jest --runInBand",
  "test:watch":"npm test -- --watch",
  "test:coverage":"npm test -- --coverage"
},
```

## Writing our scripts related to starting, developing, debugging, and building the application

For build, we're going to say, "hey, take everything from the source folder (we need to create one) and compile it inside a folder called dist": `"build":"babel ./src --out-dir ./dist",`

```
"scripts": {
  "build":"babel ./src --out-dir ./dist",
  "test": "jest --runInBand",
  "test:watch":"npm test -- --watch",
  "test:coverage":"npm test -- --coverage"
},
```

When we want to start our app, we want to run `"start":"node dist/server.js",` The server does not exist yet (will be created in later videos).

```
"scripts": {
  "build":"babel ./src --out-dir ./dist",
  "start":"node dist/server.js",
  "test": "jest --runInBand",
  "test:watch":"npm test -- --watch",
  "test:coverage":"npm test -- --coverage"
},
```

For developing, we're going to say `NODE_ENV`, we're going to set this environment variable to `development` and we're going to execute `nodemon —-exec babel-node`, which is why we installed the dev dependency `@babel/node` and we're going to say source server.js (`src/server.js`). `nodemon` and `babel` are going to compile `src/server.js` when we are developing our application

```
"scripts": {
  "build":"babel ./src --out-dir ./dist",
  "start":"node dist/server.js",
  "dev":"NODE_ENV=development nodemon --exec babel-node src/server.js",
  "test": "jest --runInBand",
  "test:watch":"npm test -- --watch",
  "test:coverage":"npm test -- --coverage"
},
```

There's one additional command that we want to use, and that is the `debug` command and basically we're going here to run `"npm run dev —- —-inspect"`, which will allow us to use Chrome Developer tools in order to debug our code with all of the power of the Chrome Developer tools or any other Chromium based browser:

```
 "scripts": {
  "build":"babel ./src --out-dir ./dist",
  "start":"node dist/server.js",
  "debug":"npm run dev -- --inspect",
  "dev":"NODE_ENV=development nodemon --exec babel-node src/server.js",
  "test": "jest --runInBand",
  "test:watch":"npm test -- --watch",
  "test:coverage":"npm test -- --coverage"
},
```

## 13. Creating and connecting to the database

We will use Docker to create our main database and our test database.

### Testing strategies

- "Real database" set with Docker

    - Pros: we are testing against a "real" database, so our tests are more reliable
    - Cons: more difficult to set up both locally and in CI environments
- "In memory" databases
    - Pros: easier to set up, it can be an NPM package
    - Cons: they simulate a "real" database, but they aren't
- SQLite
    - Pros: lightweight database
    - Cons: it's not Postgres!

We will be using the first strategy in this course.

### Creating our docker-compose.yaml file

We will now create a docker-compose.yaml file. In docker-compose.yaml, we specify that we want to use `version:'3'`

```
version:'3'
```

We then add the `services:` where we define our two databases which can be named whatever we want.
For our first database, we will call it `postgres`:

```
version:'3'
services:
 postgres:
```

Inside the database called `postgres`, we will create an image and call it `postgres` as well as add `:13`. The `13` represents the name of the tag, and the tag in this context refers to the version of Postgres we're going to use.

```
version:'3'
services:
 postgres:
   image: postgres:13
```

You can also add container_name for the container that you want to use, but adding the `container_name` is optional.
With the container name:

```
version:'3'
services:
 postgres:
   image: postgres:13
   container_name:sequelize-course-db
```

Without the container name:
```
version:'3'
services:
 postgres:
   image: postgres:13
```

We then get all of the information from `env_file`, which we will go ahead and create with the command `touch .env`
```
version:'3'
services:
 postgres:
   image: postgres:13
   container_name:sequelize-course-db
   env_file:
     -.env
```

In the `environment` we add the POSTGRES_PASSWORD as a key and for value we say, "hey, take these from the DB_PASSWORD environment variable, and if this is not set then default to the password postgres". The `:-` means that everything to the right is going to be the default value:
```
version:'3'
services:
 postgres:
   image: postgres:13
   container_name:sequelize-course-db
   env_file:
     -.env
   environment:
     POSTGRES_PASSWORD:${DB_PASSWORD:-postgres}
```

For the ports, we will say, "hey, take the port specified in the DB_PORT, and if this is not specified in the .env file, then just use the default port 5432 and map it to port 5432."

**`${DB_PORT:-5432}`**`:5432` represents the host port, so basically the port of our machine. `${DB_PORT:-5432}`**`:5432`** represents the port of the container.
```
version:'3'
services:
```

```
postgres:
  image: postgres:13
  container_name:sequelize-course-db
  env_file:
    -.env
  environment:
    POSTGRES_PASSWORD:${DB_PASSWORD:-postgres}
  ports:
    - ${DB_PORT:-5432}:5432
```

For the container, the port will always be port 5432. For the host's default port, it can be any port, but don't use port 80 or 22 because they are very important ports reserved for other things, but you can use 5433, 5434, etc.

This represents our first service, so our first database:

```
postgres:
  image: postgres:13
  container_name:sequelize-course-db
  env_file:
    -.env
  environment:
    POSTGRES_PASSWORD:${DB_PASSWORD:-postgres}
  ports:
    - ${DB_PORT:-5432}:5432
```

For the second database, we will copy and paste from the first one:

```
postgres:
  image: postgres:13
  container_name:sequelize-course-db
  env_file:
    -.env
  environment:
    POSTGRES_PASSWORD:${DB_PASSWORD:-postgres}
  ports:
    - ${DB_PORT:-5432}:5432
```

This is going to be our test database, so we will name it `postgres-test`:

```
postgres-test:
  image: postgres:13
  container_name:sequelize-course-db
  env_file:
    -.env
  environment:
    POSTGRES_PASSWORD:${DB_PASSWORD:-postgres}
  ports:
    - ${DB_PORT:-5432}:5432
```

We will leave the tag as `13` because we still want to use version 13 of Postgres.

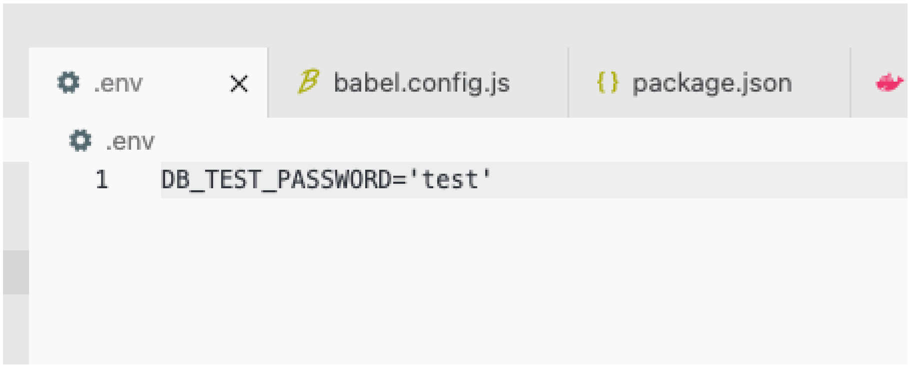
We will change the container name to `sequelize-course-test-db`:

```
postgres-test:
  image: postgres:13
  container_name:sequelize-course-test-db
  env_file:
    -.env
  environment:
    POSTGRES_PASSWORD:${DB_PASSWORD:-postgres}
  ports:
    - ${DB_PORT:-5432}:5432
```

`docker-compose.yaml` will get everything from the `.env` file again.
For the password, instead of putting `DB_PASSWORD`, we will put `DB_TEST_PASSWORD`, to make it flexible.

```
postgres-test:
  image: postgres:13
  container_name:sequelize-course-test-db
  env_file:
    -.env
  environment:
    POSTGRES_PASSWORD:${DB_TEST_PASSWORD:-postgres}
  ports:
    - ${DB_PORT:-5432}:5432
```

So if we put something like, `DB_TEST_PASSWORD='test'` in the file `.env`, like so:



```
DB_TEST_PASSWORD='test'
```

DB_TEST_PASSWORD in the docker-compose.yaml file will take the password from the .env file. If the environment variable for DB_TEST_PASSWORD was not set in the .env file, then the default password specified in the docker-compose.yaml file will be used, and we'll leave the default password set to postgres.

We will change DB_PORT to DB_TEST_PORT to make it different. If the port is not set in the .env file, we will make the default port be 5433.

```
 postgres-test:
   image: postgres:13
   container_name:sequelize-course-test-db
   env_file:
     -.env
   environment:
     POSTGRES_PASSWORD:${DB_TEST_PASSWORD:-postgres}
   ports:
     - ${DB_TEST_PORT:-5433}:5432
```

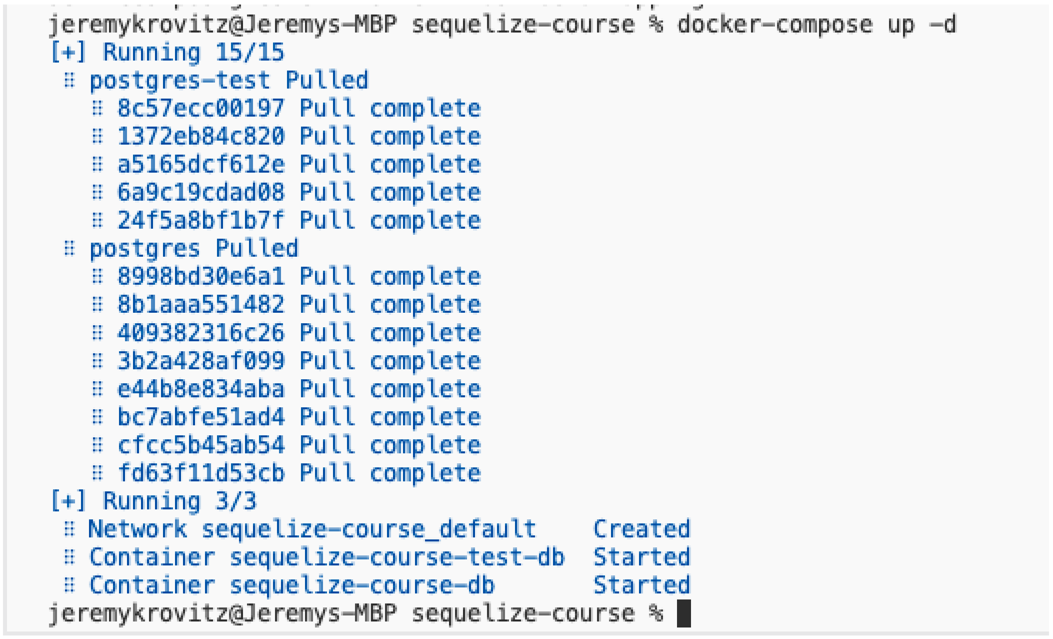
The host's default port of the testing database has to be different from the host's default port of the main database because we're going to have to use these two databases at the same time, so they can't have the same port.

This is the entire finished docker-compose.yaml file:

```
version: "3.9"
services:
 postgres:
   image: postgres:13
   container_name: sequelize-course-db
   env_file:
     - .env
   environment:
     POSTGRES_PASSWORD: ${DB_PASSWORD:-postgres}
   ports:
     - ${DB_PORT:-5432}:5432
 postgres-test:
   image: postgres:13
   container_name: sequelize-course-test-db
   env_file:
     - .env
   environment:
     POSTGRES_PASSWORD: ${DB_TEST_PASSWORD:-postgres}
   ports:
    - ${DB_TEST_PORT:-5433}:5432
```

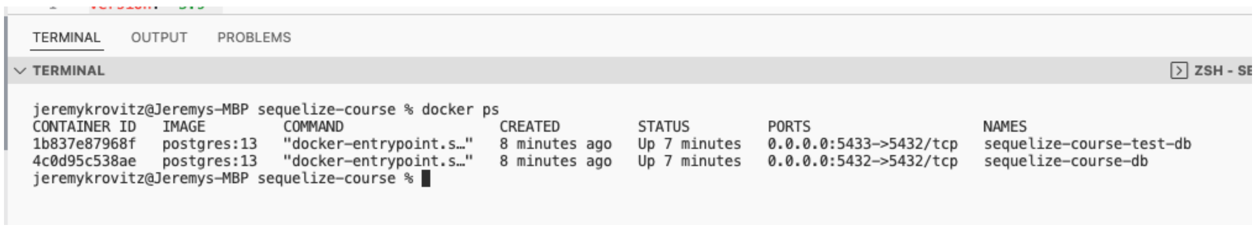
Be very careful about the formatting of the `docker-compose.yaml` file. Make sure to add a space after each of the colons if there is a value that comes after it on the same line; otherwise, there will be an error.

Now, to start docker, we can run the following command in the terminal (the first time you run it, the images will be downloaded):
`docker-compose up -d`



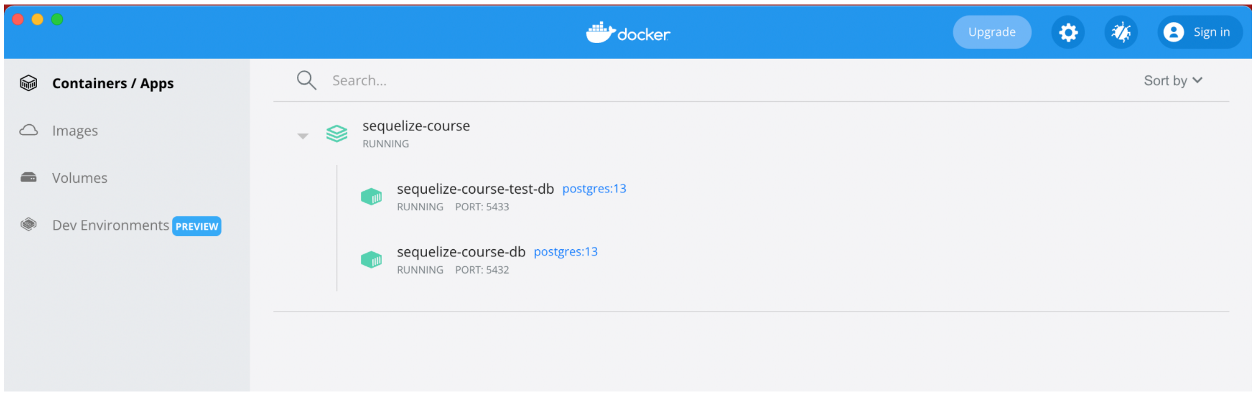
```
jeremykrovitz@Jeremys-MBP sequelize-course % docker-compose up -d
[+] Running 15/15
 ⠿ postgres-test Pulled
   ⠿ 8c57ecc00197 Pull complete
   ⠿ 1372eb84c820 Pull complete
   ⠿ a5165dcf612e Pull complete
   ⠿ 6a9c19cdad08 Pull complete
   ⠿ 24f5a8bf1b7f Pull complete
 ⠿ postgres Pulled
   ⠿ 8998bd30e6a1 Pull complete
   ⠿ 8b1aaa551482 Pull complete
   ⠿ 409382316c26 Pull complete
   ⠿ 3b2a428af099 Pull complete
   ⠿ e44b8e834aba Pull complete
   ⠿ bc7abfe51ad4 Pull complete
   ⠿ cfcc5b45ab54 Pull complete
   ⠿ fd63f11d53cb Pull complete
[+] Running 3/3
 ⠿ Network sequelize-course_default    Created
 ⠿ Container sequelize-course-test-db  Started
 ⠿ Container sequelize-course-db       Started
jeremykrovitz@Jeremys-MBP sequelize-course %
```

You can run `docker ps` to see the running containers:



```
jeremykrovitz@Jeremys-MBP sequelize-course % docker ps
CONTAINER ID   IMAGE         COMMAND                  CREATED         STATUS         PORTS                    NAMES
1b837e87968f   postgres:13   "docker-entrypoint.s…"   8 minutes ago   Up 7 minutes   0.0.0.0:5433->5432/tcp   sequelize-course-test-db
4c0d95c538ae   postgres:13   "docker-entrypoint.s…"   8 minutes ago   Up 7 minutes   0.0.0.0:5432->5432/tcp   sequelize-course-db
jeremykrovitz@Jeremys-MBP sequelize-course %
```

If you look at Docker Desktop, you will see the two containers running



sequelize-course
RUNNING

sequelize-course-test-db postgres:13
RUNNING PORT: 5433

sequelize-course-db postgres:13
RUNNING PORT: 5432

Now that we have 2 databases, we can connect to these two databases.

## Section 3: JWT and Bcrypt

### 14. Understanding JWT

#### What is a JSON Web Token?

A "JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA (jwt.io/introduction, 2022)".

We are going to have access tokens and refresh tokens, which are basically, JSON web tokens. We have a JSON web token package that will hide all of the complexity for us. We're just going to generate images of tokens and verify them. In this course, we're going to build middleware that is going to make the verification of these JSON tokens just as described here. JSON web tokens are very easy to use. They are stateless, not like cookies. So that is why JSON web tokens are widely used now days. JSON web tokens are good for authorization and for secure information exchange.

## Parts of a JSON Web Token

JSON web tokens consist of three parts separated by dots(.):

- Header
- Payload
- Signature

Therefore, a JWT typically looks like the following:
`xxxxx.yyyyy.zzzzz` (header.payload.signature)

## Information about Header, Payload, and Signature from https://jwt.io/introduction

**Header**
The header *typically* consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

For example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Then, this JSON is **Base64Url** encoded to form the first part of the JWT.

**Payload**

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: *registered*, *public*, and *private* claims.

- **Registered claims**: These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub** (subject), **aud** (audience), and others.
  Notice that the claim names are only three characters long as JWT is meant to be compact.
- **Public claims**: These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token

  Registry or be defined as a URI that contains a collision resistant namespace.
- **Private claims**: These are the custom claims created to share information between parties that agree on using them and are neither *registered* or *public* claims.

An example payload could be:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

The payload is then **Base64Url** encoded to form the second part of the JSON Web Token.

Do note that for signed tokens this information, though protected against tampering, is readable by anyone. Do not put secret information in the payload or header elements of a JWT unless it is encrypted.

### Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

### Putting all together

The output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.

The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

### How do JSON Web Tokens work?

When a user successfully logins in using their credentials, a JSON Web Token will be returned. This is very important because this is how we're going to design our controllers. Since tokens are credentials, great care must be taken to prevent security issues, which is why we'll expire our JSON Web Tokens.
In our case we are going to make a query to the database using the email field.

## 15. Understanding Bcrypt

Passwords should NEVER be stored as plain text. We want to provide a one-way-road to security by hashing passwords. Hashing means to encrypt a password somehow. However, we also explained that hashing alone is not sufficient to mitigate more involved attacks such as rainbow tables.

### Rainbow Table Example

If user1 uses the password "Test123!", for example, and user2 uses the same password "Test123!", they will have the same hash. So a hacker is going to have all of these hashes stored and mapped to this password. So basically, they would have an object like this:

```
{
  "Lskfjdfklsdjklfjklsfsljkdf": "Test123!"
}
```

So if a hacker goes and reaches your database and sees that the password is this, then they're going to say, "hey, this user has this password.", which would allow the hacker to login to your account and do bad things.

### Better Way to Store Passwords

A better way to store passwords is to add a salt to the hashing process. If a salt is added, you would have something like:

user1, Test123! = Lskfjdfklsdjklfjklsfsljkdf.234233543
user2, Test123! = Lskfjdfklsdjklfjklsfsljkdf.212312034

A salt makes it more complicated for the hacker to know your password; however, it's still not sufficient to have a salt. The ideal authentication platform would integrate these two processes, hashing and salting seamlessly. There are plenty of cryptographic families to choose from such as the SHA2 family and the SHA-3 family. However, one design problem with the SHA families is that they were designed to be computationally fast. How fast a cryptographic function can calculate a has an immediate and significant bearing on how safe the password is: Faster calculations mean faster brute-force attacks.

Bcrypt was designed based on the Blowfish cipher>; b for Blowfish and crypt for the name of the hashing function used by the UNIX password system. The Bcrypt algorithm is intended to be slow and it can be slower if we say so, and it will be beneficial because the years are going to pass and hardware is going to be more and more fast, so we need a way to avoid this fast cryptographic function.

## 16. Adding environment variables

We need to create some configurations, so we are going to create a folder called config inside of the src folder. Inside the config folder, we are going to have three files — index.js, database.js, environment.js.

In index.js (the entry folder for the config directory), we're going to write:

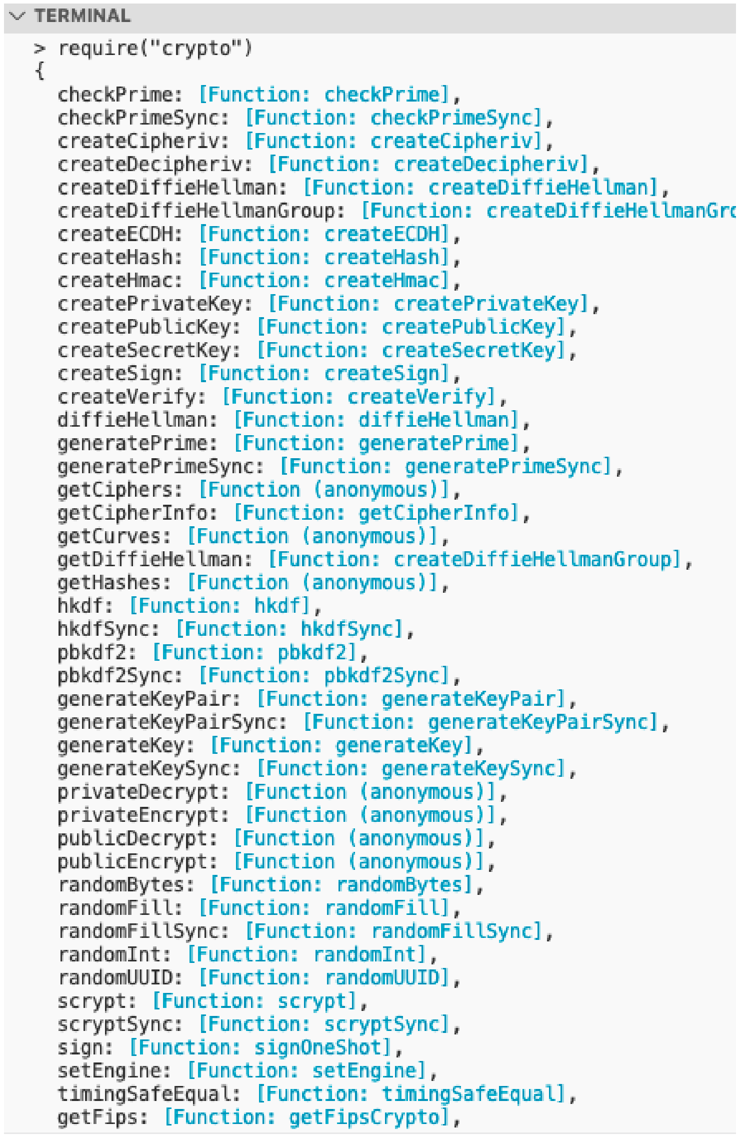
```
import dotenv from 'dotenv';

dotenv.config();
```

In database.js, we're going to write the following (we're not using the ES6 way of export because this file is going to be used by a .sequelizerc file, and that .sequelizerc file also uses the old JavaScript syntax)
. For each of the database key value pairs, whatever comes after the `||` represents the default if a value isn't specified for the environment variable.

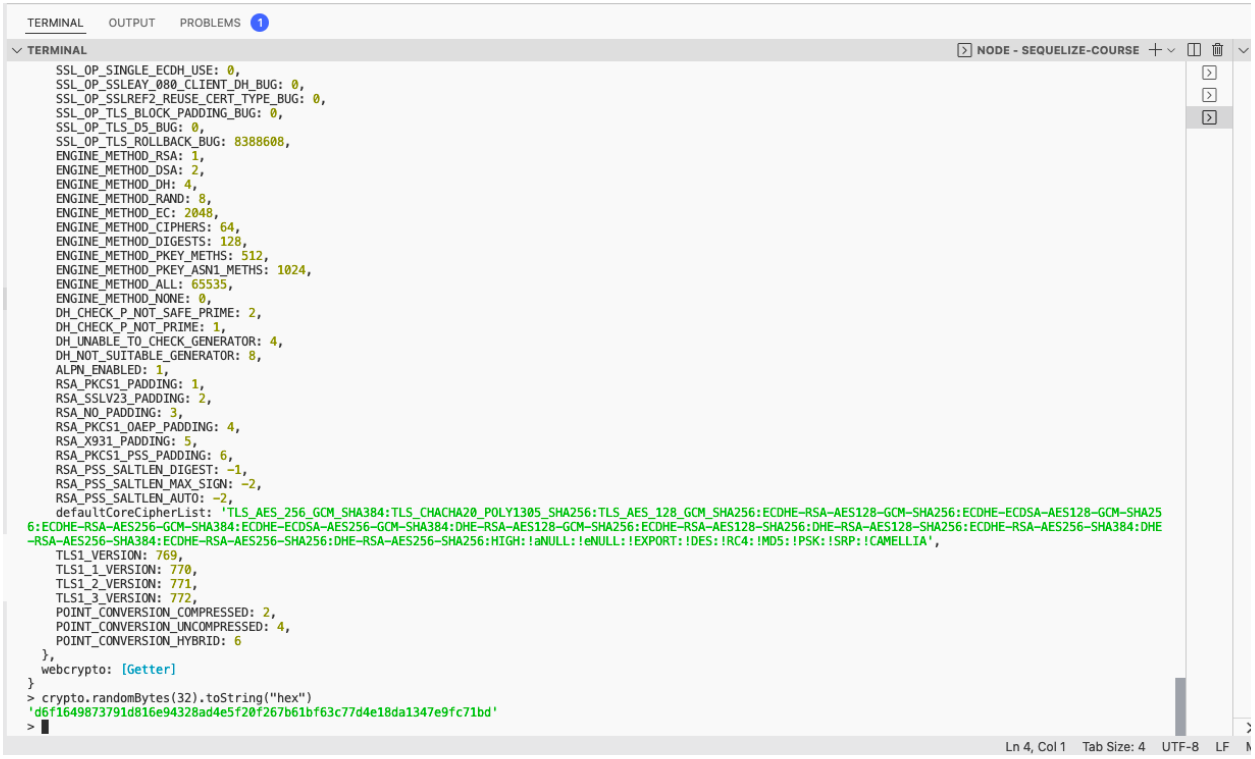
```
module.exports = {
  development: {
      username: process.env.DB_USERNAME || 'postgres',
      password: process.env.DB_PASSWORD || 'postgres',
      host: process.env.DB_HOST || 'localhost',
      port: parseInt(process.env.DB_PORT) || 5432,
        database: process.env.DB_DATABASE || 'postgres',
        dialect: 'postgres',
    },
    test: {
        username: process.env.DB_TEST_USERNAME || 'postgres',
        password: process.env.DB_TEST_PASSWORD || 'postgres',
        host: process.env.DB_TEST_HOST || 'localhost',
        port: parseInt(process.env.DB_TEST_PORT) || 5433,
        database: process.env.DB_TEST_DATABASE || 'postgres',
        dialect: 'postgres',
    },
}
```

The third file created is called environment.js, which will contain all of our environment variables.
For the jwtAccessTokenSecret, we can generate a default access token by generating a random string through the following process:

1. Type `node` in the terminal and press Enter.
2. Type `require("crypto")` and press Enter // crypto is a library that comes with Node
3. Then type `crypto.randomBytes(32).toString("hex")` // generates 32 random bytes
4. Press Enter.



```
> require("crypto")
{
  checkPrime: [Function: checkPrime],
  checkPrimeSync: [Function: checkPrimeSync],
  createCipheriv: [Function: createCipheriv],
  createDecipheriv: [Function: createDecipheriv],
  createDiffieHellman: [Function: createDiffieHellman],
  createDiffieHellmanGroup: [Function: createDiffieHellmanGro
  createECDH: [Function: createECDH],
  createHash: [Function: createHash],
  createHmac: [Function: createHmac],
  createPrivateKey: [Function: createPrivateKey],
  createPublicKey: [Function: createPublicKey],
  createSecretKey: [Function: createSecretKey],
  createSign: [Function: createSign],
  createVerify: [Function: createVerify],
  diffieHellman: [Function: diffieHellman],
  generatePrime: [Function: generatePrime],
  generatePrimeSync: [Function: generatePrimeSync],
  getCiphers: [Function (anonymous)],
  getCipherInfo: [Function: getCipherInfo],
  getCurves: [Function (anonymous)],
  getDiffieHellman: [Function: createDiffieHellmanGroup],
  getHashes: [Function (anonymous)],
  hkdf: [Function: hkdf],
  hkdfSync: [Function: hkdfSync],
  pbkdf2: [Function: pbkdf2],
  pbkdf2Sync: [Function: pbkdf2Sync],
  generateKeyPair: [Function: generateKeyPair],
  generateKeyPairSync: [Function: generateKeyPairSync],
  generateKey: [Function: generateKey],
  generateKeySync: [Function: generateKeySync],
  privateDecrypt: [Function (anonymous)],
  privateEncrypt: [Function (anonymous)],
  publicDecrypt: [Function (anonymous)],
  publicEncrypt: [Function (anonymous)],
  randomBytes: [Function: randomBytes],
  randomFill: [Function: randomFill],
  randomFillSync: [Function: randomFillSync],
  randomInt: [Function: randomInt],
  randomUUID: [Function: randomUUID],
  scrypt: [Function: scrypt],
  scryptSync: [Function: scryptSync],
  sign: [Function: signOneShot],
  setEngine: [Function: setEngine],
  timingSafeEqual: [Function: timingSafeEqual],
  getFips: [Function: getFipsCrypto],
```



```
    SSL_OP_SINGLE_ECDH_USE: 0,
    SSL_OP_SSLEAY_080_CLIENT_DH_BUG: 0,
    SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG: 0,
    SSL_OP_TLS_BLOCK_PADDING_BUG: 0,
    SSL_OP_TLS_D5_BUG: 0,
    SSL_OP_TLS_ROLLBACK_BUG: 8388608,
    ENGINE_METHOD_RSA: 1,
    ENGINE_METHOD_DSA: 2,
    ENGINE_METHOD_DH: 4,
    ENGINE_METHOD_RAND: 8,
    ENGINE_METHOD_EC: 2048,
    ENGINE_METHOD_CIPHERS: 64,
    ENGINE_METHOD_DIGESTS: 128,
    ENGINE_METHOD_PKEY_METHS: 512,
    ENGINE_METHOD_PKEY_ASN1_METHS: 1024,
    ENGINE_METHOD_ALL: 65535,
    ENGINE_METHOD_NONE: 0,
    DH_CHECK_P_NOT_SAFE_PRIME: 2,
    DH_CHECK_P_NOT_PRIME: 1,
    DH_UNABLE_TO_CHECK_GENERATOR: 4,
    DH_NOT_SUITABLE_GENERATOR: 8,
    ALPN_ENABLED: 1,
    RSA_PKCS1_PADDING: 1,
    RSA_SSLV23_PADDING: 2,
    RSA_NO_PADDING: 3,
    RSA_PKCS1_OAEP_PADDING: 4,
    RSA_X931_PADDING: 5,
    RSA_PKCS1_PSS_PADDING: 6,
    RSA_PSS_SALTLEN_DIGEST: -1,
    RSA_PSS_SALTLEN_MAX_SIGN: -2,
    RSA_PSS_SALTLEN_AUTO: -2,
    defaultCoreCipherList: 'TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_SHA256:TLS_AES_128_GCM_SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA25
6:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-SHA256:DHE-RSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384:DHE
-RSA-AES256-SHA384:ECDHE-RSA-AES256-SHA256:DHE-RSA-AES256-SHA256:HIGH:!aNULL:!eNULL:!EXPORT:!DES:!RC4:!MD5:!PSK:!SRP:!CAMELLIA',
    TLS1_VERSION: 769,
    TLS1_1_VERSION: 770,
    TLS1_2_VERSION: 771,
    TLS1_3_VERSION: 772,
    POINT_CONVERSION_COMPRESSED: 2,
    POINT_CONVERSION_UNCOMPRESSED: 4,
    POINT_CONVERSION_HYBRID: 6
  },
  webcrypto: [Getter]
}
> crypto.randomBytes(32).toString("hex")
'd6f1649873791d816e94328ad4e5f20f267b61bf63c77d4e18da1347e9fc71bd'
> 
```

Copy and paste it to be the default JWT_ACCESS_TOKEN_SECRET.

For the jwtRefreshTokenSecret, we can generate a default refresh token by generating another random string using the same process we did for jwtAccessTokenSecret.

This is the environment.js file:

```
export default {
   port: parseInt(process.env.PORT) || 8080,
   nodeEnv: process.env.NODE_ENV || 'production',
   saltRounds: parseInt(process.env.SALT_ROUNDS) || 10, // This line is for the bcrypt algorithm; tells the difficulty of hashing function, if number is higher, the opperation will be slower to hash a password
   jwtAccessTokenSecret: process.env.JWT_ACCESS_TOKEN_SECRET || 'd6f1649873791d816e94328ad4e5f20f267b61bf63c77d4e18da1347e9fc71bd',
   jwtRefreshTokenSecret: process.env.JWT_REFRESH_TOKEN_SECRET || '24ffd98a0c09bd007cd8f5ac0e980bd610c463ee19582c0b1336df01283fb051'
};
```

It is a best practice because we are defaulting things inside the environment.js file.

## 17. Creating utils for JWT

### Create src/utils/jwt-utils.js

We will add some JSON Web Token Utils. We will generate four methods—generate access token, generate refresh token, verify access token, and verify refresh token.

Inside src we create a new folder called utils. Inside of utils, we will create a file called jwt-utils.js.

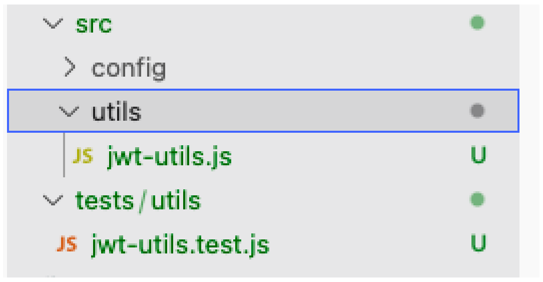
We need the refresh token to create another access token, so we will not make the refresh token expire. You can make the refresh token expire, though, we just are not going to avoid making our application too complex.

This is the code for our src/utils/jwt-utils.js file:

```
import jwt from 'jsonwebtoken';
import environment from '../config/environment';

export default class JWTUtils{
   static generateAccessToken(payload, options = {}) {
       const { expiresIn = '1d' } = options; // make json access token expire in one
day
       return jwt.sign(payload, environment.jwtAccessTokenSecret, {expiresIn})
   }

   static generateRefreshToken(payload) {
       return jwt.sign(payload, environment.jwtRefreshTokenSecret);
   }

   static verifyAccessToken(accessToken) {
       return jwt.verify(accessToken, environment.jwtAccessTokenSecret);
   }

   static verifyRefreshToken(refreshToken) {
       return jwt.verify(refreshToken, environment.jwtRefreshTokenSecret);
   }
}
```

## Testing src/utils/jwt-utils.js

We will create a top-level folder called tests, a folder within test called utils, and a file within utils called jwt-utils.test.js. We structure our tests folder in the same way we structure our src folder:



```
src
  config
  utils
    jwt-utils.js        U
tests / utils
  jwt-utils.test.js     U
```

In tests/utils/jwt-utils.test.js, we add the following imports:

```
import jwt from 'jsonwebtoken' // we are still going to use this library for handling some errors
import JWTUtils from '../../src/utils/jwt-utils'
```

We then write the test for the generateAccessToken function:

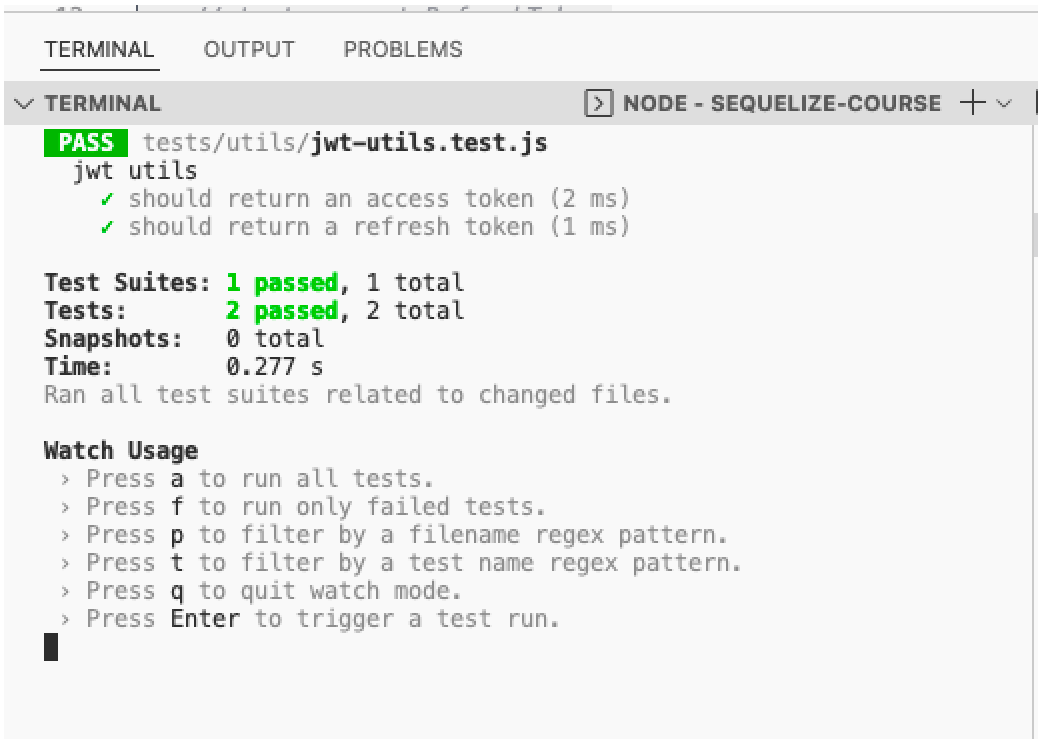
```
describe('jwt utils', () => {
   it('should return an access token', () => {
       const payload = { email: 'test@example.com' }
       expect(JWTUtils.generateAccessToken(payload)).toEqual(expect.any(String));
   });
});
```

After that we write the test for the generateRefreshToken function:

```
describe('jwt utils', () => {

   // test generateAccessToken
   it('should return an access token', () => {
       const payload = { email: 'test@example.com' }
       expect(JWTUtils.generateAccessToken(payload)).toEqual(expect.any(String));
   });

   // test generateRefreshToken
   it('should return a refresh token', () => {
       const payload = { email: 'test@example.com' }
       expect(JWTUtils.generateRefreshToken(payload)).toEqual(expect.any(String));
   });
});
```

We can then check to see if the test passes by typing `npm run test:watch` in the terminal.

We get the following output:



```
PASS  tests/utils/jwt-utils.test.js
  jwt utils
    ✓ should return an access token (2 ms)
    ✓ should return a refresh token (1 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.277 s
Ran all test suites related to changed files.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press q to quit watch mode.
 › Press Enter to trigger a test run.
```

We add two more tests for the other two functions—verifyAccessToken and verifyRefreshToken. This is the entire tests/utils/jwt-utils.test.js file (the two added tests are bolded):

```
import jwt from 'jsonwebtoken' // we are still going to use this library for handling some errors
import JWTUtils from '../../src/utils/jwt-utils'

describe('jwt utils', () => {

   // test generateAccessToken
   it('should return an access token', () => {
       const payload = { email: 'test@example.com' }
       expect(JWTUtils.generateAccessToken(payload)).toEqual(expect.any(String));
   });

   // test generateRefreshToken
   it('should return a refresh token', () => {
       const payload = { email: 'test@example.com' }
```

```
        expect(JWTUtils.generateRefreshToken(payload)).toEqual(expect.any(String));
    });

    it('should verify that the access token is valid', () => {
        const payload = { email: 'test@example.com' }
        const jwt = JWTUtils.generateAccessToken(payload)

expect(JWTUtils.verifyAccessToken(jwt)).toEqual(expect.objectContaining(payload)
        )
    })

    it('should verify that the refresh token is valid', () => {
        const payload = { email: 'test@example.com' }
        const jwt = JWTUtils.generateRefreshToken(payload)

expect(JWTUtils.verifyRefreshToken(jwt)).toEqual(expect.objectContaining(payload)
        )
    })
});
```

Add two more tests to check for errors. This is the entire tests/util/jwt-utils.test.js file (tests added are bolded):

```
import jwt from 'jsonwebtoken' // we are still going to use this library for handling
some errors
import JWTUtils from '../../src/utils/jwt-utils'

describe('jwt utils', () => {

    // test generateAccessToken
    it('should return an access token', () => {
        const payload = { email: 'test@example.com' }
        expect(JWTUtils.generateAccessToken(payload)).toEqual(expect.any(String));
    });

    // test generateRefreshToken
    it('should return a refresh token', () => {
        const payload = { email: 'test@example.com' }
        expect(JWTUtils.generateRefreshToken(payload)).toEqual(expect.any(String));
    });

    it('should verify that the access token is valid', () => {
        const payload = { email: 'test@example.com' }
```

```
       const jwt = JWTUtils.generateAccessToken(payload)

expect(JWTUtils.verifyAccessToken(jwt)).toEqual(expect.objectContaining(payload)
       )
   })

   it('should verify that the refresh token is valid', () => {
       const payload = { email: 'test@example.com' }
       const jwt = JWTUtils.generateRefreshToken(payload)

expect(JWTUtils.verifyRefreshToken(jwt)).toEqual(expect.objectContaining(payload)
       )
   })

   it('should error if the access token is invalid', () => {
       expect(() =>
JWTUtils.verifyAccessToken('invalid.token')).toThrow(jwt.JsonWebTokenError)
   })

   it('should error if the refresh token is invalid', () => {
       expect(() =>
JWTUtils.verifyRefreshToken('invalid.token')).toThrow(jwt.JsonWebTokenError)
   })
});
```

## Section 4: Databases and models

### 18. Creating the database class

It is time to set-up our database connection. For that we're going to use a class in order to create the database class that handles that connection. Inside the src folder, we create a folder called database. Inside the database folder, we create an index.js file.

#### Sequelize

Sequelize supports two ways of using transactions:

1. **Unmanaged transactions:** basically committing and rolling back manually
2. **Managed transactions:** Sequelize will automatically rollback the transaction if an error is thrown, or commit the transaction otherwise. Also, if CLS (Continuation Local Storage) is enabled, all queries within the transaction callback will automatically receive the transaction object

When you define a model, you're telling Sequelize a few things about its table in the database. However, what if the table actually doesn't even exist in the database? What if it exists, but it has different columns, less columns, or any other difference?

This is where model synchronization comes in. A model can be synchronized with the database by calling model. sync (options), an asynchronous function (that returns a Promise). With this call, Sequelize will automatically perform a SQL query to the database. Note that this changes only the table in the database, not the model in the JavaScript side.

`User. sync()` - This creates the table if it doesn't exist (and does nothing if it already exists)
`User.sync({ force: true })` - This creates the table, dropping it first if it already existed
`User.sync({ alter: true })` - This checks what is the current state of the table in the database (which columns it has, what are their data types, etc), and then performs the necessary changes in the table to make it match the model.

Example:

```
await User.sync({ force: true }):
console. log ("The table for the User model was just
(re)created!");
```

For production:

In production, we don't want the ORM to determine how the tables should be created and things like that. That is why we are going to create migrations for our tables.

For tests:

We still want the `force: true` for tests because for each test, we want a clean database.

### src/database/index.js file

```
import { hashSync } from "bcrypt"
import cls from 'cls-hooked'
import {Sequelize} from 'sequelize'

export default class Database {
  constructor(environment, dbConfig) {
       this.environment = environment
       this.dbConfig = dbConfig
       this.isTestEnvironment = this.environment === 'test' // creating this variable
because it will be useful for logging

   }
   async connect() {
       // Set up the namespace for transactions, as Sequlize does not use transactions
by default
       const namespace = cls.createNamespace('transactions-namespace');
       Sequelize.useCLS(namespace);

       // Create the conenction
       const { username, password, host, port, database, dialect } =
this.dbConfig[this.environment]; // this.dbConfig is going to be the object
database.js, and this.environment can be development or test (one of the two keys in
the outer-most object in the database.js file)
       this.connection = new Sequelize({ username, password, host, port, database,
dialect, logging: this.isTestEnvironment ? false : console.log }) // If we're in a
test environment, logging is false, otherwise use console.log in order to log the
queries which are very useful

       // Check if we connected successfully
       await this.connection.authenticate({ logging: false }) // we don't want to see
the query, so we set logging to false

       if (!this.isTestEnvironment) { // checking that we are not in a test
environment
           console.log('Connection to the database has been established
successfully.')
       }

       // Register the models - we're not going to do in this video 18
       // call function registerModels(this.connection)

       // Sync the models
       await this.sync();
   }

   async disconnect() {
       await this.connection.close();
```

```
    }

    async sync(){
        await this.connection.sync({
            logging: false,
            force: this.isTestEnvironment, // if this.isTestEnvironment equals true
        })

        if (!this.isTestEnvironment) {
            console.log('Connection synced successfully')
        }
    }
}
```

## 19. Register the models

The Database class in src/database/index.js handles our connection to the database. We are missing the registerModels function that we need to create.

The registerModels function is going to do two things:

1. It's going to put all of the models inside an object that we're going to call the models object. The keys of these objects are going to be the names of the models, and the values are going to be the class representing the models.
2. It's going to call the associate static method from each of these classes representing the models, and this associate method is going to say, "hey, for example, this model has a relationship, a one-to-one relationship with this other model, or this model belongs to this model or this model has a one-to-many relationship with these other models." Those are the associations that we are going to register in these static methods called associate.

We create a folder called models. In this folder we are going to have a file called index.js. The other files in the models folder are going to represent each of our tables in the database.

The src/models/index.js file:

```
import fs from 'fs'; // file system
import path from 'path';

let models = {}; // models object, the object we are going to populate with the
classes that represent our models

export function registerModels(sequelize) {
    const thisFile = path.basename(__filename); // __filename means the absolute path
to the index.js file
    const modelFiles = fs.readdirSync(__dirname); // __dirname is the path to the
models folder; readdirSync is a function to read all of the files inside that folder
    const filteredModelFiles = modelFiles.filter(file => {
        return file !== thisFile && file.slice(-3) === '.js' // read all of the files
that are not this file, so read all of the files that are not index.js and all of the
files that end with a .js extension.
    }) // we now have an array of files and need to iterate through the array

    // populate all of the object for all of these models
    for (const file of filteredModelFiles) {
        const model = require(path.join(__dirname, file)).default(sequelize) // import
or require the file and use path.join dirname with the name of the file and we're
going to import the default export. To the default export we're going to pass a
sequelize instance.
        models[model.name] = model; // we have the model, the class representing the
model; this is going to be an object representing the keys as the name of the model
and the value is going to be the class representing that model
    }

    // Call associate method on each of the models
    Object.keys(models).forEach(modelName => {
        if (models[modelName].associate) {
            models[modelName].associate(models); // To this associate method we are
going to pass all of the models
        }
    })

    // set additional key apart from the models and that is going to be the sequelize
instance; so apart from the models, the models object is going to have the sequelize
instance as well
    models.sequelize = sequelize;
}

export default models
```

In the file src/database/index.js, uncomment `registerModels(this.connection)` and add the import `import { registerModels } from "../models"`

## 20. Adding the server

We are now going to be adding the server. The server is going to be responsible for connecting to the database, creating the Express application, and listening for the port that we specify.

In the src folder, we create a file called `server.js`.

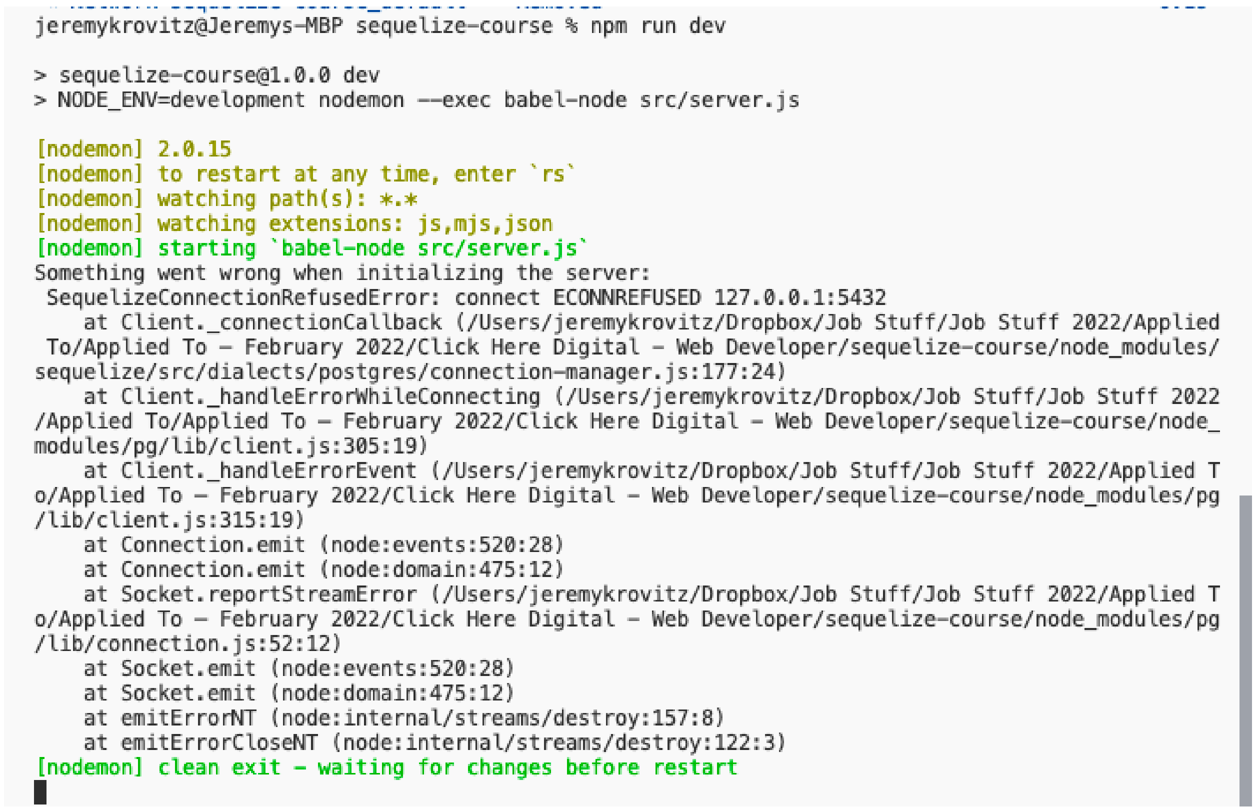
In `src/server.js` we create an immediately invoked function expression (iife), which means a function that executes after it has been defined.

The file `src/server.js` looks like this:

```
import './config' // we put import './config' at the top of the file because this is going to execute first
import Database from './database'
import environment from './config/environment' // and then we need to execute or import the environment variables after we have loaded all of our environment variables
import dbConfig from './config/database'

// immediately invoked function expression
(async () => {
   try {
       const db = new Database(environment.nodeEnv, dbConfig) // create new database and pass in environment and dbConfig
       await db.connect()

   } catch (err) {
       console.error('Something went wrong when initializing the server:\n', err.stack) // Logging message and printing the error stack
   }
})()
```

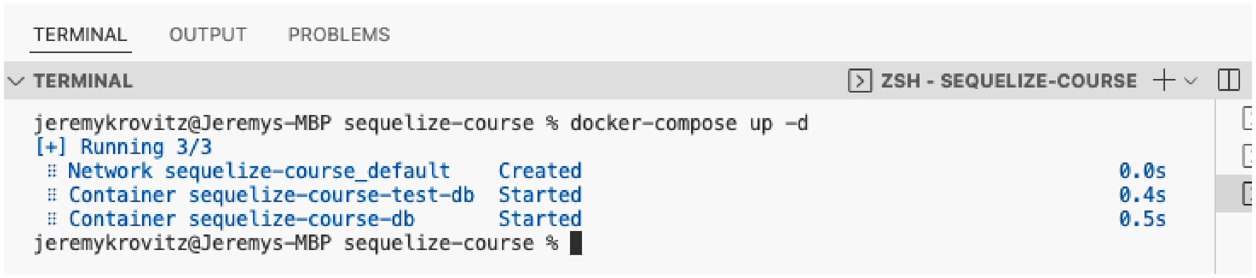
We can now run the server. We first try running the server without the database by typing the following command in the terminal and get an error because we are not connected to the database:

```
npm run dev
```



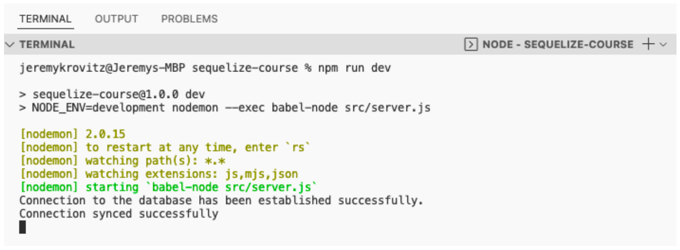
```
jeremykrovitz@Jeremys-MBP sequelize-course % npm run dev

> sequelize-course@1.0.0 dev
> NODE_ENV=development nodemon --exec babel-node src/server.js

[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `babel-node src/server.js`
Something went wrong when initializing the server:
 SequelizeConnectionRefusedError: connect ECONNREFUSED 127.0.0.1:5432
    at Client._connectionCallback (/Users/jeremykrovitz/Dropbox/Job Stuff/Job Stuff 2022/Applied
 To/Applied To - February 2022/Click Here Digital - Web Developer/sequelize-course/node_modules/
sequelize/src/dialects/postgres/connection-manager.js:177:24)
    at Client._handleErrorWhileConnecting (/Users/jeremykrovitz/Dropbox/Job Stuff/Job Stuff 2022
/Applied To/Applied To - February 2022/Click Here Digital - Web Developer/sequelize-course/node_
modules/pg/lib/client.js:305:19)
    at Client._handleErrorEvent (/Users/jeremykrovitz/Dropbox/Job Stuff/Job Stuff 2022/Applied T
o/Applied To - February 2022/Click Here Digital - Web Developer/sequelize-course/node_modules/pg
/lib/client.js:315:19)
    at Connection.emit (node:events:520:28)
    at Connection.emit (node:domain:475:12)
    at Socket.reportStreamError (/Users/jeremykrovitz/Dropbox/Job Stuff/Job Stuff 2022/Applied T
o/Applied To - February 2022/Click Here Digital - Web Developer/sequelize-course/node_modules/pg
/lib/connection.js:52:12)
    at Socket.emit (node:events:520:28)
    at Socket.emit (node:domain:475:12)
    at emitErrorNT (node:internal/streams/destroy:157:8)
    at emitErrorCloseNT (node:internal/streams/destroy:122:3)
[nodemon] clean exit - waiting for changes before restart
```

To start up docker-compose, we type the following in the terminal:
`docker-compose up -d`



```
jeremykrovitz@Jeremys-MBP sequelize-course % docker-compose up -d
[+] Running 3/3
 ⠿ Network sequelize-course_default    Created                                       0.0s
 ⠿ Container sequelize-course-test-db  Started                                       0.4s
 ⠿ Container sequelize-course-db       Started                                       0.5s
jeremykrovitz@Jeremys-MBP sequelize-course %
```

Now if we type the following command in the terminal, we no longer have an error and get a message that says "Connection to the database has been established successfully" and a message that says, "Connection synced successfully":
`npm run dev`



```
jeremykrovitz@Jeremys-MBP sequelize-course % npm run dev

> sequelize-course@1.0.0 dev
> NODE_ENV=development nodemon --exec babel-node src/server.js

[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `babel-node src/server.js`
Connection to the database has been established successfully.
Connection synced successfully
```

## 21. Adding test helpers

We need to create our models, but before doing that, we need some test helpers that are going to allow us to make our testing easier. For that, we will create a new file inside of the tests directory and name it tests-helpers.js.

tests/test-helpers.js:

```
import '../src/config' // import necessary environment variables
import Database from '../src/database'
import dbConfig from '../src/config/database'

let db

export default class TestHelpers {
   static async startDb() {
       db = new Database('test', dbConfig); // dbConfig contains the test object which
contains information about the test database
       await db.connect()
       return db // in case some of our tests need access to this instance of the
database
   }

   static async stopDb() {
       await db.disconnect();
   }

   // Method to reset database; each time we run a test, we are going to drop all of
the tables and have a test with a clean database, so we run the syncDb command before
each test
   static async syncDb() {
       await db.sync();
   }

}
```

## 22. Models overview

We're going to start writing our models.

## Models

We are going to have three models. I will be adding a Task model.

- User
- Refresh Token
- Roles
- Task - I will add this

### User model

- User will have many roles (I might just say tasks instead of roles) - one-to-many relationship
- User will have one refresh token - one-to-one relationship
- Properties
    - email (string)
    - password (string)
    - username (string)
    - firstName (string)
    - lastName (string)

### Roles model (may call it tasks or create a separate model for tasks)

- Roles will belong to a User
- Properties
    - role (string)

### Refresh Token model

- Refresh Token will belong to a User
- Properties
    - token (string)

The has-many relationship, the has-one relationship, the belongs to relationship are abstracted with sequelize.

If we go to src/models/index.js, we have the following line of code, which registers all of the associations such as the has-many relationship, the has-one relationship, and the belongs to relationship:

```
models[modelName].associate(models);
```

## 23. Creating the User model (Part 1)

We will start by defining the User model by creating a file inside of src/models called user.js.

We start by writing the following: `export default(sequelize)`

We do this because in src/models/index.js we are requiring this file

```
const model = require(path.join(__dirname, file)).default(sequelize) // import or
require the file and use path.join dirname with the name
```

Additionally, in src/models/index.js we are calling the default export with this piece of code: **`default(sequelize)`**:

```
const model = require(path.join(__dirname, file)).default(sequelize) // import or
require the file and use path.join dirname with the name
```

The default export that we're calling is: `export default(sequelize)` in src/models/user.js, and that returns a function, so we are going to pass this sequelize function here.

Inside the function, we need to actually return a class, so we're going to create the User class:

```
class User extends Model{}
```

We also need to import the Model class:

```
import {Model} from 'sequelize'
```

Inside the class we are going to have a static associate method, which is going to receive the models object, which remember is passed in the models/index.js file in the following line:

```
const model = require(path.join(__dirname, file)).default(sequelize)
```

So, we're saying if that model has this associate method, then call it with the models object, which is empty because we need to create all of our models.

In src/models/user.js, `static associate(`**`models`**`)` will have all of the models once we create them. We will pretend that we already have the role model, the token model, and the user model.

The following line of code is going to set up a one-to-one relationship between the User model and the RefreshToken model

```
User.RefreshToken = User.hasOne(models.RefreshToken)
```

The following line of code is going to set up a one-to-many relationship between the User model and the Roles model

```
User.Roles = User.hasMany(models.Roles)
```

When we create properties for a model in Node.js, and we have a type, we need to import DataTypes from sequelize.

The second argument to User.init is an object containing the sequelize instance, and we need to pass the modelName as a key and the name of the model as a value which in this case is 'User'.

```
// Defining the properties of the user model
   User.init({
       email: {
           type: DataTypes.STRING(100),
           allowNull: false,
           unique: true,
           validate: {
               isEmail: {
                   msg: 'Not a valid email address',
               },
           },
       },
       password: {
           type: DataTypes.STRING,
           allowNull: false,
       },
       username: {
           type: DataTypes.STRING(50),
           unique: true,
           validate: {
               len: {
                   args: [2, 50], // min and max number of characters
                   msg: 'Username must contain between 2 and 50 characters',
               },
           },
       },
       firstName: {
           type: DataTypes.STRING(50),
           validate: {
               len: {
                   args: [3, 50], // min and max number of characters
                   msg: 'First name must contain between 3 and 50 characters',
               },
           },
       },
       lastName: {
           type: DataTypes.STRING(50),
           validate: {
               len: {
                   args: [3, 50], // min and max number of characters
                   msg: 'Last name must contain between 3 and 50 characters',
               },
           },
       },
   }, {sequelize, modelName: 'User'})
```

Remember, that the model name "User" is going to be used in src/models/index.js in the below line of code to populate our models object (in src/models/user.js).

```
models[model.name] = model;
```

The init method in src/models/user.js returns User.

The complete src/models/user.js file:

```
import {Model, DataTypes} from 'sequelize'

export default (sequelize) => {
    class User extends Model {
        static associate(models) {
            User.RefreshToken = User.hasOne(models.RefreshToken)
            User.Roles = User.hasMany(models.Roles)
        }

        static async hashPassword(password) { }

        static async createNewUser({ email, password, roles, username, firstName,
lastName, refreshToken }) { }
    }

    // Defining the properties of the user model
    User.init({
        email: {
            type: DataTypes.STRING(100),
            allowNull: false,
            unique: true,
            validate: {
                isEmail: {
                    msg: 'Not a valid email address',
                },
            },
        },
        password: {
            type: DataTypes.STRING,
            allowNull: false,
        },
        username: {
            type: DataTypes.STRING(50),
            unique: true,
            validate: {
                len: {
                    args: [2, 50], // min and max number of characters
                    msg: 'Username must contain between 2 and 50 characters',
                },
            },
        },
        firstName: {
            type: DataTypes.STRING(50),
            validate: {
                len: {
                    args: [3, 50], // min and max number of characters
                    msg: 'First name must contain between 3 and 50 characters',
                },
            },
        },
        lastName: {
            type: DataTypes.STRING(50),
            validate: {
                len: {
                    args: [3, 50], // min and max number of characters
                    msg: 'Last name must contain between 3 and 50 characters',
                },
            },
        },
    }, { sequelize, modelName: 'User' }
    )
    return User;
}
```

## 24. Creating the User model (Part 2)

We are first going to work on the hashPassword function in src/models/user.js, which takes a password and hashes it with the bcrypt algorithm. We need to import Bcrypt.

```
import bcrypt from 'bcrypt';
```

```
// Takes a password and hashes it with the bcrypt algorithm
       static async hashPassword(password) {
           return bcrypt.hash(password, environment.saltRounds) // first argument is a string which will be our password and the second argument is going to be the saltOrRounds, which can be a string or number and it returns a Promise that resolves to a string, which will be the hash string. We will get the value for the SaltOrRounds by importing environment from '../config/environment' at the top of the file.
        }
```

### Implement a Hook

What is a hook? It's basically some code that runs after a certain action or before a certain action.

This is the hook that we added to src/models/user.js

```
   // This hook is going to guarantee that every user we save, the password is going to be hashed, so we will NEVER be saving a plain text password.
   User.beforeSave(async (user, options) => {
       const hashedPassword = await User.hashPassword(user.password)
       user.password = hashedPassword
   })
```

In the init function in src/models/user.js, before the return, we add a hook where we say User.beforeSave(), where before we save a new user, we're going to say that we have this callback. Inside the callback, we're going to hash the password, so we'll say const hashedPassword = await User.hashPassword and we're going to pass the user a password, (user.password), and we're going to say that the user.password needs to be equal to the hashedPassword.

### Creating an Instance Method

That means that if we create an instance of a user, then that instance is going be able to use this method.

In src/models/user.js:
```
   // user = await User.findOne({where: {email: 'test@example.com'}})
   // user.email, user.username, user.firstName, user.lastName, user.password (even if the password is hashed, we don't want to expose that password to anyone. When we need the password, we're going to pass in scope, and say, "hey, include in the query the password but otherwise, we're not going to include the password"  )
   // user.comparePasswords('test123#') // if this is the same as the hashed password, true is returned otherwise false is returned
   User.prototype.comparePasswords = async function (password) {
       return bcrypt.compare(password, this.password)  // returns a promise of whether the passed password is the same as the hashed password
   }
```

## Creating Scopes

Let's try to create some scopes in src/models/user.js:

The default scope excludes the password. There is going to be a time that we are going to need our password,  and that is going to be for the Login controller. So for that, we create a scope that is called 'withPassword', and then we assign a value that is an object that has a key called 'attributes' that has a value that is an object with a the key 'include' and a value that is a list with one item - the string 'password'.
```
{
       sequelize,
       modelName: 'User',
       defaultScope: { attributes: { exclude: ['password'] } },
       scopes: {
           withPassword: {
               attributes: { include: ['password'] }
           }
       }
   }
```

# 25. Creating the User model (Part 3)

User model is pretty complex. We need to implement the createNewUser method in the file src/models/user.js. We're going to create a new user with roles, a refresh token, and with all of the properties from the user model. We need to do multiple insertions. We need to do an insertion into the user table, another insertion to the refreshTokenTable and multiple insertions to the Roles table because we can have multiple roles.

If one of these insertions fails, then we want the whole thing to fail. This is where transactions come in. Transactions allow us to rollback all of these changes and if all of these changes / insertions were successful, then we're just going to commit the rollback.

Remember that in src/database/index.js, we are using this namespace in order to have managed transactions:

```
        const namespace = cls.createNamespace('transactions-namespace');
        Sequelize.useCLS(namespace);
```

So now in the createNewUser method in src/models/user.js, we `return sequelize.transaction`. `sequelize` is available here because we have `export default (sequelize)` towards the top of the file.

`sequelize.transaction` expects a callback function, so we write the callback and inside of it we put all of the logic for creating a new user.

```
        static async createNewUser({
            email,
            password,
            roles,
            username,
            firstName,
            lastName,
            refreshToken
        }) {
            return sequelize.transaction(async() => {
                let rolesToSave = []; // empty array

                // roles = ["customer", "admin"]
                //rolesToSave = [{role: "customer"}, {role: "admin"}]
                if (roles && Array.isArray(roles)) { // if roles is defined and if
roles is an array
                    rolesToSave = roles.map(role => ({ role })); // rolesToSave =
roles.map and we're going to convert the role to an object
                }

                await User.create({
                    email,
                    password,
                    username,
                    firstName,
                    lastName,
                    RefreshToken: { token: refreshToken }, // this is how we create an
 associated refreshToken with this user. We pass an object specifying the token.
                    Roles: rolesToSave, // needs to be an array of objects; the
 parameter roles is just going to be an array of strings, so we need to convert that
 array of string to this array of objects
                }, {
                    include: [User.RefreshToken, User.Roles]
                })

                // we have added all of the properties to the users table
                // we need to find a way to create this user with roles and a
 refreshToken and that's where User.RefreshToken = User.hasOne(models.RefreshToken) and
 User.Roles = User.hasMany(models.Roles) are going to help us.

            })
        }
    }
```

## 26. Defining the Role Model

Create a file in src/models called role.js.

After committing the code for role.js to GitHub, I changed role to task in both the src/models/task.js (formerly called src/models/role.js) file and the src/models/user.js file.

## 27. Adding Refresh Token Model

We create a new file called src/models/refresh-token.js, and can basically copy and paste from our file src/models.tasks.js. We then need to make sure we change Task to RefreshToken and task to token in the src/models/refresh-token.js file.

## 28. Inspecting the new tables with DBeaver

Towards the end of the src/database/index.js file we have the line of code shown below that will create the table if it does not exist.

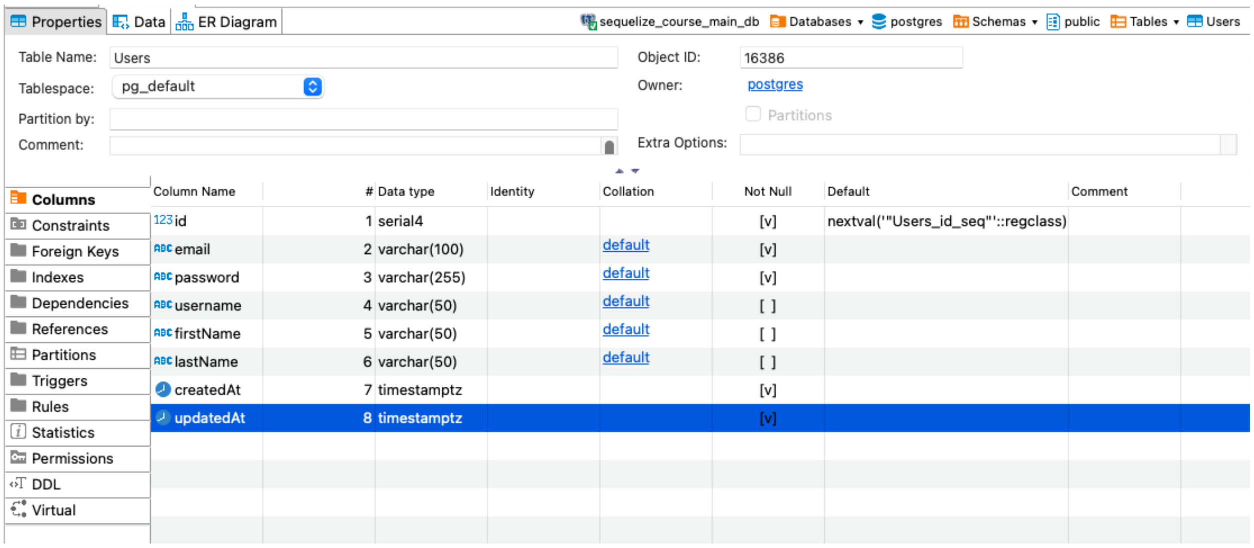
```
       // Sync the models
       await this.sync();
```

We can start up docker using `docker-compose up -d` and then run the command `npm run dev`.

## In Dbeaver

In Dbeaver, we see that the tables get created because the `sync()` was successful.

If we click on the users, and look at the properties, we see that we have the following properties: id, email, password, username, firstName, lastName, createdAt, updatedAt

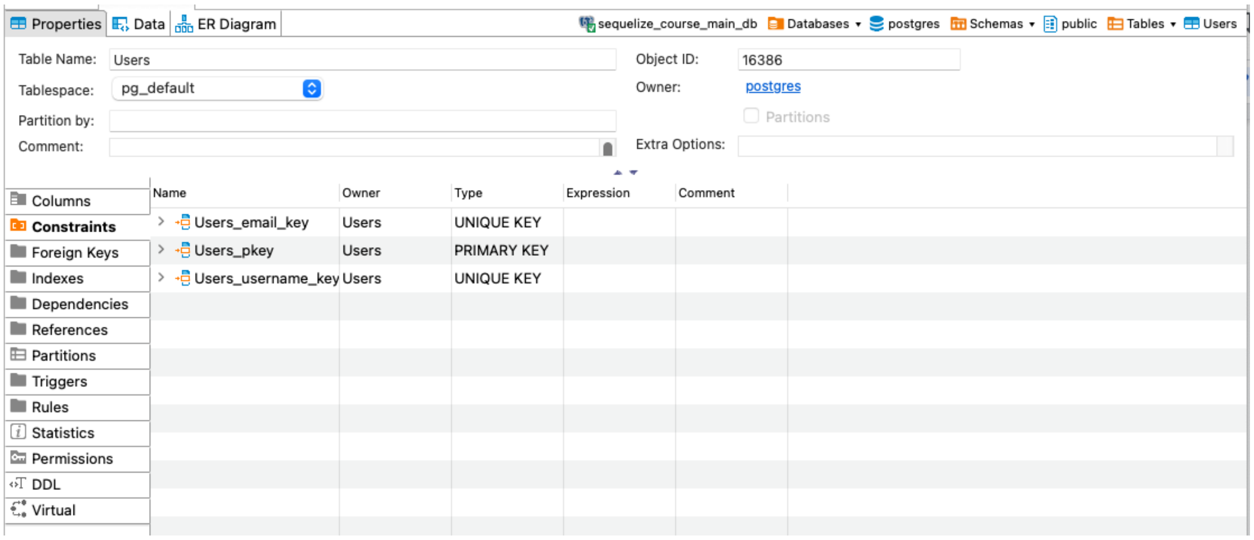


| | |
|---|---|
| Table Name: | Users |
| Tablespace: | pg_default |
| Partition by: | |
| Comment: | |
| Object ID: | 16386 |
| Owner: | postgres |
| Partitions | |
| Extra Options: | |

| Column Name | # | Data type | Identity | Collation | Not Null | Default | Comment |
|---|---|---|---|---|---|---|---|
| id | 1 | serial4 | | | [v] | nextval('"Users_id_seq"'::regclass) | |
| email | 2 | varchar(100) | | default | [v] | | |
| password | 3 | varchar(255) | | default | [v] | | |
| username | 4 | varchar(50) | | default | [ ] | | |
| firstName | 5 | varchar(50) | | default | [ ] | | |
| lastName | 6 | varchar(50) | | default | [ ] | | |
| createdAt | 7 | timestamptz | | | [v] | | |
| updatedAt | 8 | timestamptz | | | [v] | | |

The id, createdAt, and updatedAt fields were added automatically by sequelize.

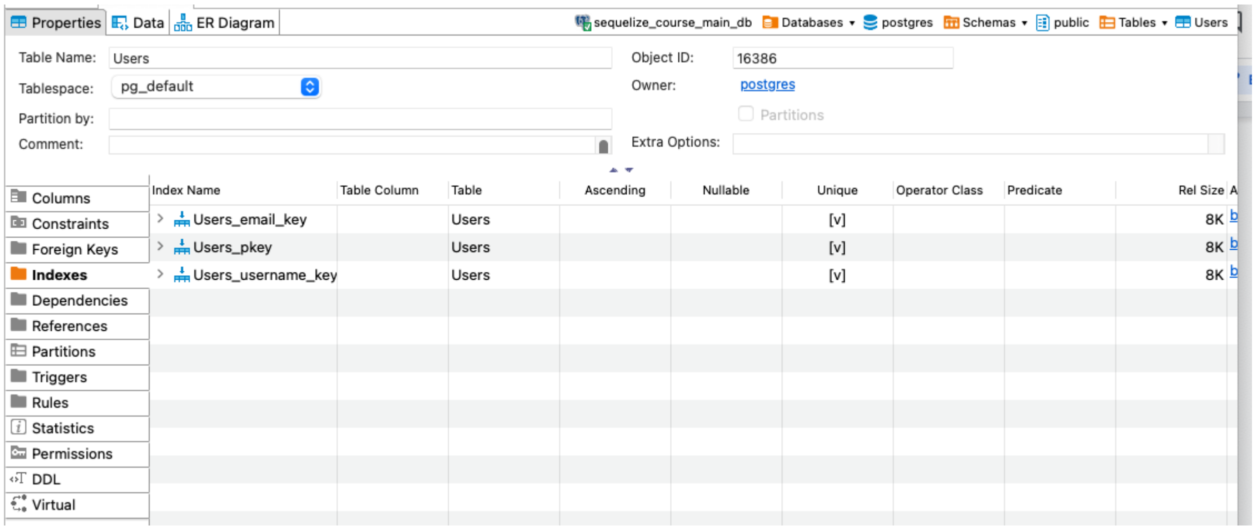
We can take a look at the constraints. We can see that the User_email_key is unique, the Users_pkey is the primary key, and the Users_username_key is unique.



| | |
|---|---|
| Table Name: | Users |
| Tablespace: | pg_default |
| Partition by: | |
| Comment: | |
| Object ID: | 16386 |
| Owner: | postgres |
| Partitions | |
| Extra Options: | |

| Name | Owner | Type | Expression | Comment |
|---|---|---|---|---|
| Users_email_key | Users | UNIQUE KEY | | |
| Users_pkey | Users | PRIMARY KEY | | |
| Users_username_key | Users | UNIQUE KEY | | |

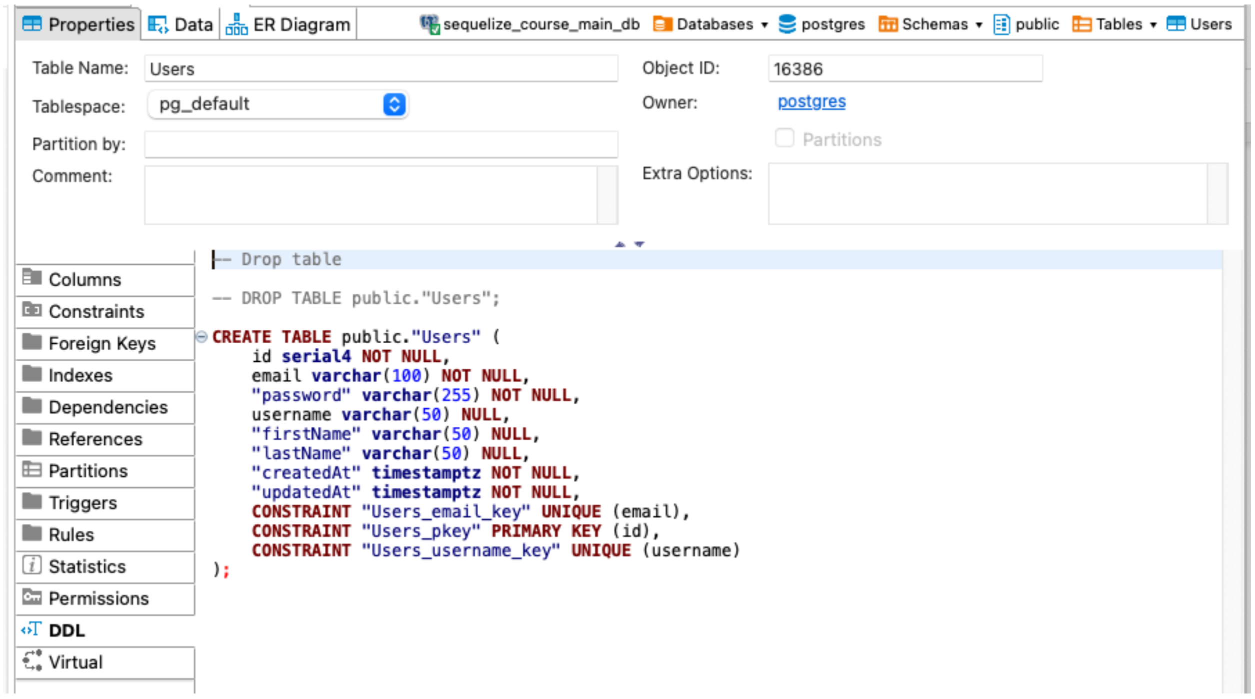
With the indexes, we have Users_email_key as an index because it is a unique column, Users_username_key as an index because it is a unique column, and obviously the Users_pkey is a unique column and has an index because it is a primary key.



|  |  |  |  |
|---|---|---|---|
| Table Name: | Users | Object ID: | 16386 |
| Tablespace: | pg_default | Owner: | postgres |
| Partition by: |  |  | Partitions |
| Comment: |  | Extra Options: |  |

| Index Name | Table Column | Table | Ascending | Nullable | Unique | Operator Class | Predicate | Rel Size |
|---|---|---|---|---|---|---|---|---|
| Users_email_key |  | Users |  |  | [v] |  |  | 8K |
| Users_pkey |  | Users |  |  | [v] |  |  | 8K |
| Users_username_key |  | Users |  |  | [v] |  |  | 8K |

In the DDL we can see the query that Sequelize needed to do in order to create this table. The same can be done with the Tasks, and the RefreshTokens.



|  |  |  |  |
|---|---|---|---|
| Table Name: | Users | Object ID: | 16386 |
| Tablespace: | pg_default | Owner: | postgres |
| Partition by: |  |  | Partitions |
| Comment: |  | Extra Options: |  |

```
-- Drop table

-- DROP TABLE public."Users";

CREATE TABLE public."Users" (
    id serial4 NOT NULL,
    email varchar(100) NOT NULL,
    "password" varchar(255) NOT NULL,
    username varchar(50) NULL,
    "firstName" varchar(50) NULL,
    "lastName" varchar(50) NULL,
    "createdAt" timestamptz NOT NULL,
    "updatedAt" timestamptz NOT NULL,
    CONSTRAINT "Users_email_key" UNIQUE (email),
    CONSTRAINT "Users_pkey" PRIMARY KEY (id),
    CONSTRAINT "Users_username_key" UNIQUE (username)
);
```

For Tasks, what is interesting is that in the DDL we have this constraint called "Tasks_UserId_fkey" and it specifies things to do On Delete and On Update.

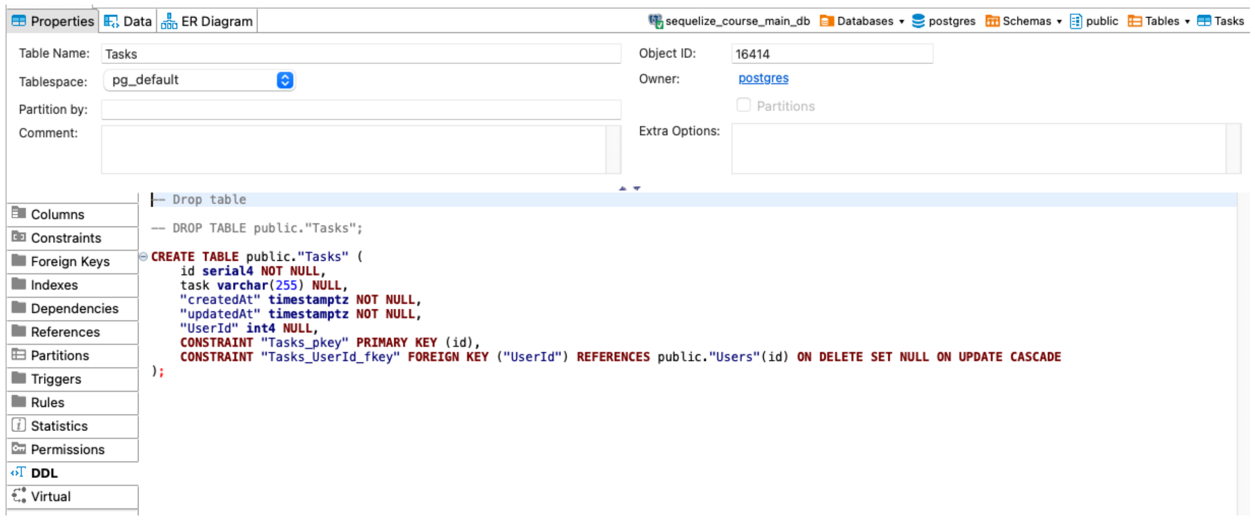


Table Name: Tasks
Tablespace: pg_default
Partition by:
Comment:
Object ID: 16414
Owner: postgres
Partitions
Extra Options:

```
-- Drop table

-- DROP TABLE public."Tasks";

CREATE TABLE public."Tasks" (
    id serial4 NOT NULL,
    task varchar(255) NULL,
    "createdAt" timestamptz NOT NULL,
    "updatedAt" timestamptz NOT NULL,
    "UserId" int4 NULL,
    CONSTRAINT "Tasks_pkey" PRIMARY KEY (id),
    CONSTRAINT "Tasks_UserId_fkey" FOREIGN KEY ("UserId") REFERENCES public."Users"(id) ON DELETE SET NULL ON UPDATE CASCADE
);
```

All of these things should not be determined by Sequelize because we don't have any mechanism, for example, to rollback on migration. This is where the concept of migrations come in. When we want to create a new table, when we want to create a new column or a new index, we need a migration. We cannot let Sequelize determine what is going to happen because migrations are a way to have a record history of what changes were made to the database. This is great, but we need migrations and that is the thing that we're going to do next in this course.

## 29. Configuring the Sequelize CLI

We installed the dependency sequelize-cli at the beginning of this course. In order to use it, we actually need to configure it first. Therefore, we need to create a .sequelizerc file, which we do by typing the following command in the terminal:

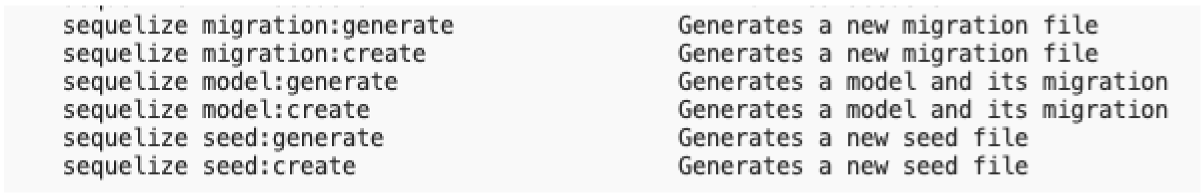
```
touch .sequelizerc
```

In .sequelizerc:

```
const path = require('path');

module.exports = {
   'config': path.resolve('src', 'config', 'database.js'), // This means that all of the configuration is going to be taken from the database object located in the file src/config/database.js
   'models-path': path.resolve('src', 'models'),
   'migrations-path': path.resolve('src', 'database', 'migrations')
}
```

We haven't created the 'migrations-path' yet, so inside of the database folder, we will create a folder called mgirations.

In the terminal we can type `npx sequelize` which will give you a list of commands for example `sequelize db:migrate, sequelize db:drop`, etc. There are also commands for migrations:



```
  sequelize migration:generate                Generates a new migration file
  sequelize migration:create                  Generates a new migration file
  sequelize model:generate                    Generates a model and its migration
  sequelize model:create                      Generates a model and its migration
  sequelize seed:generate                     Generates a new seed file
  sequelize seed:create                       Generates a new seed file
```

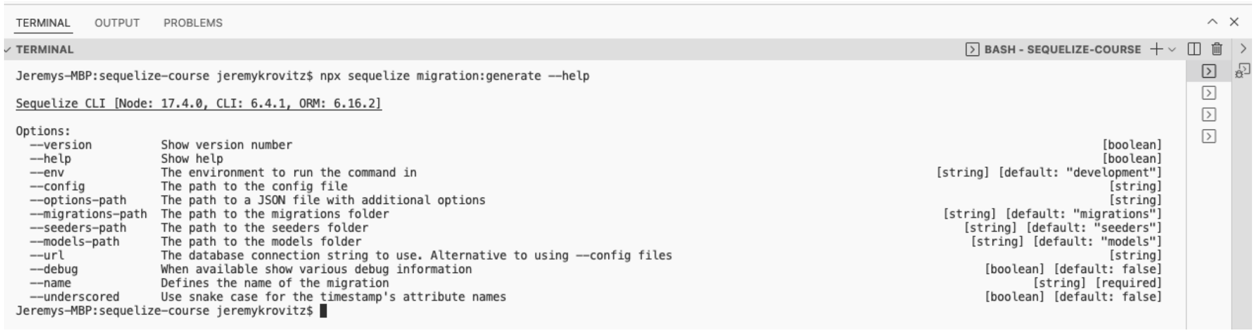
In the next video, we will be using `sequelize migration:generate` to generate a skeleton for the users table and we're going to see how to write this migration.

## 30. Adding User migration

### Generating the Migration File

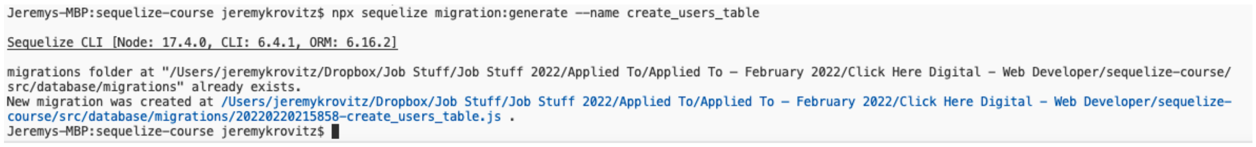
We will be generating our first migration to create the users table. We will run the command in the terminal: `npx sequelize` to see how to generate this migration.

It looks like it's `sequelize migration:generate`, so in the terminal we run: `npm sequelize migration:generate —-help` to see the options. We can pass in the name, which will be the name of the migration. You can pass all of these options basically. We can read all of them if we want, but we will just use the name option.



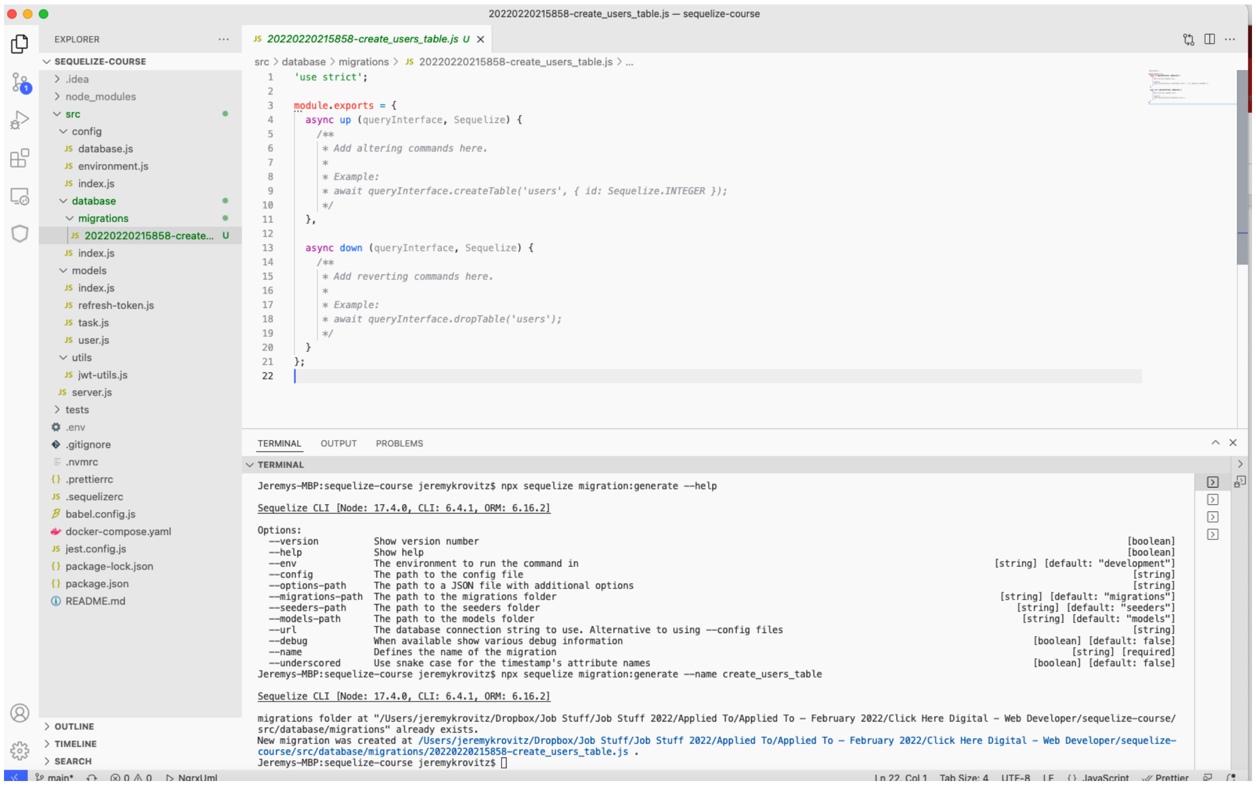
```
Jeremys-MBP:sequelize-course jeremykrovitz$ npx sequelize migration:generate --help

Sequelize CLI [Node: 17.4.0, CLI: 6.4.1, ORM: 6.16.2]

Options:
  --version          Show version number                                                               [boolean]
  --help             Show help                                                                         [boolean]
  --env              The environment to run the command in                      [string] [default: "development"]
  --config           The path to the config file                                                        [string]
  --options-path     The path to a JSON file with additional options                                    [string]
  --migrations-path  The path to the migrations folder                           [string] [default: "migrations"]
  --seeders-path     The path to the seeders folder                                 [string] [default: "seeders"]
  --models-path      The path to the models folder                                   [string] [default: "models"]
  --url              The database connection string to use. Alternative to using --config files         [string]
  --debug            When available show various debug information                   [boolean] [default: false]
  --name             Defines the name of the migration                                      [string] [required]
  --underscored      Use snake case for the timestamp's attribute names               [boolean] [default: false]
Jeremys-MBP:sequelize-course jeremykrovitz$ 
```

We can then type in the terminal: `npx sequelize migration:generate --name create_users_table`



```
Jeremys-MBP:sequelize-course jeremykrovitz$ npx sequelize migration:generate --name create_users_table

Sequelize CLI [Node: 17.4.0, CLI: 6.4.1, ORM: 6.16.2]

migrations folder at "/Users/jeremykrovitz/Dropbox/Job Stuff/Job Stuff 2022/Applied To/Applied To - February 2022/Click Here Digital - Web Developer/sequelize-course/
src/database/migrations" already exists.
New migration was created at /Users/jeremykrovitz/Dropbox/Job Stuff/Job Stuff 2022/Applied To/Applied To - February 2022/Click Here Digital - Web Developer/sequelize-
course/src/database/migrations/20220220215858-create_users_table.js .
Jeremys-MBP:sequelize-course jeremykrovitz$
```

This automatically generated a skeleton for us:



```js
'use strict';

module.exports = {
  async up (queryInterface, Sequelize) {
    /**
     * Add altering commands here.
     *
     * Example:
     * await queryInterface.createTable('users', { id: Sequelize.INTEGER });
     */
  },

  async down (queryInterface, Sequelize) {
    /**
     * Add reverting commands here.
     *
     * Example:
     * await queryInterface.dropTable('users');
     */
  }
};
```

```
Jeremys-MBP:sequelize-course jeremykrovitz$ npx sequelize migration:generate --help

Sequelize CLI [Node: 17.4.0, CLI: 6.4.1, ORM: 6.16.2]

Options:
  --version          Show version number                                                                   [boolean]
  --help             Show help                                                                             [boolean]
  --env              The environment to run the command in                          [string] [default: "development"]
  --config           The path to the config file                                                            [string]
  --options-path     The path to a JSON file with additional options                                        [string]
  --migrations-path  The path to the migrations folder                               [string] [default: "migrations"]
  --seeders-path     The path to the seeders folder                                     [string] [default: "seeders"]
  --models-path      The path to the models folder                                       [string] [default: "models"]
  --url              The database connection string to use. Alternative to using --config files             [string]
  --debug            When available show various debug information                        [boolean] [default: false]
  --name             Defines the name of the migration                                         [string] [required]
  --underscored      Use snake case for the timestamp's attribute names                   [boolean] [default: false]
Jeremys-MBP:sequelize-course jeremykrovitz$ npx sequelize migration:generate --name create_users_table

Sequelize CLI [Node: 17.4.0, CLI: 6.4.1, ORM: 6.16.2]

migrations folder at "/Users/jeremykrovitz/Dropbox/Job Stuff/Job Stuff 2022/Applied To/Applied To - February 2022/Click Here Digital - Web Developer/sequelize-course/
src/database/migrations" already exists.
New migration was created at /Users/jeremykrovitz/Dropbox/Job Stuff/Job Stuff 2022/Applied To/Applied To - February 2022/Click Here Digital - Web Developer/sequelize-
course/src/database/migrations/20220220215858-create_users_table.js .
Jeremys-MBP:sequelize-course jeremykrovitz$
```

up:async - is basically for migrating
down:async - is basically rollback

## Editing the Migration file

We edit our file Users migration file:

```js
'use strict';

module.exports = {
   async up(queryInterface, Sequelize) {
       await queryInterface.createTable('Users', {
           id: {
               allowNull: false,
               autoIncrement: true,
               primaryKey: true,
               type: Sequelize.INTEGER,
           },
           email: {
               type: Sequelize.STRING(100),
               allowNull: true,
               unique: true,
               // We cannot include validate in a migration; Postgres is not capable
of email validation
           },
           password: {
               type: Sequelize.STRING,
               allowNull: false,
           },
           username: {
               type: Sequelize.STRING(50),
               unique: true,
           },
           firstName: {
               type: Sequelize.STRING(50),
           },
           lastName: {
               type: Sequelize.STRING(50),
           },
           createdAt: {
               allowNull: false,
               type: Sequelize.Date,
           },
           updatedAt: {
               allowNull: false,
               type: Sequelize.Date,
           },
       })
   },

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

### Running the Migration

We're going to run the migration on the test database. If we try to do it right now in the main database, we're going to run into problems.

In the terminal we type the following: `npx sequelize db:migrate --env test`, which basically says, "hey, migrate the database but only for the test environment."

### Rollback a Migration

Type the following in the command line: `npx sequelize db:migrate:undo:all --env test`

## 31. Adding Role (Task) migration

Creating the migration for the task table: `npx sequelize migration:generate —-name create_tasks_table`

We want the main database to match the test database. The sync command must yield the same results as the migrations and vice-versa.

We can also undo one migration by running the following command in the terminal:

```
npx sequelize db:migrate:undo --name 20220220235307-create_tasks_table --env test
```

## 32. Adding Refresh Token migration

Creating the migration for the refresh token table: `npx sequelize migration:generate --name create_refresh_token_table`

## 33. Applying Migrations

To see the status of the migrations, you can type in the terminal: `npx sequelize db:migrate:status`

To migrate to the development database we write the following in the terminal: `npx sequelize db:migrate`

When we run `npm run dev`, it says Connection synced successfully, but in reality, the sync method in src/database/index.js is not going to do anything right now because the tables already exist:

```
    async sync(){
        await this.connection.sync({
            logging: false,
            force: this.isTestEnvironment, // if this.isTestEnvironment equals true
        })

        if (!this.isTestEnvironment) {
            console.log('Connection synced successfully')
        }
    }
```

For testing, this command `force: this.isTestEnvironment, // if this.isTestEnvironment equals true` is going to drop the database and recreate it with the same thing. This is why we need that match between the migrations and the models.

For production, we obviously want the database created with the migrations.

## 34. Small improvement to the User model

In the file src/models/user.js, the following returns undefined:

```
        await User.create({
                    email,
                    password,
                    username,
                    firstName,
                    lastName,
                    RefreshToken: { token: refreshToken }, // this is how we create an associated refreshToken with this user. We pass an object specifying the token.
                    Tasks: tasksToSave, // needs to be an array of objects; the parameter tasks is just going to be an array of strings, so we need to convert that array of string to this array of objects
                }, {
                    include: [User.RefreshToken, User.Tasks]
                })
```

We replace `await` with `return`, and the transaction will now return a promise and this basically means that createNewUser is going to return a promise but if we await that promise, we're going to get a newly created user.

```
return User.create({
                    email,
                    password,
                    username,
                    firstName,
                    lastName,
                    RefreshToken: { token: refreshToken }, // this is how we create an
associated refreshToken with this user. We pass an object specifying the token.
                    Tasks: tasksToSave, // needs to be an array of objects; the
parameter tasks is just going to be an array of strings, so we need to convert that
array of string to this array of objects
                }, {
                    include: [User.RefreshToken, User.Tasks]
                })
```

There is a drawback of returning the User the way that it is done in the above code in that the password is not going to be hidden. We have two options here:

One option is to add async and assign await User.create(...) to a variable called result:

```
            return sequelize.transaction(async() => {
                let tasksToSave = []; // empty array

                // tasks = ["task1", "task2"]
                //rolesToSave = [{task: "task1"}, {task: "task2"}]
                if (tasks && Array.isArray(tasks)) { // if tasks is defined and if
tasks is an array
                    tasksToSave = tasks.map(task => ({ task })); // rolesToSave =
tasks.map and we're going to convert the task to an object
                }

                const result = await User.create({
                    email,
                    password,
                    username,
                    firstName,
                    lastName,
                    RefreshToken: { token: refreshToken }, // this is how we create an
associated refreshToken with this user. We pass an object specifying the token.
```

```
                    Tasks: tasksToSave, // needs to be an array of objects; the
parameter tasks is just going to be an array of strings, so we need to convert that
array of string to this array of objects
                }, {
                    include: [User.RefreshToken, User.Tasks]
                })

                // we have added all of the properties to the users table
                // we need to find a way to create this user with tasks and a
refreshToken and that's where User.RefreshToken = User.hasOne(models.RefreshToken) and
User.Tasks = User.hasMany(models.Tasks) are going to help us.

            })
```

A better way to hide the password is with hooks. We can add this hook right before we return User.

```
   User.afterCreate((user, options) => {
       delete user.dataValues.password
   })

```

## 35 Adding User model tests (Part 1)

We start by creating a folder called models inside of the tests folder. We then create a new file called user.test.js.

user.test.js:

```
import TestHelpers from '../tests-helpers'
import models from '../../src/models'

describe('User', () => {

   // set up
   beforeAll(async() => {
       await TestHelpers.startDb()
   })

   // tare down
   afterAll(async() => {
       await TestHelpers.stopDb()
```

```
  })

  // drops the tables before each test and re-creates them using the models
definition, not the migration but the model's definition. That's why it's very
important to match the mgirations with the models definitions.
  beforeEach(async() => {
      await TestHelpers.syncDb()
  })

  describe('static methods', () => {
      describe('hashPassword', () => {
          it('should hash the password passesd in the arguments', async() => {
              const { User } = models
              const password = 'Test123#'
              const hashedPassword = await User.hashPassword(password)
              expect(password).not.toEqual(hashedPassword);
          })
      })

      describe('createNewUser', () => {
          it('should create a new user successfully', async() => {
              const { User } = models
              const data = {
                  email: 'test@example.com',
                  password: 'Test123#',
                  tasks: ['work on report', 'work on write up'],
                  username: 'test',
                  firstName: 'Jeremy',
                  lastName: 'Krovitz',
                  refreshToken: 'test-refresh-token'
              }
              const newUser = await User.createNewUser(data)
              const usersCount = await User.count()
              expect(usersCount).toEqual(1)
              expect(newUser.email).toEqual(data.email)
              expect(newUser.password).toBeUndefined
              expect(newUser.username).toEqual(data.username)
              expect(newUser.firstName).toEqual(data.firstName)
              expect(newUser.lastName).toEqual(data.lastName)
              expect(newUser.RefreshToken.token).toEqual(data.refreshToken)
              expect(newUser.Tasks.length).toEqual(2)
```

```
               const savedTasks = newUser.Tasks.map(savedTask =>
savedTask.task).sort(); // each one of these tasks elements, because remember, tasks
is an array of tasks objects,, so each saved tasks is a task object and each task
object you can call this task method
               expect(savedTasks).toEqual(data.tasks.sort()) // we use sort to make
ure that the two arrays are equal
           })
       })
   })
})
```

## 36. Adding User model tests (Part 2)

Add more tests for the createNewUser method. We check to see if the test passes by typing `npm run test:watch` in the terminal.

It is better to make things fail one at a time, as it will facilitate better testing.

## 37. Adding User model tests (Part 3)

Here we are writing tests for the scopes:

```
  describe('scopes', () => {
       let user

       beforeEach(async() => {
           user = await TestsHelpers.createNewUser()
       })

       describe('defaultScope', () => {
           it('should return a user without a password', async() => {
               const { User } = models
               const userFound = await User.findByPk(user.id)
               expect(userFound.password).toBeUndefined()

           })
       })

       describe('withPassword', () => {
           it('should return a user with the password', async() => {
               const { User } = models
               const userFound = await User.scope('withPassword').findByPk(user.id)
               expect(userFound.password).toEqual(expect.any(String))
           })
       })
   })
```

## 38. Adding User Tests (Part 4)

We are writing tests for the instance methods:

```
describe('instance methods', () => {
       describe('comparePasswords', () => {
           let password = 'Test123#'
           let user

           beforeEach(async() => {
               user = await TestHelpers.createNewUser({ password })
           })
           it('should return true if the password is correct', async() => {
               const { User } = models
               const userFound = await User.scope('withPassword').findByPk(user.id)
               const isPasswordCorrect = await userFound.comparePasswords(password)
               expect(isPasswordCorrect).toEqual(true)
           })

           it('should return false if the password is incorrect', async() => {
               const { User } = models
               const userFound = await User.scope('withPassword').findByPk(user.id)
               const isPasswordCorrect = await
userFound.comparePasswords("invalidpassword")
               expect(isPasswordCorrect).toEqual(false)
           })
       })
   })
```

## 39. Adding User model tests (Part 5)

We will be testing the hooks as well. There is a bug in the code where it will try to hash a password even if a password is not passed.

Before fixing the bug:
```
  // This hook is going to guarantee that every user we save, the password is going to
be hashed, so we will NEVER be saving a plain text password.
   User.beforeSave(async(user, options) => {
       const hashedPassword = await User.hashPassword(user.password)
       user.password = hashedPassword
   })
```

Fixed the bug (add an if statement to make sure a password is passed):
```
   // This hook is going to guarantee that every user we save, the password is going
to be hashed, so we will NEVER be saving a plain text password.
   User.beforeSave(async(user, options) => {
       if (user.password) {
           const hashedPassword = await User.hashPassword(user.password)
           user.password = hashedPassword
       }
   })
```

## 40. Adding Task model tests

We can test by creating a new user with some tasks and then delete the user or update it and see if this indeed is updating the id in this case or deleting those records in the database.

Create the file task.test.js in Tests/Models.

Primary keys should NEVER change. Changing a primary key at runtime is very very dangerous.

# Section 5: The Express app

## 41. Create the Express app

Inside of the src folder we create a file called app.js:
```
import express from 'express'
import environment from './config/environment'
import logger from 'morgan'

export default class App {
   constructor() {
       this.app = express()
       this.app.use(logger('dev', {
           skip: (req, res) => environment.nodeEnv === 'test'
       }))
       this.app.use(express.json()) // parse json
       this.app.use(express.urlencoded({ extended: true }))
       this.setRoutes()

   }

   setRoutes() {}

   getApp() {
       return this.app;
   }

   listen() {
       const { port } = environment
       this.app.listen(port, () => {
           console.log('Listening at port ${port}')
       })
   }
}
```

We also add a test in tests/tests-helpers.js:

```
static getApp() {
        const App = require('../src/app').default
        return new App().getApp()
    }

```

## 42. Creating an Errors middleware

We will create a very simple middleware, which is going to be an error universe. If something goes unexpectedly wrong in one of our controllers, then we will catch the errors in our middleware and return the response with 500, which means internal service error.

In src we create a new folder called middlewares. Inside of middlewares, we create a file called errors.js.

The errors.js file looks like this:

```
export default function errorsMiddleware(err, req, res, next) {
   console.error('Error in erros middleware\n', err.stack)
   res.status(500).send({ success: false, message: err.message })
}
```

Then in src/app.js, we add the following:

At the top of the file:

```
import errorsMiddleware from './middlewares/errors'
```

In the setRoutes() method:

```
   setRoutes() {
       this.app.use(errorsMiddleware)
   }
```

## 43. Creating a wrapper to handle async code

We haven't created any tests for this errors middleware yet because we can do these tests on the controller, which will be better. If we wanted to make a test for only the function in src/middlewares/errors.js, we would have to do a bunch of mocks.

We create a file in the src/utils folder called asyncWrapper.js:

```
export default function asyncWrapper(callback) {} //When we have async code in our
controllers, which is like 99% of the time, if we have any error that is thrown there,
then right now that error is not caught by the errors middleware unless we have a way
to catch the error. This is not going to be a problem in Express 5. In express 5, this
will be implemented natively, so you don't have to do anything if you are in the
future and you are already using express 5.

return function(req, res, next) {
   callback(req, res, next).catch(next) // This will make sure that we catch any error
that is thrown by an async function
}
```

## 44. Creating an authentication middleware

We're going to create another middleware in the src/middlewares folder and we're going to call it requiresAuth.js. This middleware is going to help us if we want to verify either the access token or the refresh token depending on what we want to verify.

# Section 6: Controllers

## 45. Adding structure to the controllers

It is time to add our controllers, so let's create a folder for our controllers within the src folder. Inside of the src/controllers folder, we create a file called index.js. Inside of src/controllers, we create a new folder called v1. Inside of v1, we create another index.js file.

In the working world, it's very possible that there are different versions of the API, and that is why we're creating a v1 folder.  As things advance and new endpoints are added, these may be added within a v2 route.

## 46. Adding the register controller

We will start building our four controllers, which we do in the src/controllers/v1 folder. The first controller is the register controller, which we call register.js.

We have almost created everything in the model, which is good because we need to have thin controllers. All of the heavy logic should be in the models.

## 47. Adding tests for the register controller

Inside the tests folder, we'll create a new folder called controllers. Inside the controllers folder, we will create a folder called v1, inside v1, we will create a file called register.test.js.

## 48. Adding the login controller

We are going to create our next controller for our login endpoint inside of src/controller/v1. The file will be called login.js.

### Lazy Loading vs. Eager Loading

While lazy loading delays the initialization of a resource, eager loading initializes or loads a resource as soon as the code is executed. Eager loading also involves pre-loading related entities referenced by a resource.

## 49. Adding another test helper

We're going to add a new test helper in tests/tests-helpers.js, and it is going to be called registerNewUser().

## 50. Adding tests for the login controller

We will create a file called login.test.js in tests/controllers/v1. There were some issues with the login test. Moving onward.

## 51. Adding a token controller

Create a token controller where its function is going to be to return a new access token provided we have a refresh token.

## 52. Adding tests for the token controller (Part 1)

We go to the test/controllers/v1 folder. We click to add a new file and create a new file called token.test.js.

## 53. Adding tests for the token controller (Part 2)

We will be working on the tests for the token controller.

## 54. Adding the logout controller

Adding the last controller. We create a file called logout.js in the folder called controllers/v1.

## 55. Adding tests for the logout controller

Create a file in tests/controllers/v1 called logout.tests.js.

## 56. Looking at the coverage of our tests

We can see the coverage of our tests by running: `npm run test:coverage`. This command will run all of our tests, and it creates a coverage folder.

It is fine not to cover the `listen` method because in the tests, we're not calling this method. That is expected because we don't want to listen on our ports for our express app.

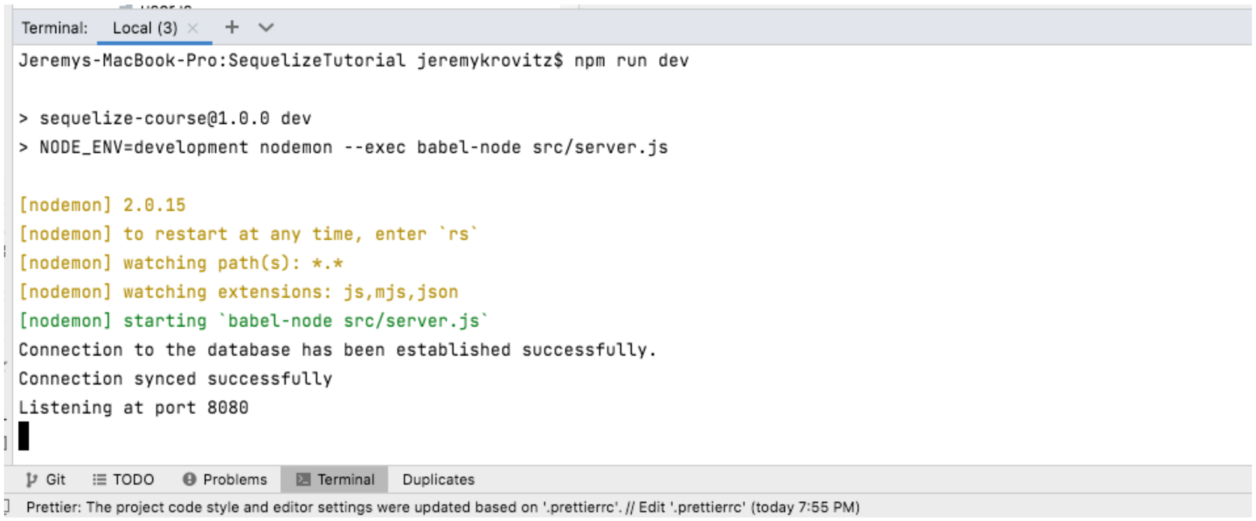
We don't want to test the test-helpers.

## 57. Listening for Connections

We need to put our app on the server, which we do in the server.js file by writing the following:

```
const App = require('./app').default
const app = new App();
app.listen()
```

Now we can type in the terminal `npm run dev` and we should now see that we're listening on port 8080



```
Jeremys-MacBook-Pro:SequelizeTutorial jeremykrovitz$ npm run dev

> sequelize-course@1.0.0 dev
> NODE_ENV=development nodemon --exec babel-node src/server.js

[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `babel-node src/server.js`
Connection to the database has been established successfully.
Connection synced successfully
Listening at port 8080
```

Prettier: The project code style and editor settings were updated based on '.prettierrc'. // Edit '.prettierrc' (today 7:55 PM)

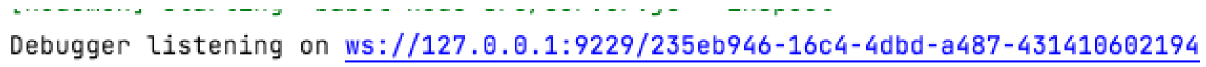
## 58. Testing the app manually with Postman

See postman tests.

## 59. Learning how to use the debug script

We can run the following from the terminal: `npm run debug`

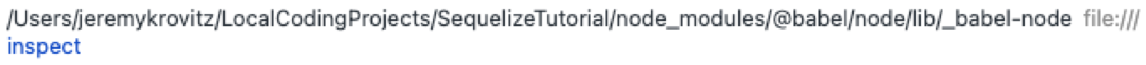
You can see in the output that the debugger is listening:



```
Debugger listening on ws://127.0.0.1:9229/235eb946-16c4-4dbd-a487-431410602194
```

We can go to: chrome://inspect/#devices. Once there, if you don't see Remote Target,, you can click the button that says Configure next to Discover network Targets and enter localhost:<specify port debugger is listening on (without angle brackets)>

We can then click inspect.



/Users/jeremykrovitz/LocalCodingProjects/SequelizeTutorial/node_modules/@babel/node/lib/_babel-node file:///
inspect

This will give us access to the console that we have inside of our ide.

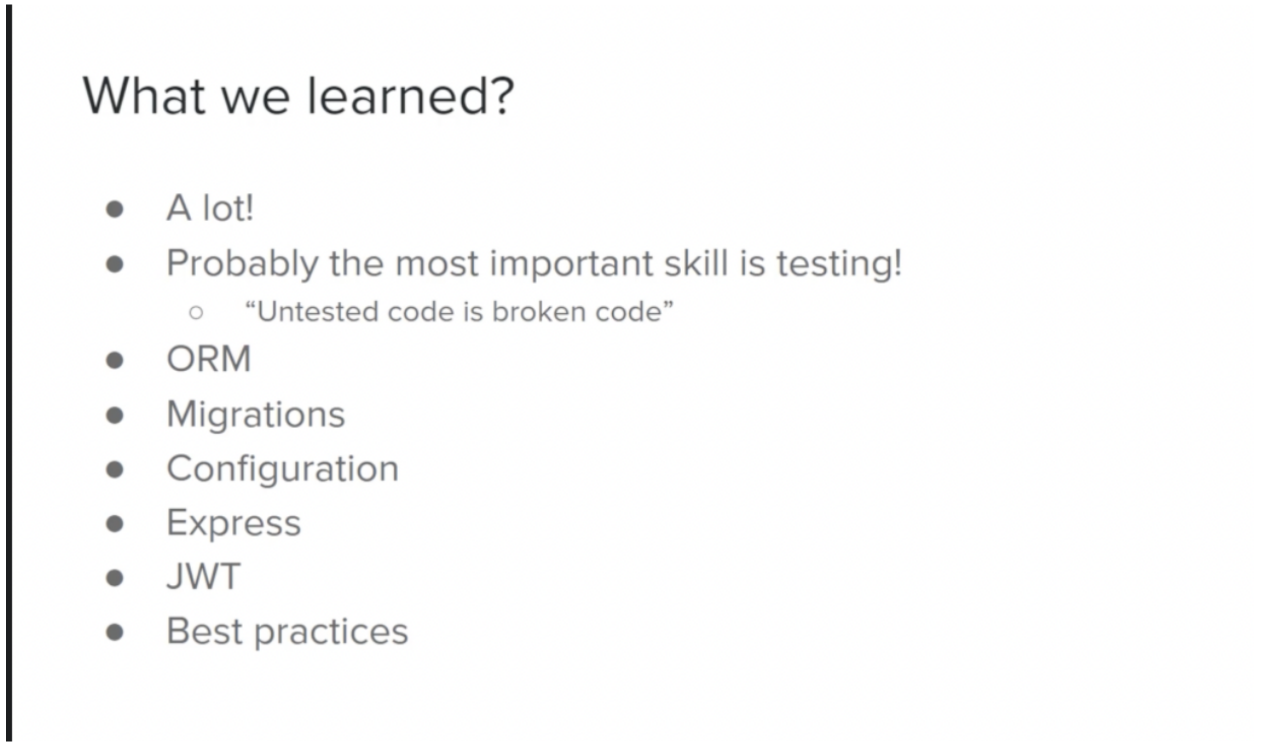
We can add `debugger;` in a part of our script, make a request in Postman, and then be able to debug the specific code.

*** I want to get a better understanding of using the debugger in this way.

## Section 7: Conclusion

### 60. Conclusion

This is the end of this course.



What we learned?

- A lot!
- Probably the most important skill is testing!
  - "Untested code is broken code"
- ORM
- Migrations
- Configuration
- Express
- JWT
- Best practices

## Additional Notes While Working on Project

To run the seeders located at a specific path, type the following in the terminal:

```
npx sequelize-cli db:seed:all --seeders-path src/seeders
```