Set up

1. **Create a new project** directory (ex: authSecurity)
2. Run: npx express-generator --git authApp //must be in the backend app.

This will generate authApp with multiple folders:

bin / public / routes / views / .gitignore / app.js / package.json

package.json includes

Text

Description automatically generated

1. Run: npm install

This generate dependencies that were created by step 2, those are

(This will create: node\_modules / package-lock.json)

1. Run: npx express-generator

Text

Description automatically generated

1. Run: npm start //works same nodemon to run the file or the app

On browser:

<http://localhost:3000/>

must show:

Graphical user interface, text, application, chat or text message, email

Description automatically generated

Then run

1. npx express-generator –-ejs

you seedestination is not empty, continue? [y/N] y

then you get

Text

Description automatically generated

Run

1. npm install bcrypt. //installs a package for hashing the password that entered by the user

Text

Description automatically generated

//The bcrypt hashing function allows **us to build a password security platform that scales with computation power and always hashes every password with a salt**.

bcrypt

It’s a library for NodeJS

on user.app

const bcrypt = require('bcrypt')

**CREAT DATABASE**

run those commands: or go down after the command history to follow the steps

1. npm i sequelize
2. npm i pg

Text

Description automatically generated

then run

1. npx sequelize-cli init. // this will create the

To create an empty project.

This will create following folders

* config.json, contains config file, which tells CLI how to connect with database
* models, contains all models for your project
* migrations, contains all migration files
* seeders, contains all seed files

[go to config and change some parts](#_Reading) (click me)

Text

Description automatically generated

Text

Description automatically generated

1. npx sequelize-cli db:create. //use to create an empty database.

#### **Creating the Model**

Run the following command to create the User model:

npx sequelize-cli model:generate --name User --attributes firstName:string,password:string,email:string

(step 5 creates migration file, we can customize it based on the model file)

#### **Running the migration**

The up function contains the code for performing the migration, creating the table. down is like an "undo" - it will drop the table.

npx sequelize-cli db:migrate

#### **Seeding the Database**

Sequelize can create a file that has "seed" data - essentially placeholder data so that there's something in the database.

npx sequelize-cli seed:generate --name user

This will create a new file in your seeders directory. Unlike the other files Sequelize has created for us, we'll need to customize it more heavily.

Past this inside then right click then format with (prettier)

Then change lastName to [password] to match the models/user

"use strict";

module.exports = {

up: async (queryInterface, Sequelize) => {

await queryInterface.bulkInsert(

"Users",

[

{

firstName: "Annie",

password: "Easley",

email: "ajeasley@nasa.gov",

createdAt: new Date(),

updatedAt: new Date(),

},

],

{}

);

},

down: async (queryInterface, Sequelize) => {

return queryInterface.bulkDelete("Users", null, {});

},

};

#### **9- Running the Seed File**

After customizing our seed file, let's run it:

npx sequelize-cli db:seed:all

1. **Import Sequelize and our User model using the require() function.**

THIS MUST BE ADDED IN THE USER FILE

const Sequelize = require('sequelize');

const { User } = require('./models');

The User model generated by Sequelize has methods for all of the CRUD operations we would want:

* .create()
* .findAll()
* .findByPk()
* .update()
* .destroy()

#### **Adding routes (Creating a New User)**

Ex:

router.get('/user', async (req, res) => {

const users = await User.findAll({

attributes: ['email']

});

res.json(users);

console.log(users)

});

To run the file: (nodemon users.js)

Working on a template using html or ejs

1. **to add the template in index.ejs, (views/index.ejs):**

<https://getbootstrap.com/docs/5.1/examples/sign-in/>

inspect => get the html source





Change: (must add them) / the rest are extra and optional

form <form action="/login" method="post" >

* + Username

type="text" name="username"

<label for="floatingInput">Username</label>

* + password
* <input type="password" name="password"

<label for="floatingPassword">Password</label>

* + Button
* type="submit"

**on user.js,**

**add this route**

router.post("/login", async (req, res, next) => {

const {username, password} = req.body

const users = await User.findOne({

where: {

firstName: username *//firstName is the column name in our database*

}

});

res.json(users)

console.log(users)

});

**To run the page,**

* + **on terminal: nodemon users.js**
  + **on browser:** [**http://localhost:3000/user**](http://localhost:3000/user) **or** [**http://localhost:3000/**](http://localhost:3000/)
  + **username / password (get them from SQL database)**

**Graphical user interface, text, application, email

Description automatically generated**

**Note: in my migration and seeds, I set the username as firstName (so the username in login is the firstName).**

**Commands History:**

1. 6304 npx express-generator --git authApp
2. authApp
3. 6306 npm install
4. 6307 npx express-generator
5. 6308 npm start

6309 cd dcBackEnd

6310 code authApp

6311 npm app.js

6312 npm start

1. 6313 npx express-generator –ejs
2. 6314 npm install bcrypt
3. 6315 npm start
4. 6316 npm i sequelize
5. 6317 npm i pg
6. 6318 npx sequelize-cli init

The rest of commands will be in the lesson down

1. 6319 npx sequelize-cli db:create\n
2. 6320 npx sequelize-cli model:generate --name User --attributes firstName:string,lastName:string,email:string

6321 npx sequelize-cli db:migrate\n

6322 npx sequelize-cli seed:generate --name user

6323 npx sequelize-cli db:seed:all\n

6324 npm start

1. 6325 npx sequelize-cli model:generate --name User --attributes username:string,password:string,email:string

6326 npx sequelize-cli model:generate --name User --attributes username:string,password:integer,email:string

6327 npx sequelize-cli db:migrate\n

6328 npx sequelize-cli seed:generate --name user

6329 npx sequelize-cli init\n

6330 npx sequelize-cli seed:generate --name user

6331 npx sequelize-cli model:generate --name User --attributes username:string,password:integer,email:string

6332 npx sequelize-cli db:migrate

6333 -\tnpm i sequelize

6334 npm i sequelize

6335 npm i pg

6336 npx sequelize-cli init

6337 npx sequelize-cli db:create\n

6338 npx sequelize-cli model:generate --name User --attributes username:string,password:integer,email:string

6339 npx sequelize-cli db:create\n

6340 npx sequelize-cli model:generate --name User --attributes username:string,password:integer,email:string

6341 seqelize init

6342 ssequelize init

6343 npx sequelize init

6344 npm i sequelize

6345 npm i pg

6346 npx sequelize-cli init

6347 npx sequelize-cli init\n

6348 npx sequelize-cli db:create\n

6349 npx sequelize-cli model:generate --name User --attributes username:string,password:integer,email:string

6350 npx sequelize-cli db:create\n

6351 npx sequelize-cli model:generate --name User --attributes username:string,password:integer,email:string

6352 npx sequelize-cli db:migrate\n

6353 npx sequelize-cli seed:generate --name user

6354 npx sequelize-cli db:seed:all\n

6355 npx sequelize-cli model:generate --name User --attributes username:string,password:integer,email:string

6356 npx sequelize-cli db:seed:all\n

6357 npx sequelize-cli seed:generate --name user1

6358 npx sequelize-cli model:generate --name Use1r --attributes username:string,password:integer,email:string

6359 npx sequelize-cli db:migrate\n

6360 npx sequelize-cli model:generate --name User1 --attributes username:string,password:integer,email:string

6361 npx sequelize-cli seed:generate --name user1

6362 npx sequelize-cli db:migrate\n

6363 npx sequelize-cli init\n

6364 npx sequelize-cli model:generate --name User --attributes username:string,password:integer,email:string\n

6365 npx sequelize-cli db:migrate\n

6366 npx sequelize-cli seed:generate --name user

6367 npx sequelize-cli db:seed:all\n

6368 npm start

6369 cd Desktop

6370 ls

6371 cd ..

6372 code myPortfolio-X

6373 git add .

6374 ls

6375 cd myPortfolio

6376 git status

6377 git add .

6378 git commit -m "commited out testimonial"

6379 git push -u origin feb2

6380 git push -u origin feb24

6381 git init

6382 git add .

6383 git commit -m "first commit"

6384 git remote add origin git@github.com:MDarmussa/Authentication.git

6385 git push -u origin main

Links related:

* <https://expressjs.com/en/starter/generator.html>

express-generator

* <https://expressjs.com/en/starter/generator.html>

sequelize (queries)

* [https://sequelize.org/master/manual/model-querying-basics.html](https://www.npmjs.com/package/bcrypt)

sequelize-cli

* [https://www.npmjs.com/package/sequelize-cli](https://www.npmjs.com/package/bcrypt)

bcrypt-

* <https://www.npmjs.com/package/bcrypt>
* <https://sequelize.org/master/manual/migrations.html>

**Creating DATABASE**

# Reading: Sequelize ORM

## **Reading**

## **Learning Objectives**

1. Install the sequelize-cli and sequelize Node modules
2. Create models and migrations
3. Insert, retrieve, update, and delete records
4. Add model relationships
5. Update the schema using migrations
6. (consider deleting) Perform on-demand fetching of children

Writing SQL by hand for every query can be time-consuming and error-prone. Also, it can be confusing for a developer to shift focus between their server code (in JavaScript or another backend language) and SQL.

To solve these problems, developers created Object-Relational-Mapper (ORM) libraries, which gives you special Objects that represent tables in a database. Instead of writing SQL, you call these Objects' methods to insert, retrieve, update, or delete rows.

In this lesson, we'll look at Sequelize, an ORM written in JavaScript. Sequelize is not the only ORM available for Node.js apps, but it is one of the most popular. It is versatile and works with most popular databases. We'll use it with PostgreSQL and Express.

### **Sequelize and the Sequelize CLI**

Sequelize comes in two parts:

1. sequelize-cli - A tool for managing databases and tables
2. sequelize - A Node module that translates between SQL and JavaScript

#### **Create the Project**

We'll begin by creating an example project. It won't be a complete backend application. Rather, it'll show the pertinent Sequelize code snippets that would power a photo-sharing application.

mkdir photo-sharing-app

cd photo-sharing-app

npm init -y

npm i express sequelize pg

npm i --save-dev nodemon sequelize-cli

touch index.js

#### **Use the Sequelize CLI from**npx

We installed the sequelize-cli program as a dependency, making it available at the path ./node\_modules/.bin/sequelize-cli.

That's more than we want to type. Luckily, the npx command (which was installed with node and npm) will automatically look in our node\_modules directory and find sequelize-cli.

The first thing to do is to initialize the Sequelize portion of our project:

npx sequelize-cli init

This is a separate step from npm init -y; it creates four new Sequelize-specific directories in our project:

* config
* migrations
* models
* seeders

Throughout this lesson, you'll learn the purpose of each directory.

#### **Editing**config/config.json

We'll modify the file config/config.json so that it knows how to access our PostgreSQL database. By default, the config file is set up to connect to three different databases: one for developing locally, one for testing, and one for the live database.

Change each dialect key, giving it the value postgres. Depending on how you configured PostgreSQL on your machine, you might need to change the username from root to your PostgreSQL user. (Ideally, you'd customize the database name to match your project, too. For this example, we'll leave it as-is.)

{

"development": {

"username": "",

"password": null,

"database": "database\_development",

"host": "127.0.0.1",

"dialect": "postgres"

},

"test": {

"username": "root",

"password": null,

"database": "database\_test",

"host": "127.0.0.1",

"dialect": "postgres"

},

"production": {

"username": "root",

"password": null,

"database": "database\_production",

"host": "127.0.0.1",

"dialect": "postgres"

}

}

By default, Sequelize assumes that you're using the development configuration.

#### **Creating the Database**

Use npx sequelize-cli to create an empty database.

npx sequelize-cli db:create

We'll create the tables in the following section.

### **Models and Migrations**

Now that sequelize-cli is configured to work with PostgreSQL, it's time to define our first Model. A Model is an Object that corresponds to a database table. Once the Model is defined, we'll have Sequelize create the table.

The first Model we'll create is the User, which will have the following attributes:

| **Attribute** | **Example value** |
| --- | --- |
| firstName | Annie |
| lastName | Easley |
| email | ajeasley@nasa.gov |

When we worked with PostgreSQL initially, we manually wrote the SQL for creating our database tables. The Sequelize CLI lets you describe and create a table in a single command.

However, creating a table happens in two steps:

1. Putting the table-creation code in a file
2. Running the file

The file with the table-creation code is a migration file. You have an opportunity to inspect and edit the migration before running it.

|  |  |
| --- | --- |
|  | **About migrations**  A migration is a command (or a set of commands) that makes a change to the database - such as creating a table, removing a table, or inserting batches of data.  Every change to the database occurs in two steps:   1. Creating a migration file 2. Running the migration file.   Yes, you can do this by hand, too - but what you want are a series of migrations that you can "play back" to recreate the schema of the database. Or, you can "roll back" to a specific version of your database schema by recreating the database and running only the migrations up to a certain point in time. |

#### **Creating the Model**

Run the following command to create the User model:

npx sequelize-cli model:generate --name User --attributes firstName:string,lastName:string,email:string

|  |  |
| --- | --- |
|  | **No spaces and no line breaks!**  Be careful when typing the model:generate command. It's a bit longer, and you'll be tempted to add spaces or line breaks.  All of that information should go into a single command, and sequelize expects the attributes to be a single string.  After running the command, it prints out the path to the new model and migration files. |

#### **Inspecting the Model**

Take a look at models/user.js. Don't worry if it seems overwhelming. This is primarily for Sequelize to use.

'use strict';

const {

Model

} = require('sequelize');

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

class User extends Model {

/\*\*

\* Helper method for defining associations.

\* This method is not a part of Sequelize lifecycle.

\* The `models/index` file will call this method automatically.

\*/

static associate(models) {

// define association here

}

};

User.init({

firstName: DataTypes.STRING,

lastName: DataTypes.STRING,

email: DataTypes.STRING

}, {

sequelize,

modelName: 'User',

});

return User;

};

The main things to pay attention to are the field names (lines 17-19 in this example). Make sure that there aren't any typos, since we'll be writing this to the database in the next section.

#### **Running the migration**

Similarly, the migration file will contain some unfamiliar code, but you only need to pay attention to the up and down functions.

The up function contains the code for performing the migration, creating the table. down is like an "undo" - it will drop the table.

'use strict';

module.exports = {

up: async (queryInterface, Sequelize) => {

await queryInterface.createTable('Users', {

id: {

allowNull: false,

autoIncrement: true,

primaryKey: true,

type: Sequelize.INTEGER

},

firstName: {

type: Sequelize.STRING

},

lastName: {

type: Sequelize.STRING

},

email: {

type: Sequelize.STRING

},

createdAt: {

allowNull: false,

type: Sequelize.DATE

},

updatedAt: {

allowNull: false,

type: Sequelize.DATE

}

});

},

down: async (queryInterface, Sequelize) => {

await queryInterface.dropTable('Users');

}

};

Run the migration with the following command:

npx sequelize-cli db:migrate

Congratulations! You now have a User table in the database. Now it's time to add some data.

#### **Seeding the Database**

Sequelize can create a file that has "seed" data - essentially placeholder data so that there's something in the database.

npx sequelize-cli seed:generate --name user

This will create a new file in your seeders directory. Unlike the other files Sequelize has created for us, we'll need to customize it more heavily.

Open the file and pay attention to the large block comments in the up and down sections.

'use strict';

module.exports = {

up: async (queryInterface, Sequelize) => {

/\*\*

\* Add seed commands here.

\*

\* Example:

\* await queryInterface.bulkInsert('People', [{

\* name: 'John Doe',

\* isBetaMember: false

\* }], {});

\*/

},

down: async (queryInterface, Sequelize) => {

/\*\*

\* Add commands to revert seed here.

\*

\* Example:

\* await queryInterface.bulkDelete('People', null, {});

\*/

}

};

We'll add the information for a single user in the up function, and a command to delete all users in the down function:

'use strict'; module.exports = { up: async (queryInterface, Sequelize) => { await queryInterface.bulkInsert('Users', [{ firstName: 'Annie', lastName: 'Easley', email: 'ajeasley@nasa.gov', createdAt: new Date(), updatedAt: new Date() }], {}); }, down: async (queryInterface, Sequelize) => { return queryInterface.bulkDelete('Users', null, {}); } };

We've customized the Array of Objects being passed to the bulkInsert() function. In it, we've included the key-value pairs corresponding to the attributes used to generate our User model.

There are two additional keys to account for: the createdAt and updatedAt fields. These are automatically added to every model created by Sequelize.

To undo the migration, we've added a bulkDelete() call to the down section. This removes all the rows from the User table.

#### **Running the Seed File**

After customizing our seed file, let's run it:

npx sequelize-cli db:seed:all

#### **Manually Checking the Database**

Connect to your database using the psql command. From the psql prompt, we'll manually run the SQL to confirm that our seed data made it into the database.

psql database\_development

At the prompt, run a select \*, making sure to put the table name in quotes:

select \* from "Users";

You should see a one row result print out. Success!

|  |  |
| --- | --- |
|  | **Why quotes around the table name?**  Without the quotes, Postgres will disregard any capitalization and look a table named "users". This will result in an error.  When we generated the model, we capitalized User. To make sure that Postgres queries Users (with a capital U) and not users (with lower case u), put quotes around "Users". |

### **Performing CRUD Operations**

Let's assume that our example index.js already contains the basic code for an Express app. We'll add Sequelize-specific code throughout this section.

Import Sequelize and our User model using the require() function.

const Sequelize = require('sequelize');

const { User } = require('./models');

The User model generated by Sequelize has methods for all of the CRUD operations we would want:

* .create()
* .findAll()
* .findByPk()
* .update()
* .destroy()

In addition, they will accept extra arguments for adding clauses, such as where.

#### **Creating a New User**

Use .create() and pass it key-value pairs for the column names and values for the new row:

app.post('/users', async (req, res) => {

// req.body contains an Object with firstName, lastName, email

const { firstName, lastName, email } = req.body;

const newUser = await User.create({

firstName,

lastName,

email

});

// Send back the new user's ID in the response:

res.json({

id: newUser.id

});

})

#### **Retrieving Users**

Getting all rows is performed with .findAll():

app.get('/users', async (req, res) => {

const users = await User.findAll();

res.json(users);

});

.findAll() selects all columns by default. Provide an attributes option to specify column names:

app.get('/users/by-last-name', async (req, res) => {

const users = await User.findAll({

attributes: ['lastName']

});

res.json(users);

});

To retrieve a row by the id use .findByPk() (find by primary key):

app.get('/users/:id', async (req, res) => {

const oneUser = await User.findByPk(req.params.id);

res.json(oneUser);

});

Of course, you should do some basic error handling:

app.get('/users/:id', async (req, res) => {

try{

const oneUser = await User.findByPk(req.params.id);

res.json(oneUser);

} catch (e) {

console.log(e);

res.status(404).json({

message: 'User not found'

});

}

});

To do a text search, use a where option:

app.post('/users/search', async (req, res) => {

const users = await User.findAll({

where: {

firstName: req.body.term,

}

});

res.json(users);

});

To perform a search on multiple fields in raw SQL, you'd use an OR to chain multiple where clauses together. With Sequelize, you specify a special key inside the where Object:

app.post('/users/search', async (req, res) => {

const users = await User.findAll({

where: {

[Sequelize.Op.or]: [

{

firstName: req.body.term,

lastName: req.body.term

}

]

}

});

res.json(users);

});

The square brackets around Sequelize.Op.or tells JavaScript to use the value of Sequelize.Op.or as the key (as opposed to using the literal String "Sequelize.Op.or" as the key).

#### **Updating Existing Users**

.update() accepts the key-value pairs to update. It is best to use a where option so that you don't update all rows:

app.post('/users/:id', async (req, res) => {

const { id } = req.params;

// Assuming that `req.body` is limited to

// the keys firstName, lastName, and email

const updatedUser = await User.update(req.body, {

where: {

id

}

});

res.json(updatedUser);

});

#### **Deleting a User**

The destroy() method will delete rows:

app.delete('/users/:id', async (req, res) => {

const { id } = req.params;

const deletedUser = await User.destroy({

where: {

id

}

});

res.json(deletedUser);

});

### **Foreign Keys and Associations**

#### **Creating a Related Model**

We won't worry about handling file uploads in our app. Instead, we'll assume that the user is putting the URL of a photo that already exists.

npx sequelize-cli model:generate --name Photo --attributes title:string,url:string,userId:integer

To associate Users and Photos, we add a userId field to our Photo model. This will serve as our foreign key. To let Sequelize know that, we'll modify models/photos.js before we run the migration.

#### **Setting Up the Foreign Key**

Update the body of the associate() method to include a .belongsTo() call:

'use strict';

const {

Model

} = require('sequelize');

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

class Photo extends Model {

/\*\*

\* Helper method for defining associations.

\* This method is not a part of Sequelize lifecycle.

\* The `models/index` file will call this method automatically.

\*/

static associate(models) {

Photo.belongsTo(models.User, {

foreignKey: 'userId',

onDelete: 'CASCADE'

});

}

};

Photo.init({

title: DataTypes.STRING,

url: DataTypes.STRING,

userId: DataTypes.INTEGER

}, {

sequelize,

modelName: 'Photo',

});

return Photo;

};

It's best for us to update the User model:

'use strict';

const {

Model

} = require('sequelize');

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

class User extends Model {

/\*\*

\* Helper method for defining associations.

\* This method is not a part of Sequelize lifecycle.

\* The `models/index` file will call this method automatically.

\*/

static associate(models) {

// define association here

User.hasMany(models.Photo, {

foreignKey: 'userId'

});

}

};

User.init({

firstName: DataTypes.STRING,

lastName: DataTypes.STRING,

email: DataTypes.STRING

}, {

sequelize,

modelName: 'User',

});

return User;

};

#### **Modifying the Migration**

There's one more step to make sure that the foreign key from the Photos table to the Users table is created correctly.

Edit the migration file for creating the Photos table, adding additional information to the userId Object:

'use strict';

module.exports = {

up: async (queryInterface, Sequelize) => {

await queryInterface.createTable('Photos', {

id: {

allowNull: false,

autoIncrement: true,

primaryKey: true,

type: Sequelize.INTEGER

},

title: {

type: Sequelize.STRING

},

url: {

type: Sequelize.STRING

},

userId: {

type: Sequelize.INTEGER,

onDelete: 'CASCADE',

references: {

model: 'Users',

key: 'id',

as: 'userId'

}

},

createdAt: {

allowNull: false,

type: Sequelize.DATE

},

updatedAt: {

allowNull: false,

type: Sequelize.DATE

}

});

},

down: async (queryInterface, Sequelize) => {

await queryInterface.dropTable('Photos');

}

};

Then, run the migration:

npx sequelize-cli db:migrate

#### **Generating Seed Data**

npx sequelize-cli seed:generate --name photo

Edit the seed file and enter some data to insert into the photos table.

'use strict';

module.exports = {

up: async (queryInterface, Sequelize) => {

await queryInterface.bulkInsert('Photos', [{

title: Something I helped put in space',

url: 'https://solarsystem.nasa.gov/system/resources/detail\_files/17761\_cassinihuygens\_BTN\_16\_purple\_final\_01.jpg',

userId: 1,

createdAt: new Date(),

updatedAt: new Date()

}], {});

},

down: async (queryInterface, Sequelize) => {

await queryInterface.bulkDelete('Photos', null, {});

}

};

Run the seed file:

npx sequelize-cli db:seed:all

#### **Getting Related Photos**

Import the Photo model as well as the User:

const { User, Photo } = require('./models);

To get all Users with their Photos:

app.get('/users/photos', async (req, res) => {

const users = await User.findAll({

include: [{

model: Photo

}]

});

res.json(users);

});

You can combine the include with other options. Here is a modified version of our /users/search handler that includes a User's Photos:

app.post('/users/search', async (req, res) => {

const users = await User.findAll({

where: {

[Sequelize.Op.or]: [

{

firstName: req.body.term,

lastName: req.body.term

}

]

},

include: [{

model: Photo

}]

});

res.json(users);

});

You can use the same syntax to get all Photos, and include the User Object with each Photo.

app.get('/photos/users, async (req, res) => {

const photos = await Photos.findAll({

include: [{

model: User

}]

});

res.json(photos);

});

### **Summary**

In this lesson, you learned how to use Sequelize. You used the CLI tool to create a database, Models, and migrations. Using the npm module, you accessed your PostgreSQL database via the Model Objects generated by Sequelize.

You performed the typical CRUD Operations (Create, Retrieve, Update, and Delete) using the Model methods instead of writing SQL queries by hand. In addition, you created a relationship between Models, using methods instead of writing JOINs or subqueries.

It's important to be familiar with at least one ORM. They are used in real-world applications because they speed up development by replacing repetitive code with function calls and they help protect your application from certain kinds of security vulnerabilities (like SQL injection).

----set up---- bartel

setting up an express app with ejs template engine, git (for gitignore) and bcrypt for authorization/hashing

1)express --ejs --git (or your app name)

This will generate all the files [????npx express generator????]

2)npm install

This command installs all the dependencies that were generated in the previous command.

3)npm start

This will run the app- at this point text a sample route.

4)add nodemon to package.json in the “scripts” section.

5)npm install bcrypt

This installs the package needed for hashing.

Documents needed:

express-generator-- https://expressjs.com/en/starter/generator.html

sequelize (queries)-- https://sequelize.org/master/manual/model-querying-basics.html

sequelize-cli-- https://www.npmjs.com/package/sequelize-cli

bcrypt-- https://www.npmjs.com/package/bcrypt

NOTE: When running commands with npx vs npm means we are running selectively, from the cloud, instead of installing the entire package/module locally.

----Authentication Basics----

-Passwords get entered by users as plain text.

-Encrypting plain text user-input can be encrypted/decrypted with a ‘key’. If a security breach finds your key then content can be easily decrypted. This is only good got user content- not user passwords.

-Hashed plain text means taking a user-input and converting it to something else completely different, know as a hash. The hash is generated used a “salting” method which gives it a highly unique string of char/ints. SaltRounds is a number the dictates the level of intricacy of the salting process. High level saltRounds come at a “cost”. See example:

GHz core you can roughly expect:

rounds=8 : ~40 hashes/sec

rounds=9 : ~20 hashes/sec

rounds=10: ~10 hashes/sec

rounds=11: ~5 hashes/sec

rounds=12: 2-3 hashes/sec

rounds=13: ~1 sec/hash

rounds=14: ~1.5 sec/hash

rounds=15: ~3 sec/hash

rounds=25: ~1 hour/hash

rounds=31: 2-3 days/hash

-----create instance/DB-----

1)Install packages/modules/etc

npm install sequelize pg

npx sequelize-cli init

2)Configure the config file for your localhost name, etc that was generated from step 1

3)Create a database db

npx sequelize-cli db:create

4)Create model using custom name and attributes:

npx sequelize-cli model:generate --name User --attributes firstName:string,lastName:string,email:string

(this creates a migration file that will run into the model after you have customized this, run the next command)

5)Run migration (from step 3) into model

npx sequelize-cli db:migrate

6)Create a seed file with a custom name:

npx sequelize-cli seed:generate --name user

7)Fill/customize the seed file and then run:

npx sequelize-cli db:seed:all

Note: check file structure after each command, navigate to file and customize. Also use postico ‘refresh’ to ensure step migrations and seeds are populating to db.

---creating express routes----

Express routes will be created using sequelize

expressjs.comexpressjs.com

Express application generator (27 kB)

https://expressjs.com/en/starter/generator.html

sequelize.orgsequelize.org

Sequelize

An easy-to-use multi SQL dialect ORM for Node.js

npmnpm

sequelize-cli

The Sequelize CLI. Latest version: 6.4.1, last published: a month ago. Start using sequelize-cli in your project by running npm i sequelize-cli. There are 402 other projects in the npm registry using sequelize-cli.

npmnpm

bcrypt

A bcrypt library for NodeJS.. Latest version: 5.0.1, last published: a year ago. Start using bcrypt in your project by running npm i bcrypt. There are 3197 other projects in the npm registry using bcrypt.