# Configuración de Postgres en Docker

## INSTALAR DOCKER

## CREAR ARCHIVO PARA LEVANTAR EL DOCKER CON POSTGRES



(a nivel raíz del proyecto)

## Correr el Docker por comando en el proyecto:

docker-compose up -d postgres

## Verificar que esté corriendo:

Docker-compose ps







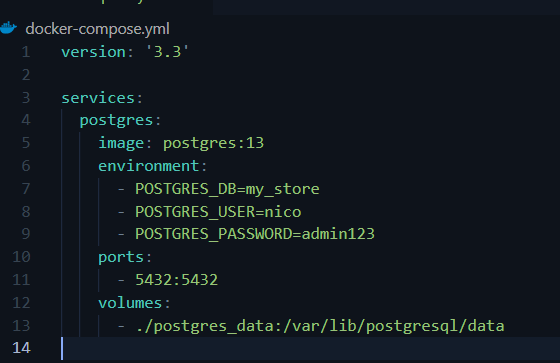
## Apagar servicios docker:

docker-compose down

## Agregar persistencia al contenedo:

Con esto si se guardará la data que vayamos registrando

En el archivo de config del docker-compose se le agrega la opción de **“volumes”**



# Explorando Postgres: interfaces gráficas vs. Terminal

## Correr la terminal:

docker-compose exec postgres bash

## Conectarme a la base de datos desde la terminal:

psql -h localhost -d my\_store -U nico

## Salirnos de la base de datos:

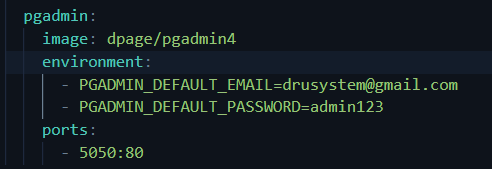
\q

## Salir de la terminal:

Exit

## Conectarnos por motor gráfico

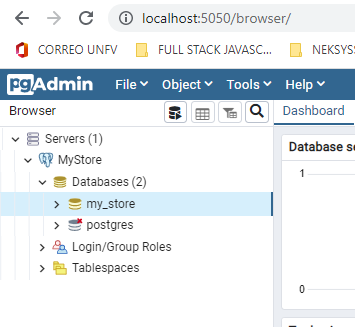
Agregamos el servicio al archivo “docker-compose.yml”



Correr el servicio:

docker-compose up -d pgadmin

Accedemos al pgadmin desde el navegador:



Creamos la conexión al servidor en Object -> new conection

Ojo en ves del IP podemos poner el nombre que se le dio a contenedor de postgres creado en docker.

# Integración de node-postgres

Usaremos la librería “node-postgres.com”

Comando: npm install pg

Crear archivo de conexión en libs/postgres.js

const { Client } = require('pg')

async function getConnection(){

  const client = new Client({

      host:'localhost',

      port:5432,

      user:'nico',

      password:'admin123',

      database:'my\_store'

  });

  await client.connect();

  return client;

}

module.exports = getConnection;

usarlo en el servicio:

const getConnetion = require('../libs/postgres')

async find() {

    const client = await getConnetion();

    const response = await client.query('SELECT \* FROM tasks');

    return response.rows;

  }

# Manejando un Pool de conexiones

const { Pool } = require('pg');

const {config } = require('./../config/config')

const USER = encodeURIComponent(config.dbUser);

const PASSWORD = encodeURIComponent(config.dbPassword);

const URI = `postgres://${USER}:${PASSWORD}@${config.dbHost}:${config.dbHost}/${config.dbName}`

const pool = new Pool({ connectionString:URI });

module.exports = pool;

En el servicio:

const pool = require('../libs/postgres.pool')

constructor() {

*this*.pool = pool;

*this*.pool.on('error',(err)=>console.error(err));

  }

async find() {

    const query = 'SELECT \* FROM tasks';

    const response = await *this*.pool.query(query);

    return response.rows;

  }

# Variables de ambiente en Node.js

Instalar la librería:

Comando: npm i dotenv

Config/config.js

require('dotenv').config();

const config = {

  env: process.env.NODE\_ENV || 'dev',

  port: process.env.PORT || 3000,

  dbUser: process.env.DB\_USER,

  dbPassword: process.env.DB\_PASSWORD,

  dbHost: process.env.DB\_HOST,

  dbName: process.env.DB\_NAME,

  dbPort: process.env.DB\_PORT,

}

module.exports = {

  config,

}

# ¿Qué es un ORM? Instalación y configuración de Sequelize ORM

Usaremos sequelize:

<https://sequelize.org/docs/v6/>

npm install --save sequelize

Conexión en libs/sequelize.js

const { Sequelize } = require('sequelize');

const {config } = require('./../config/config')

const USER = encodeURIComponent(config.dbUser);

const PASSWORD = encodeURIComponent(config.dbPassword);

const URI = `postgres://${USER}:${PASSWORD}@${config.dbHost}:${config.dbPort}/${config.dbName}`

const sequelize = new Sequelize(URI,{

  dialect:'postgres',

  logging:true,

});

module.exports = sequelize;

Consumiendo sequelize:

const sequelize = require('../libs/sequelize')

 async find() {

    const query = 'SELECT \* FROM tasks';

    const [data] = await sequelize.query(query);

    return data;

  }

# Tu primer modelo en Sequelize

Se crea la carpeta db/models:

Se crea el archivo user.model.js:

const { Model,DataTypes,Sequelize } = require('sequelize');

const USER\_TABLE  = 'users';

const UserSchema = {

  id:{

    allowNull:false,

    autoIncrement:true,

    primaryKey:true,

    type:DataTypes.INTEGER

  },

  email:{

    allowNull:false,

    type:DataTypes.STRING,

    unique:true

  },

  password:{

    allowNull:false,

    type:DataTypes.STRING

  },

  createdAt:{

    allowNull:false,

    type:DataTypes.DATE,

    field:'create\_at',

    defaultValue:Sequelize.NOW

  }

}

class User extends Model{

  static associate(){

  }

  static config(sequelize){

    return {

      sequelize,

      tableName:USER\_TABLE,

      modelName:'User',

      timestamps:false

    }

  }

}

module.exports = {

  USER\_TABLE,

  UserSchema,

  User

}

Se crea el archivo index.js (donde van ir todos los modelos)

const { UserSchema,User } = require('./user.model');

function setupModels(sequelize){

  User.init(UserSchema,User.config(sequelize));

}

module.exports = setupModels;

Se enlaza el index.js en lib/sequelize.js

const { Sequelize } = require('sequelize');

const {config } = require('./../config/config');

const setupModels = require('./../db/models');

const USER = encodeURIComponent(config.dbUser);

const PASSWORD = encodeURIComponent(config.dbPassword);

const URI = `postgres://${USER}:${PASSWORD}@${config.dbHost}:${config.dbPort}/${config.dbName}`

const sequelize = new Sequelize(URI,{

  dialect:'postgres',

  logging:true,

});

setupModels(sequelize);

sequelize.sync();

module.exports = sequelize;

# Crear, actualizar y eliminar

async findOne(id) {

    const user = await models.User.findByPk(id);

    if(!user){

      throw boom.notFound('usuario no encontrado');

    }

    return user;

  }

async update(id, changes) {

    const user = await *this*.findOne(id);

    const response = await user.update(changes);

    return response;

  }

  async delete(id) {

    const user = await *this*.findOne(id);

    await user.destroy();

    return {id};

  }

async create(data) {

    const response = await models.User.create(data);

    return response;

  }

# CAMBIAR A MYSQL

Configuración del contenedor:

 mysql:

    image: mysql:5

    environment:

      - MYSQL\_DATABASE=my\_store

      - MYSQL\_USER=nico

      - MYSQL\_ROOT\_PASSWORD=admin123

      - MYSQL\_PORT=3306

    ports:

      - 3306:3306

    volumes:

      - ./mysql\_data:/var/lib/mysql

  phpmyadmin:

    image: phpmyadmin/phpmyadmin

    environment:

      - MYSQL\_ROOT\_PASSWORD=admin123

      - PMA\_HOST=mysql

    ports:

      - 8080:80

Instalando docker contenedor de mysql:

docker-compose up -d mysql

Instalar contenedor de phpMyAdmin:

docker-compose up -d phpmyadmin

Instalar driver para node-mysql:

npm install mysql2

Cambiar el puerto en .env

DB\_PORT='3306'

*# PUERTO POSTGRES'5432'*

*# PUERTO MYSQL '3306'*

Cambiar en el archivo lib/sequelize.js

const { Sequelize } = require('sequelize');

const {config } = require('./../config/config');

const setupModels = require('./../db/models');

const USER = encodeURIComponent(config.dbUser);

const PASSWORD = encodeURIComponent(config.dbPassword);

const URI = `mysql://${USER}:${PASSWORD}@${config.dbHost}:${config.dbPort}/${config.dbName}`

const sequelize = new Sequelize(URI,{

  dialect:'mysql',

  logging:true,

});

setupModels(sequelize);

sequelize.sync();

module.exports = sequelize;

# 13. ¿Qué son las migraciones? Migraciones en Sequelize ORM

Para trabajar con migraciones necesitamos instalar:

npm i sequelize-cli -D

Creamos el archivo sequelize de migraciones en la raíz:

Archivo: “.sequelizerc”

module.exports = {

  'config':'./db/config.js',

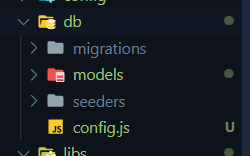
  'models-path':'./db/models/',

  'migrations-path':'./db/migrations/',

  'seeders-path':'./db/seeders/',

}

Luego creamos las carpetas que estamos indicando y el archivo config.js



En config.js

const {config} = require('./../config/config')

const USER = encodeURIComponent(config.dbUser);

const PASSWORD = encodeURIComponent(config.dbPassword);

const URI = `mysql://${USER}:${PASSWORD}@${config.dbHost}:${config.dbPort}/${config.dbName}`

module.exports ={

  development:{

    url:URI,

    dialect:'mysql',

  },

  production:{

    url:URI,

    dialect:'mysql',

  }

}

CONFIGURANDO EL PACKAGE.JSON PARA PODER CREAR MIGRACIONES:

"scripts": {

    "dev": "nodemon index.js",

    "start": "node index.js",

    "lint": "eslint",

    "migrations:generate":"sequelize-cli migration:generate --name"

  },

Quitamos el sync de sequelize:

Ejecutando comando:

npm run migrations:generate create-user

Codificando migración para unirla a un esquema:

'use strict';

const { UserSchema,USER\_TABLE } = require('./../models/user.model')

*/\*\* @type {import('sequelize-cli').Migration} \*/*

module.exports = {

  async up (queryInterface) {

    await queryInterface.createTable(USER\_TABLE,UserSchema);

  },

  async down (queryInterface) {

    await queryInterface.drop(USER\_TABLE);

  }

};

Correr las migraciones:

"migrations:run":"sequelize-cli db:migrate",

(Commando puesto en los script de package.json)

Agregando campo faltante a tabla creada en migración:

'use strict';

const { UserSchema,USER\_TABLE } = require('./../models/user.model')

*/\*\* @type {import('sequelize-cli').Migration} \*/*

module.exports = {

  async up (queryInterface) {

    await queryInterface.addColumn(USER\_TABLE,'role',UserSchema.role);

  },

  async down (queryInterface) {

    await queryInterface.removeColumn(USER\_TABLE,'role',UserSchema.role);

  }

};

# 16. Relaciones uno a uno

Un usuario puede ser un único cliente

*this*.hasOne(models.Customer,{

      as:'customer',

      foreignKey:'userId'

    })

Un cliente le pertenece a un único usuario:

*this*.belongsTo(models.User,{as:'user'});

# 18. Relaciones uno a muchos

Crear migración:

npm run migrations:generate products

Correr migración:

npm run migrations:run

Un cliente puede tener muchas ordenes:

*this*.hasMany(models.Order,{as:'orders',

      foreignKey:'customerId'

    })

Una orden le pertenece a un solo cliente:

*this*.belongsTo(models.Customer,{as:'customers'});