

backend con node y express

guia con ejemplos para construir una API que consuma desde una api pública de nuestra eleccion (api de rick y morty, api de noticias, api de comidas, etc) e interactúe con una base de datos de postgresQL mediante Sequelize.

para iniciar:

en la carpeta **raíz** de nuestro proyecto backend:

npm i :

```
express
nodemon,morgan,axios
pg, pgh-store, sequelize(para postgres)
dotenv (en versiones mas recientes no es necesario instalarla, ya que viene con node)
```

cabe destacar que no instalamos node explicitamente porque npm es el manejador de paquetes de node, por ende ya instala node

en la carpeta raíz crearemos archivos como:

index.js

que se encarga de poner a correr el servidor que crearemos

en la misma carpeta, cuando usemos el comando npm install , obtendremos un

package.json

que contiene un objeto json con las dependencias instaladas y sus versiones

asi como los scripts y sus llamadas. para usar nodemon y sus beneficios, definiremos e script *start* como *nodemon index.js* (siendo lo normal *node index.js*)

```
"scripts":
{
  "start": "nodemon index.js"
}
```

dentro de nuestra carpeta raíz, crearemos la carpeta src,
dentro de la cual crearemos **app.js** que se encarga de
requerir a express y definir el servidor, el cual
va a ser ejecutado por index.js.

app.js

```
const express = require("express");

const app = express();

module.exports = app;
```

de este modo guardamos *la ejecución de express* en una variable que llamaremos app (puede ser server, api, express, etc...), para luego exportar la variable app y que la importe index.js del siguiente modo:

index.js

```
const app = require(".src/app");
const PORT = "3001";
app.listen(PORT, () => console.log(`server listening at port ${PORT}`));
```

en index.js guardamos la variable app (que acabamos de importar del archivo app.js)
en la variable app (de nuestro archivo index.js)

(los nombres no son obligatorios pero conviene ser descriptivos)
luego, con metodo .listen le damos como primer parametro el numero de puerto en el cual debe escuchar el servidor (PORT), y como segundo parametro una funcion callback que usaremos para ejecutar un console.log con el cual veremos en consola cuando el servidor esté corriendo

volviendo a app.js

Middlewares

un middleware es un punto medio en el camino de una request que el front le hace a nuestra api antes de llegar al endpoint.
y sirve para asegurar que la request que llega al endpoint cumpla con ciertas validaciones de formato o contenido.

para aplicarlos se usa la siguiente estructura:

```
app.use(middlewareEjemplo(req, res, next));
```

Los dos primeros parámetros se usan para ejecutar la logica interna del middleware, y si surge algún error, efectúa un return enviando un error descriptivo en la *res* pero si completa su tarea, ejecuta el *next* para proseguir con la siguiente instrucción.

un middleware personalizado puede verse de la siguiente manera:

```
app.use(req, res, next);
{
  let message =
    "la request del cliente ha pasado a traves del custom middleware";
  console.log(message);
  next();
}
```

los middlewares personalizados normalmente se guardan como funciones en una carpeta llamada middlewares y se importan para ser usados

Cuando nuestro cliente (el frontend) nos envía datos que debemos manipular, o usar para efectuar acciones con un servicio externo, nos envía esta información en el Body de la request, lo recibiremos en formato JSON, pero necesitamos convertirlo a formato *objeto de javascript* para manipularlo de forma comoda y eficiente. Para este fin, usaremos este middleware:

```
const express = require("express");
app.use(express.json());
```

y un poco mas antiguo pero aun vigente, esta este otro que sirve para lo mismo.

```
const bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: true, limit: "50mb" }));
app.use(bodyParser.json({ limit: "50mb" }));
```

Un middleware que usamos comunmente es **morgan**, el cual ya viene diseñado y lo descargamos desde npm y sirve para darnos informacion sobre los procesos que ocurren en nuestra api por medio de la consola.

morgan permite darle formato a su output dependiendo de como sea llamado.

para usarlo, primero lo requerimos y lo guardamos en una variable (comunmente la llamamos 'morgan'), para luego ejecutarla como middleware con el formato

```
const morgan = require("morgan");
app.use(morgan("dev"));
```

El parametro que recibe, resume lo que en un middleware personalizado sería req,res,next.

dicho parametro puede ser una de las siguientes alternativas:

- **tiny:** proporciona la salida minima de datos del servidor y el status lo da con color (verde para exito, rojo para error del servidor, amarillo para error del cliente y sin color para info).
- **short:** tiene las mismas características de tiny, pero agrega informacin sobre el tiempo de respuesta del server y longitud del contenido.
- **dev:** es aun mas detallado que short, maniene los colores, es el que solemos usar como desarrolladores por la cantidad de datos que da.
- **common:** da informacion mas extensa que dev, fecha y hora de la request, la url solicitada y el metodo http usado.
- **combined:** da todos los datos que los formatos anteriores y suma informacion del cliente como la direccion IP desde la que se hace la request.
- ó puede personalizarse segun explica su [documentacion](#)

Otros ejemplos útiles de middlewares son aquellos que controlan las peticiones desde o hacia otros sitios web (CORS) y otras funciones similares. algunos de estos middlewares útiles son:

```
const cookieParser = require("cookie-parser");
const cors = require("cors");

app.use(cors());
app.use(cookieParser());
// para obtener en un objeto de javascript, las cookies del navegador del
cliente. el objeto se encuentra en req.cookies
app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "*");
  // el asterisco indica que este backend va a aceptar peticiones de
  cualquier cliente, se puede especificar UN cliente o dejarlo asi
  res.header("Access-Control-Allow-Credentials", "true");
  res.header(
    "Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept"
  );
  res.header(
    "Access-Control-Allow-Methods",
    "GET, POST, OPTIONS, PUT, DELETE"
  );
  // aqui estamos dando permiso a ciertas peticiones HTTP
  next();
});
```

los middlewares prediseñados como morgan o cors, simplemente se importan y se usan ya que no necesitamos definir la funcion

Endpoints

luego de los middlewares, que pueden ser varios, llegan los endpoints o rutas.
cada ruta recibe un metodo http especifico como get, put, post, o delete.

y si la request enviada por el cliente coincide con lo que espera recibir el endpoint, se ejecuta una lógica. (similar al funcionamiento del statement SWITCH) por ejemplo:

```
const handler = (req, res) => {  
  res.status(200).send("entrando a la ruta GET en / ");  
};  
  
//endpoint:  
app.get("/", handler);
```

en este ejemplo, las requests o peticiones de tipo **GET** a la ruta "/" logran ser aceptadas por nuestro endpoint, asi que se dispara su funcion **handler**.

Este handler ejemplo, da como resultado un status de éxito 200, y el contenido de la respuesta que en este caso es tan simple como la frase *entrando a la ruta GET en /*

recordar que los [codigos de status http](#) tienen un significado:

Éxito(200-299)

Informativo(300-399)

Error del cliente(400-499)

Error del server (500-599)

lo endpoints pueden ser estáticos o dinámicos.

un endpoint estático es por ejemplo "/home" ó "/ropa/pantalones"

los endpoints dinámicos son por ejemplo "/users/:userID" ó "/ropa/pantalones/:talla"

en los endpoints dinámicos puede cambiar el valor dinámico siempre que cumpla cierto formato, en user ID podria ser un numero de documento, o en talla una letra como S, M o L.

```
app.get("/users/:ID", (req, res) => {  
  res.status(200).send("texto con los detalles del usuario solicitado");  
});
```

el cliente tendría acceso a este endpoint si a demás de hacer una peticion de tipo get, la hace a "/users/6" ó "/users/15", por ejemplo.

hasta este punto nuestro archivo contiene un custom middleware, el middleware Morgan, cors, body parser, rutas estáticas y rutas dinámicas, y se ve algo así:

app.js

```
const express = require("express");
const app = express();
const morgan = require("morgan");

/* middlewares*/
app.use(express.json());
app.use(req, res, next){
  let message = "la request del cliente ha pasado a traves del custom
middleware";
  console.log(message);
  next();
};
app.use(morgan("dev"));
/*más middlewares*/

/*endpoints*/
app.get("/", (req, res) => {
  res.status(200).send("ok");
});
app.get("/users/:ID", (req, res) => {
  res.status(200).send("texto con los detalles del usuario solicitado");
});
/*más endpoints*/

module.exports = app;
```

pero ¿que pasa cuando aumente la cantidad de rutas?, recordemos que debemos tener rutas para cada funcionalidad de nuestra API, y sus handlers serán mucho mas complejos que devolver una palabra o una oración.

Para evitar tener un archivo enorme lleno de rutas y cada ruta con sus handlers internos, *debemos a modularizar*, recordando que lo ideal es que cada archivo debe tener una *única responsabilidad*.

para ello dentro de nuestra carpeta SRC crearemos la carpeta *routes*, la cual va a alojar nuestros endpoints.

la carpeta routes debe contener un archivo index.js que recibirá todas las requests y determinará a que endpoint debe dirigirse, y a demás del archivo index.js contendrá las crpetas de cada servicio y de cada endpoint.

importante entender:

index.js **es un router** por ende, debe:

- importar express
 - definir el router
 - exportar el router
- para que el modulo *del nivel superior* (**en este caso app.js**) pueda importar dicho router y mediante un middleware pueda usarlo.

Para que esta maniobra logre el resultado que esperamos, es necesario agregar **un nuevo middleware** a app.js, el cual **será nuestro Router** el cual debe ir de ultimo entre los middlewares, de modo que la request de nuestro cliente, solamente se vaya a las rutas una vez que haya pasado por los middlewares que le darán el formato correcto a esta request.

volviendo a app.js:

app.js

```
const express = require("express");
const cookieParser = require("cookie-parser");
const cors = require("cors");
const mainRouter = require("./routes");//importando el router de index.js (de
rutas)
const app = express();
const morgan = require("morgan");

/* middlewares*/
app.use(req, res, next){
  console.log(message);
  next();
};
app.use(morgan("dev"));
app.use(cors());
app.use(cookieParser());
app.use(mainRouter);//<--- agregamos nuestro router como un middleware.

/*eliminamos todos los endpoints,
y los trasladamos a la carpeta routes*/

module.exports = app;
```

al requerir `./routes` javascript entra a esa carpeta y aunque solo hay un monton de archivos ahí, se da cuenta de que hay un index y toma lo que ese archivo le dá.

Es buena idea dividir las rutas de un servicio y las rutas de otro servicio (por ejemplo users y posts) en dos routers distintos para dividir la responsabilidad y acortar los archivos, de modo que usaremos dentro de nuestro main router (index.js) dos middlewares que redireccionarán a usersRouter ó a postsRouter respectivamente.

Para esto debemos efectuar la misma maniobra que antes:

en postsRouter.js y en usersRouter.js debemos:

- importar express
- definir el router
- y exportarlo, para que el modulo *del nivel superior* (**en este caso index.js**) lo importe y mediante *dos* middlewares pueda usarlos.

Pasando en limpio, el flujo de nuestro proyecto hasta este momento se ve así:

1. nuestro archivo index.js (de la carpeta raíz) "enciende" el servidor que se encuentra en app.js
2. la request es enviada por el usuario y recibida por nuestro archivo app.js
3. la request entra y sale de cada uno los middlewares hasta que llega a nuestro middleware "mainRouter".
4. la request llega a nuestro archivo index.js (de la carpeta routes) y desde ahí es redireccionada a el router del servicio correspondiente.
5. el router del servicio puede aplicar middlewares para validar o verificar el tipo de contenido, el formato o lo que sea necesario antes de enviar la request a la ruta que le corresponde.
6. al entrar en la ruta correspondiente, la request ejecuta el handler de ese endpoint y se obtiene una respuesta del servidor, la cual va de regreso por todo este camino hasta el cliente.

en seguida veremos lo concerniente a las posibles respuestas, y como desarrollar el contenido de todo lo que vive dentro de la carpeta routes.

un middleware en una ruta se puede aplicar del siguiente modo:

```
usersRouter.post("/", validate, createUserHandler);
```

siendo validate nuestro middleware, que podría verse algo así:

```
const validate = (req, res, next) => {  
  const { name, email, phone } = req.body;  
  if (!name) return res.status(400).json({ error: "missing name" });  
  if (!email) return res.status(400).json({ error: "missing email" });  
  if (!phone) return res.status(400).json({ error: "missing phone" });  
  next();  
};
```


con la modularización de las rutas y sus handlers, la estructura de las carpetas y archivos del proyecto, hasta este momento se debe ver algo así:

```
/
/index.js
/package.json

/src
/src/app.js

  /src/middlewares
  /src/middlewares/middlewares.js

  /src/handlers
  /src/handlers/usersHandlers.js
  /src/handlers/postsHandlers.js
  /src/handlers/service3Handlers.js

  /src/routes
  /src/routes/index.js
  /src/routes/usersRouter.js
  /src/routes/postsRouter.js
  /src/routes/otherRouter1.js
```

Handlers

los handlers son las funciones que reciben los datos que vienen en la request del cliente, y en consecuencia interactúan con una API externa o con nuestra base de datos.

a una API externa normalmente, solo se le pueden *pedir* datos, y no guardar datos en ella.

por ejemplo de la API de Rick y Morty solo podremos pedir personajes, pedir mundos o pedir episodios, no podemos crear un personaje nuevo, eliminar un mundo o modificar un episodio. Esto se debe a que es pública y otras personas están consumiendo esa misma información en sus proyectos, y si la modificamos, eliminamos o creamos a nuestro antojo, podríamos dañar el trabajo de otro desarrollador.

los handlers tienen dos partes principales que llamaremos

1. el handler
2. el controller

y es necesario dividirlos para ser consistente con el *principio de responsabilidad única*, de modo que

- el controller es la pieza de código que se encarga de interactuar con la base de datos o API
- el handler se encarga de efectuar un **try** que invoca al controller y devuelve el resultado exitoso, y el **catch** que captura los posibles errores que presente el controller.

- es buena práctica evitar que el handler interactúe con el modelo, lo debe hacer el controller.

- es buena practica evitar que el controller reciba propiedades de la request directamente, la debe recibir el handler y aislar las propiedades que el controller va a usar.

para dividirlos nuevamente conviene modularizar nuestro codigo, teniendo una carpeta de handlers y una de controllers, cada controller debe exportar su funcion, para que el handler pueda requerirla y usarla. (podemos tener un archivo con todos los handlers o un archivo para cada uno, de igual forma con los controllers) algunos ejemplos de funcionalidades mas especificas de los handlers-controllers pueden ser:

- solicitar TODOS los usuarios.
- solicitar UN usuario específico.
- solicitar los usuarios que tengan cierta CARACTERÍSTICA.
- CREAR un nuevo usuario.
- ELIMINAR un usuario existente.
- MODIFICAR datos de un usuario existente

de estas cuatro acciones, deriva el término **CRUD**, que son las siglas para **Create, Read, Update y Delete**

la capacidad del desarrollador de efectuar cada tipo de peticion depende de varios factores, por ejemplo, si la api externa a la que vamos a solicitar informacion, tiene la posibilidad de darnos usuarios filtrados por genero, edad, o ubicación por ejemplo, podemos crear una ruta para pedir usuarios con características específicas, pero si no tiene esa opcion, debemos pedir todos los usuarios e idear una lógica que se encargue de filtrar.

por otro lado, si queremos una ruta para crear, eliminar, o modificar usuarios, vamos a necesitar un servicio distinto a una API pública externa, un servicio del cual nosotros como desarrolladores tengamos completo acceso y control, **una base de datos**.

Base de datos y ORM

Estaremos utilizando una base de datos de tipo relacional, a la cual se accede, manipula, consulta y modifica haciendo consultas en lenguaje SQL. en específico usaremos PostgreSQL.

en conjunto con la base de datos, usaremos un ORM que hace las veces de "interprete bilingüe" entre el lenguaje SQL y Javascript, de modo que no tendremos la necesidad de escribir codigo en SQL para interactuar entre nuestro servidor y la base de datos.

Sólo escribiremos código SQL en algunas ocasiones específicas:

- luego de **instalar postgresql**, y
- **conectarnos** con el comando `psql` ó `psql -U postgres` para indicar el usuario postgres
- con el comando `\password postgres` podrás **establecer una contraseña** para el usuario postgres
- podremos usar el comando `\l` para visualizar las bases de datos existentes.
por lo general existirán postgres, template0 y template1, que vienen por defecto.
- procedemos a **crear nuestra base de datos**, la cual llamaremos mydb para este ejemplo, `CREATE DATABASE mydb;` respetando la sintaxis de mayusculas en las instrucciones, y el ; (semicolon) para finalizar la orden.

- ahora debemos **conectarnos** a la base de datos que acabamos de crear, para esto usaremos `\c mydb` y con ese mismo comando, cambiar entre una base de datos y otra.
- podremos salir de postgres a bash usando `\q` y ahí podemos establecer los datos que el comando `psql` usará si no se especifican, conviene usar postgres como username y postgres como base de datos, de este modo con `psql`, entraremos como el usuario principal a una base de datos que siempre existirá.
- Podemos **visualizar el contenido** de nuestra db, para esto usaremos `\dt`, seguramente no tendrá nada, y crearemos tablas nuevas desde nuestra app.
- cuando ya tengamos nuestras tablas creadas, podemos usar los comandos de `SELECT` para navegar desde la consola en nuestra db, por ejemplo `SELECT * FROM "Users";` (se usan comillas dobles porque Users tiene una mayúscula).
- con el comando `DROP DATABASE mydb;` podemos **eliminar** por completo la base de datos especificada, y podremos crearla nuevamente desde cero si es lo que necesitamos en el momento.

el ORM que usaremos es Sequelize, que mediante instrucciones en javascript manejará nuestra base de datos desde un archivo que llamaremos `db.js` ubicado en nuestra carpeta `src`.

para manipular la base de datos, Sequelize necesita tener acceso a nuestro usuario de postgres, contraseña, al puerto en el que esta corriendo la base de datos y otros datos sensibles, esto supone un riesgo ya que si subimos a github nuestro codigo con estos datos, cualquier persona tendria facil acceso a nuestra base de datos. aqui entran en juego las variables de entorno que se encargan de mantener estos datos a salvo.

variables de entorno

las variables de entorno o environment variables, las guardaremos en un archivo con nombre `".env"` en nuestra carpeta raíz. dentro de este archivo, tendremos una lista que no esta escrita en javascript, y que se ve similar a esto:

```
DB_USER=miusuario
DB_PASSWORD=miclave
DB_HOST=localhost
DB_PORT=5432
DB_NAME=midatabase API_KEY=1c2a137230ce510497693405f5f9f015
PORT=3001
```

se pueden agregar mas variables de entorno segun las necesidades del proyecto, pero esencialmente usaremos estas.

es buena idea separar `dbhost` y `dbport` ya que al desplegar el backend serán solicitadas separadas.

es recomendable que a la misma altura del archivo `.env`, tengamos un archivo `.gitignore` en el cual tendremos una linea `.env` para asegurarnos de que el archivo no se subirá a github

Modelos

son archivos .js que definen como se creará cada tabla de nuestra base de datos, los ubicaremos en `"/src/models/"`.

Es recomendable que al crear un modelo (por ejemplo el de la tabla de usuarios), nombrarlo `User.js` ó `Usuario.js` en *singular* debido a que sequelize crea la tabla y las referencias en plural, y las referencias a cada elemento de la tabla, será tal cual el nombre de la tabla:

- nombre del modelo: `User`
- nombre de la tabla creada por sequelize en base al modelo: `Users`
- nombre de cada elemento de la tabla: `User`

ahora, aprendamos a construir un modelo y conectarnos a la base de datos mediante sequelize y nuestro archivo `db.js`

conexión del servidor a la DB

el módulo **db.js** es el encargado de establecer comunicacion entre nuestra app de express, y postgres, con este módulo lograremos tres objetivos esenciales:

1. crear la conexion de express con la base de datos.
2. definir los modelos.
3. definir las relaciones entre los modelos.

1. crear la conexion de express con la base de datos

para lograrlo, debemos importar sequelize, consultando la [Documentacion de Sequelize](#), ahi encontramos:

- cómo se importa sequelize - métodos y funciones útiles de sequelize.

la sintaxis para importar sequelize es desestructurado y con inicial mayuscula:

```
const { Sequelize } = require("sequelize");
```

Una de las opciones que la documentacion nos ofrece para establecer la conexión es instanciando a sequelize pasandole los siguientes datos con formato:

```
const sequelize = new  
Sequelize("postgres://user:pass@example.com:5432/dbname");
```

editando los valores correspondientes para nuestro caso, haciendo uso de nuestro archivo `.env` para no exponer datos sensibles, quedando algo similar a:

```
require("dotenv").config();
const { Sequelize } = require("sequelize");
const { DB_USER, DB_PASSWORD, DB_NAME, DB_HOST, DB_PORT } = process.env;
const sequelize = new Sequelize(
  `postgres://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}`
);
```

es recomendable agregar `{ logging: false }` como segundo parametro al instanciar nuestro sequelize, para evitar exceso de informacion en la consola.

```
require("dotenv").config();
const { Sequelize } = require("sequelize");
const { DB_USER, DB_PASSWORD, DB_NAME, DB_HOST, DB_PORT } = process.env;
const sequelize = new Sequelize(
  `postgres://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}`,
  { logging: false }
);
```

hecho esto, debemos ir a nuestro index.js en la carpeta raíz, y agregar la sincronizacion de sequelize entre nuestro servidor de express y la base de datos, Agregando `sequelize.sync()` en la callback de `app.listen`, preferiblemente antes del `console.log "server listening"`. El método `sync`, recibe el parámetro `{ force: true }` cuando estamos desarrollando la API, Force true *borra* la base de datos y la vuelve a crear con los datos nuevos cada vez que hacemos reiniciamos el servidor. Cuando ya tengamos nuestro diseño y estructura hechos, cambiamos a `{ alter: true }` el cual en lugar de *borrar* toda la base de datos y hacerla de nuevo, simplemente actualiza lo necesario en cada sesión.

/index.js

```
const app = require(".src/app");
app.listen(3001, () => {
  sequelize
    .sync({ force: true })
    .then(() => {
      //agregamos manejo de errores asincronico
      console.log("Database synced");
    })
    .catch((error) => {
      console.error("Error syncing database:", error);
    });
});
```

2. definir los modelos

Ahora que nuestro servidor de express reconoce tener una base de datos asociada, importaremos cada una de las funciones que definen los modelos de `/src/models/` en `db.js`, para que pueda definir los modelos que crearán cada tabla de la base de datos. Al definir modelos, debemos darle a nuestra instancia de `sequelize` el nombre de la tabla, y cada uno de los campos o características que tendrá cada elemento de esa tabla, *por ejemplo*, un usuario tendría nombre, apellido, numero de documento, eMail, y también la daremos un numero de identificación único en nuestra base de datos.

A demás, cada una de estas características debe tener especificado que tipo de dato es, y que restricciones (*constraints*) debe tener, por ejemplo, si un campo es obligatorio u opcional, estos tipos de dato, vienen definidos por `sequelize` en un objeto llamado **DataTypes**, el cual debe ser importado desestructurado desde `sequelize`

```
const { DataTypes } = require("sequelize");

module.exports = (sequelize) => {
  sequelize.define(
    "User",
    {
      id: {
        type: DataTypes.UUID, //definimos que el id es de tipo
universal unique id
        defaultValue: DataTypes.UUIDV4, // como constraint, le
decimos que se debe generar automaticamente este id, (puede ser v4, v5 etc,
no cambia mucho)
        primaryKey: true, // al ser un id, es necesario que sea un
primary key, si no lo definimos como tal, surgirian problemas, a demás, el
hecho de ser primary key, indica que es unico, que es obligatorio y otros
constraints implícitos
      },
      name: {
        type: DataTypes.STRING,
        allowNull: false,
      },
      email: {
        type: DataTypes.STRING,
        unique: true,
      },
      phone: {
        type: DataTypes.STRING,
        allowNull: false,
      },
    },
    { timestamps: false }
  );
};
```

aquí hemos definido la tabla de usuarios y el contenido que debe tener cada usuario según nuestro criterio, *no es una fórmula fija*, es decir en cada proyecto podremos decidir si permitiremos que dos usuarios compartan el mismo email y en consecuencia le pondremos o no el constraint unique, ó si permitiremos que se deje el campo de email vacío, eso es a discreción del desarrollador y del proyecto.

por último en la tabla, le damos la instrucción de timestamps false, ya que por defecto se crea un registro de fecha y hora en la que ha sido creado cada elemento de cada tabla, y por el momento, no deseamos ver tanta información en nuestra base de datos.

Cada archivo de la carpeta models, exporta una función que recibe la instancia de sequelize, y crea el elemento en la tabla correspondiente, dicha función será importada en db.js para ser usada.

```
const { UserModel } = require('./models/User.js');
.
.
.
UserModel(sequelize);

module.exports = { sequelize };
```

tendremos tantos modelos importados y ejecutados como necesitemos.

otra forma de hacer "automáticamente" este proceso de traer todos los modelos a db.js es (la forma que está en los boilerplates de los PI):

```
const basename = path.basename(__filename);

const modelDefiners = [];

// Leemos todos los archivos de la carpeta Models, los requerimos y agregamos
al arreglo modelDefiners
fs.readdirSync(path.join(__dirname, "/models"))
  .filter(
    (file) =>
      file.indexOf(".") !== 0 &&
      file !== basename &&
      file.slice(-3) === ".js"
  )
  .forEach((file) => {
    modelDefiners.push(require(path.join(__dirname, "/models", file)));
  });

// Injectamos la conexión (sequelize) a todos los modelos
modelDefiners.forEach((model) => model(sequelize));
// Capitalizamos los nombres de los modelos ie: product => Product
let entries = Object.entries(sequelize.models);
let capsEntries = entries.map((entry) => [
```

```

    entry[0][0].toUpperCase() + entry[0].slice(1),
    entry[1],
  ]);
  sequelize.models = Object.fromEntries(capsEntries);

  // En sequelize.models están todos los modelos importados como propiedades
  // Para relacionarlos hacemos un destructuring
  const { User, Model2, Model3 } = sequelize.models;

```

3. definir las relaciones entre los modelos

si tuviésemos, *por ejemplo*, usuarios y posts:

- un usuario, puede hacer *varios* posts.
- un post, puede haber sido generado únicamente por *Un* usuario.
esto implica una **relacion de uno a muchos**.

para hacer la relacion debo:

1. desestructurar los modelos que voy a relacionar, obteniendolos de sequelize.models.
2. relacionarlos.

```

const { User, Post } = sequelize.models;
User.hasMany(Post);
Post.belongsTo(User);

module.exports = { sequelize };

```

si tuviese, *por ejemplo*, videojuegos y géneros:

- un videojuego, puede pertenecer a *varios* generos.
- un género, puede agrupar *varios* videojuegos.
esto implica una **relacion de muchos a muchos**.

para hacer la relacion debo:

1. desestructurar los modelos que voy a relacionar, obteniendolos de sequelize.models.
2. relacionarlos.

```

const { Videogame, Genre } = sequelize.models;
Videogame.belongsToMany(Genre, {
  through: "videogame_genres",
  timestamps: false,
});
Genre.belongsToMany(Videogame, {

```



```
    through: "videogame_genres",
    timestamps: false,
  });

module.exports = { sequelize };
```

siendo videogame_genres la tabla intermedia que relaciona las dos tablas principales.

por último, además de exportar nuestra instancia de sequelize para que pueda ser usada por los modelos, también debo exportar los modelos ya creados y relacionados:

```
module.exports = { sequelize, ...sequelize.models };
```

y obtener un archivo db.js similar a:

```
const { Sequelize } = require("sequelize");
const { UserModel } = require("../models/User.js");
const { PostModel } = require("../models/Post.js");
require("dotenv").config();
const { DB_USER, DB_PASSWORD, DB_NAME, DB_HOST, DB_PORT } = process.env;

const sequelize = new Sequelize(
  `postgres://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}`,
  { logging: false }
);

UserModel(sequelize);
PostModel(sequelize);

const { User, Post } = sequelize.models;

User.hasMany(Post);
Post.belongsTo(User);

module.exports = { sequelize, ...sequelize.models };
```

y ya estaríamos listos para pedirle a nuestros handlers que manejen la base de datos.

un handler para pedir todos los usuarios puede verse así:

```
const getUsers = require("../controllers/getUsers");
// como mencionabamos, usaremos nuestro handler dentro de su lógica, llamará
a la función del controller, por eso lo requerimos
// usaremos una función asíncronica, ya que la respuesta de la api o base de
```

datos, suele tardar un tiempo, que aunque es muy corto, el código de javascript se lee mas rapido, y para el momento en que llegue la respuesta, el compilador ya hubiese optado por llenar nuestra variable "getUsersHandler" con algun valor inesperado como NULL o undefined

```
const getUsersHandler = async (req, res) => {
  console.log("fetchingUsers-handler");
  // podemos usar logs de consola como este, que indican lo que esta
  // haciendo el programa y en que parte del código está en ese momento... estos
  // logs deben ser eliminados antes de desplegar la app a producción.
  try {
    const users = await getUsersController(); //llamamos al controller, y
    // si sale todo bien ->
    res.status(200).json(users); //enviamos al cliente una respuesta con
    // código de éxito y un contenido, que en este caso es un JSON con todos los
    // usuarios.
    //si No sale bien el error será atrapado por este catch, que enviará
    // el código de el error correspondiente y el error a consola para que pueda ser
    // analizado por el desarrollador ->
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};

module.exports = getUsersHandler;
```

este handler, además de simplemente interactuar con la api y la base de datos, debe darle formato a la información recibida, en este caso específico la api nos da una cantidad sumamente alta de usuarios, en paquetes de 20 usuarios por pagina, por lo que decidimos darle al cliente únicamente los primeros 100 usuarios.

debido a esto, tenemos que desarrollar una logica que guarde los usuarios de la primera pagina, seguidos de los usuarios de la segunda pagina y así sucesivamente hasta que tengamos en nuestro poder los primeros 100 usuarios.

además, solicitamos los valores de ciertas variables de entorno, los cuales representan datos sensibles, que no deben ser expuestos en nuestro código, por ejemplo el usuario y contraseña de nuestra base de datos, la llave privada de la api a la que estamos accediendo, u otra información sensible.

el controller correspondiente a este handler, se vería por ejemplo de la siguiente manera:

```
const { User } = require("../db"); //le pedimos a nuestro archivo db, la
// definicion de usuario
const axios = require("axios"); //usaremos la libreria axios para hacer
// peticiones http
require("dotenv").config(); // con esta linea importamos lo necesario para
// manejar las variables de entorno, la ultima version de node ya lo trae de
// forma nativa por lo que no habria necesidad de importarlo
const { USER_URL, API_KEY } = process.env; //extraemos la llave personal y la
// url del endpoint al cual solicitaremos los usuarios de la api, que se
```

encuentra en nuestro archivo de variables de entorno

```
const getUsers = async () => {
  try {
    console.log("fetchingALL-controller", USER_URL); //log a consola para
    depuracion y debugging
    const APIUsers = fetch100();
    //la funcion fetch100 la definiremos por fuera, al final del archivo,
    para hacer el código mas limpio. esta funcion se encarga de buscar los
    primeros cien usuarios de la api

    //para finalizar, uniremos los usuarios creados por el cliente en
    nuestra base de datos privada, con los 100 usuarios que recolectamos de la
    API

    const DBUsers = await User.findAll(); //aplicamos el metodo findAll
    de sequelize para pedir a la base de datos todos los elementos de la tabla
    User

    // en el return podemos concatenar los dos arrays o simplemente usar
    el spread operator del siguiente modo
    return [...DBUsers, ...APIUsers];
  } catch (error) {
    console.error({ error: error.message });
  }
};

const fetch100 = () => {
  let allUsersArray = []; //declaro un array vacío para almacenar todos los
  usuarios
  let pageNum = 1; //defino el numero de la pagina de la cual extraeré
  usuarios(referente a las paginas de 20 usuarios que me da la api)

  while (allUsersArray.length < 100) {
    let page = `&page=${pageNum}`;
    const url = `${USER_URL}?key=${API_KEY}${page}`;
    const response = await axios.get(url);
    console.log(response.data); // mas logs para debuggear(que deben ser
    eliminados una vez todo esté listo)
    const { results, next } = response.data;
    // de todo lo que devuelve la api, me quedo solo con el array de 20
    resultados y la url de la siguiente pagina de 20 resultados
    allUsersArray = allUsersArray.concat(clean(results));
    // la funcion clean tambien estará modularizada, esta funcion se
    encarga de dejar solo los datos que necesitamos ya que las api tienen muchos
    datos que no usaremos, y para que los usuarios de la API tengan la misma
    estructura que los de nuestra base de datos, es decir solo tendran id, name,
    email y phone.
```

```
    if (!next) {
```

```

        break;
        // en caso de que no haya mas paginas, corta el ciclo
    }

    pageNum++;
}
return allUsersArray;
}

const clean = (array) => array.map((element)=>{
    return {
        id: elem.id,
        name: elem.name,
        email: elem.email,
        phone: elem.phone,
    };
});

//a demas de exportar la funcion get users, dejaré las otras dos funciones
para que otros modulos las puedan usar
module.exports = {
    getUsers,
    fetch100,
    clean,
};

```

en otro ejemplo, veamos un handler para crear un nuevo usuario en nuestra base de datos.

```

const { createUser } = require("../controllers/createUserController");
const createUserHandler = async(req, res) => {
    try {
        console.log("creatingUser-handler");
        const { name, email, phone} = req.body;
        const createUser(name, email, phone);
        res.status(201).json(newUser);
    } catch (error) {
        res.status(400).json({ error: error.message });
    }
};

module.exports = createUserHandler;

```

```

const { User } = require("../db");
const createUser = async (name, email, phone) => {
    const newUser = await User.create({ name, email, phone });
    return newUser;
};

```

```
};  
  
module.exports = createUser;
```

y por último veamos un ejemplo en el que buscaremos por id un usuario diferenciando si el usuario se encuentra en una API pública, o en nuestra db.

para este fin, nos podemos valer de que en las APIs públicas que hemos usado, los elementos tienen un id numérico, y nuestra base de datos, usa UUID (alfanumérico).

```
const { getUserByID } = require("../controllers/createUserController");  
const getUserByIDHandler = async (req, res) => {  
  console.log("fetchingUserByID-handler");  
  const { id } = req.params;  
  const source = isNaN(id) ? "DB" : "API";  
  try {  
    const user = await getUserByID(id, source);  
    res.status(200).json(user);  
  } catch (error) {  
    res.status(400).json({ error: error.message });  
  }  
};  
  
module.exports = getUserByIDHandler;
```

```
const { User } = require("../db");  
const getUser = async (id, source) => {  
  const user =  
    source === "API"  
      ? (await axios.get(`apiejemplo.com/users/${id}`)).data  
      : await User.findById(id);  
  return user;  
};  
  
module.exports = createUser;
```

Ha surgido un cambio de última hora,

y ahora, nos piden que el handler que pide todos los usuarios, también tenga la capacidad de recibir un nombre, y si lo recibe debe mostrar solo los usuarios que coincidan.

para este caso, crearemos un nuevo controller, y solo haremos una modificación al handler.

el handler hasta ahora simplemente llamaba al controller y devolvía la respuesta de éxito o de error, ahora, vamos a encargarnos de que el handler, decida cuál controller va a llamar (buscar todos o buscar por nombre)

```

const getUsers = require("../controllers/getUsers");
const getUsersByName = require("../controllers/getUsersByName");
const getUsersHandler = async (req, res) => {
  console.log("fetchingUsers-handler");
  try {
    //para esta modificacion simplemente obtenemos el name (del query
    string en este caso) y si existe, pedimos por nombre, si no existe hacemos lo
    que haciamos anteriormente, es decir, buscar todos con el controller que ya
    esta construido. asi que nos falta construir nuestro nuevo controller para
    buscar por nombre.
    const { name } = req.query;
    const users = name ? await getUsersByName(name) : await getUsers();
    res.status(200).json(users);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};

module.exports = getUsersHandler;

```

```

const { User } = require("../db");
const getUsers = require("../getUsers");
const { fetch100, clean } = getUsers;
const { Op } = require("sequelize");

const getUsersByName = async (NAME) => {
  const dbUsers = await User.findAll(
    { where:
      { name:
        [Sequelize.Op.iLike]: `%${NAME}%`
        //Op.iLike hace insensible la busqueda a mayus/minus, debemos
        importar Op de Sequelize.
        //encerrar name entre % hace que la base de datos busque
        incluso si name esta antes o despues de otro caracter o valor, por ejemplo si
        busco juan, me devolverá tambien a juana, o si busco luis, me devolverá
        tambien a jose luis.
      }
    }
  );
  // suponiendo que nuestra api pública no tiene un endpoint que nos deje
  buscar por name, desrrollaremos el codigo que lo logre.

  const APIUsers = fetch100().filter(
    (user) => user.name.toLowerCase().includes(NAME.toLowerCase())
  );//de esta forma se filtran los resultados que traiga fetch100

  return [...dbUsers, ...APIUsers];

```

```
};

module.exports = getUsersByName;
```

estos han sido ejemplos de handlers de un elemento de tipo hasMany. hagamos algun ejemplo de posts, ya que es de tipo belongsTo y tiene metodos de sequelize diferentes a demas de contar con clave foránea

clave foránea

los posts en nuestro ejemplo tienen una clave foránea debido a que están relacionados necesariamente con un usuario, entonces en la tabla de la base de datos, se creará la columna `userId` de forma automática gracias a sequelize sin necesidad de que en nuestro modelo `Post` esté definida dicha columna, este `user id` representa al usuario al cual está relacionado cada post

```
//handler
const createPost = require("../controllers/createPost");

const createPostHandler = async (req, res) => {
  try {
    const { title, body, userID } = req.body;
    const newPost = await createPost(title, body, userID);
    res.status(200).json(newPost);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
}

module.exports = createPostHandler
```

```
//controller
const { Post } = require("../db");
const createPost = async (title, body, userID) => {
  const newPost = await Post.create({
    title,
    body,
  });
  await newPost.setUser(userID);
  return newPost;
};

module.exports = createPost;
```

los metodos de los modelos devuelven promesas

`model.create`

`model.findOrCreate`

```
model.findAll
model.findByPk
model.find
model.setOther
```

tomando en cuenta estas ultimas modificacines a nuestra arquitectura de carpetas y archivos, tendremos algo como lo siguiente:

Estructura básica del proyecto

```
"/" carpeta raíz.
/package.json
/.gitignore
/.env
/index.js
/src

  /src/app.js
  /src/db.js
  /src/models

    /src/models/user.js
    /src/models/otherTable.js

  /src/handlers

    /src/handlers/UsersHandlers

      /src/handlers/getUsersHandler
      /src/handlers/getUserbyIDHandler
      /src/handlers/createUserHandler
      /src/handlers/deleteUserHandler
      /src/handlers/modifyUserHandler

  /src/controllers

    /src/handlers/UsersControllers

      /src/controllers/getUsers
      /src/controllers/getUsersByName
      /src/handlers/getUserbyID
      /src/controllers/createAllUsers
      /src/controllers/deleteUser
      /src/controllers/modifyUser
```


/src/routes

/src/routes/index.js

/src/routes/usersRoutes.js

/src/routes/postsRoutes.js

/src/routes/otherRoutes2.js