Tech Returners

🧪👾 Welcome to the Connecting to Databases Lab

TypeScript / PostgreSQL Version

In this lab, you'll explore how to connect your Minimalist Book Manager API to work with a PostgreSQL database. You'll also learn about the practice of externalising application configuration/settings in the context of a Node.js Express application.

You'll learn and apply the concept of externalising application configuration/settings through interacting with the `.env` file and using the Dotenv library.

🛠️ What do I need for this Lab?

You will need:

- To have **forked** and **cloned** the following Starter Minimalist Book Manager API project:
    - GitHub - techreturners/lm-code-book-manager-api-ts
    - If you had already forked and cloned this repository as part of an earlier lab or assignment in this series, please feel free to utilise your existing Book Manager API project.
- PostgreSQL
- Postman Desktop Application
- Google Chrome Web Browser
- Visual Studio Code

0️⃣ Background Knowledge

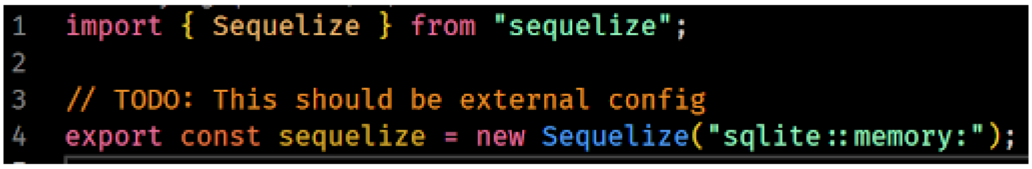
This lab assumes that you already have PostgreSQL installed and have working knowledge of starting and stopping a PostgreSQL Server, as well as performing basic interactions with a PostgreSQL database such as SELECT or INSERT statements.

This lab also assumes that you already have the Postman Desktop Application installed, and that you have some familiarity with HTTP.

1️⃣Explore the Database connection setup

The Minimalist Book Manager API is currently utilising a SQLite In-Memory database. Let's remind ourselves of the clues that indicate this.

🔎 Navigate to the `database.ts` file within the **src/database** directory and look at the code.



```
import { Sequelize } from "sequelize";

// TODO: This should be external config
export const sequelize = new Sequelize("sqlite::memory:");
```

🔎 Notice the import from the `sequelize` library. It is possible to manually write all of our database code, including manually writing SQL commands to execute, but it is much **easier** and **safer** to use a library which abstracts over the database. *Sequelize* is one of the most popular JavaScript (and hence TypeScript) libraries for connecting to databases.

💡 There are many similar libraries to choose from, including more modern approaches such as Prisma which makes powerful use of TypeScript features. However, Sequelize is a great choice and, at the time of writing, is used more widely in production thanks to its huge headstart.

🔎 Notice the parameter being passed to Sequelize is listed as `sqlite::memory:`. That parameter is the **database connection string**.

> A **connection string** is what it sounds like: a *string* which tells a database library how to connect to a particular database or server. These come in all shapes and sizes, depending on which kind of database we're connecting to. You can find handy reference for many different types of connection string at ConnectionStrings.com

Specifically this connection string tells us that the Sequelize engine is connecting to a SQLite database running in memory.

🔎 Notice that this connection string is **hard-coded**. If we wanted to change that connection string we would have to update our code and re-build then re-deploy the application 😔

2️⃣Connecting your Minimalist Book Manager API to a PostgreSQL Database - Adding the PostgreSQL Dependency

Since the Minimalist Book Manager API utilises an In-Memory database, the application does not initialise with any books.

> An In-Memory database uses temporary memory for data storage; whereas a "real" database persists data using permanent disk storage. This means when the application is shut down, the data held in an In-Memory database is cleared.

If a book is added to the In-Memory database using the POST `api/v1/books` endpoint, that book will only be available for the lifetime that the application is running. It disappears when the app stops running because the in-memory database gets cleared.

To make using the app in development more convenient, we populate the in-memory database with a couple of books on startup.

🔎 If you look in the **src/server.ts** file you'll see we call a function named `populateDummyData()` to populate the database with sample data…

… but only if we're in the `dev` environment. In production we would not want to mess with the contents of the database every time the app starts! 😱

🔧 In-memory databases are useful tools for development and testing as they reset on each app start. But this approach is clearly inadequate for production. So let's get your Minimalist Book Manager API connected to a real PostgreSQL database.

💡 The steps to configure Sequelize to connect to a database server are similar, regardless of which relational database engine you use. In this lab, we're going to connect to a PostgreSQL database, but connecting to, say, MySql or MariaDb would be very similar.

🔎 Navigate to your **package.json** file. You should notice that we already include the **sqlite3** dependency:

```
"devDependencies": {
  ...
  "sqlite3": "^5.1.4",
  ...
}
```

🔎 To use PostgreSQL, we need install a package which Sequelize can use to communicate with a PostgreSQL server.

👉 Open the terminal and ensure you are at the root of your code base (where the `package.json` file is located) and run the following command:

```
npm install -D pg pg-hstore dotenv @types/pg
```

💡 This will install a number of important packages:

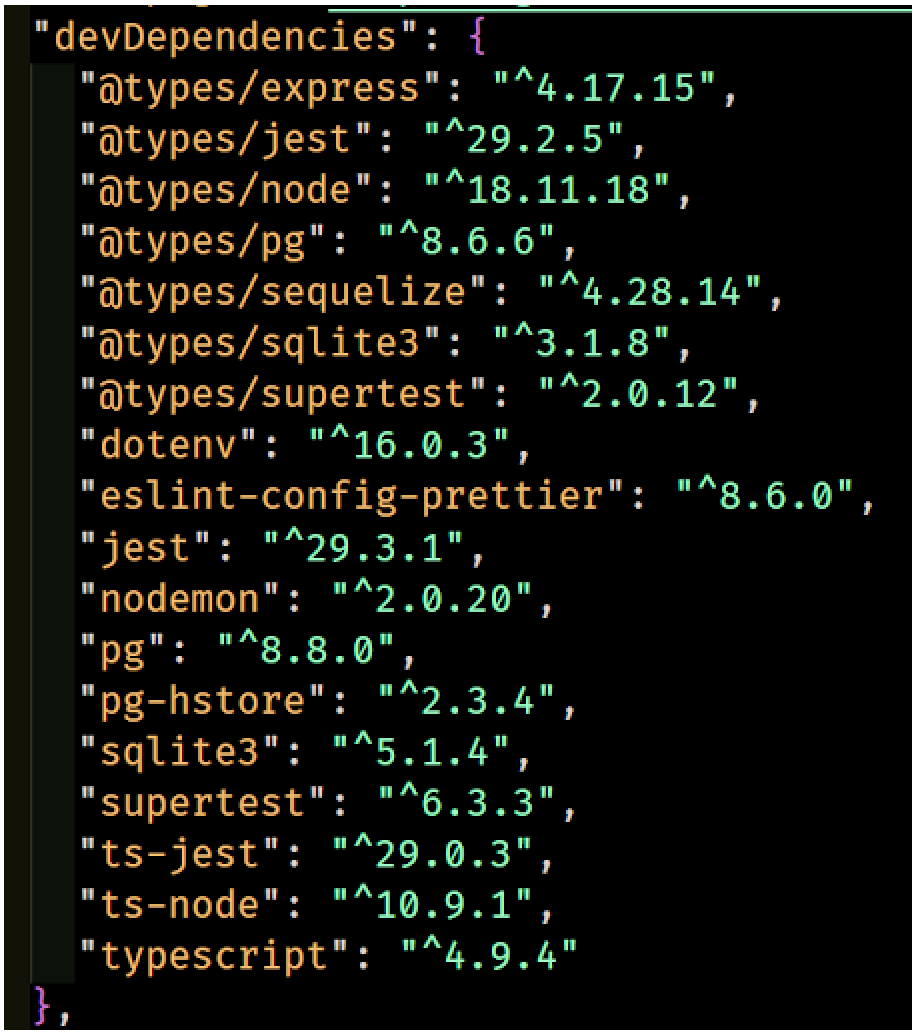
1) **pg** - This library acts as a database-driver. Since sequelize can communicate with a large number of RDBMS programs (i.e SQL Server, MySql, Postgres, etc) it doesn't include drivers for every possible RDBMS by default.
2) **@types/pg** - As usual, some packages don't include their own type information. We can add type information for most popular packages by installing from @types/package-name
3) **pg-hstore** - This will be handy for converting data to JSON
4) **dotenv** - We'll use this later to load our application configuration files.

See here for more details:

https://www.npmjs.com/package/pg

https://www.npmjs.com/package/dotenv

Your dependencies should now include `pg`, `pg-hstore` and `dotenv`, as well as some types for `pg`:



```
"devDependencies": {
  "@types/express": "^4.17.15",
  "@types/jest": "^29.2.5",
  "@types/node": "^18.11.18",
  "@types/pg": "^8.6.6",
  "@types/sequelize": "^4.28.14",
  "@types/sqlite3": "^3.1.8",
  "@types/supertest": "^2.0.12",
  "dotenv": "^16.0.3",
  "eslint-config-prettier": "^8.6.0",
  "jest": "^29.3.1",
  "nodemon": "^2.0.20",
  "pg": "^8.8.0",
  "pg-hstore": "^2.3.4",
  "sqlite3": "^5.1.4",
  "supertest": "^6.3.3",
  "ts-jest": "^29.0.3",
  "ts-node": "^10.9.1",
  "typescript": "^4.9.4"
},
```

🔎 Take a look at the **book.ts** file in the **src/models** directory and the **books.ts** file in the **src/services** directory. Notice that neither file specifies any SQL. This is the power of the libraries like Sequelize - they *abstract* over the database engine for you so that the library will generate the SQL to query your target database.

🔎 In **book.ts** we define a Sequelize model and then the library takes care of the rest. That model represents a table in your database. This means that we won't have to change our code when we connect to a real database!

✋⛔ STOP ✋⛔

📚 This would be a great time to read more about how Models work in the Sequelize documentation: Model Basics | Sequelize

---

3️⃣Using Dotenv and setting up the connection settings `.env` file

Before we try to connect to our Postgres server, it would be great to be able to change our database settings without changing any code. We can then use these settings to control whether we connect to an in-memory database or our real database.

We can do this by moving our settings into an `environment` file. These files allow for secure storage of secrets, such as database user names and passwords, as well as a convenient place to store application settings outside of code.

👉 Firstly create a file called **.env** at the root of your project. (The same location as the package.json file)

NOTE: The dot at the start of that filename is **important.**

👉 It's `.env` not `env`

🔎 Add the following contents to your `.env` file:

```
NODE_ENV=dev
PORT=3000
DB_NAME=sqlite::memory:
DB_USERNAME=
DB_PASSWORD=
DB_HOST=localhost
DB_PORT=
DB_DIALECT=sqlite
```

This configuration file is going to hold our database connection settings.

❗ `.env` files and git

🔎 Always ensure your .env files are NOT managed by git, and are not considered part of the repository. Check that your **.gitignore** includes the `.env` file

This is very important! For security purposes we don't want to include configuration files in version control, as otherwise we make our secrets *permanently part of the codebase* 😨😱

🔎 Next we need to write some code to load our new config.

👉 Create /src/config.ts

👉 Add this code to config.ts

```
import * as dotenv from "dotenv";

dotenv.config();

export const CONFIG = {

     port: process.env.PORT ?? 3000,

     dbName: process.env.DB_NAME ?? "sqlite::memory:",

     dbUserName: process.env.DB_USERNAME ?? "",

     dbPassword: process.env.DB_PASSWORD ?? "",

     dbHost: process.env.DB_HOST ?? "localhost",

     dbDialect: process.env.DB_DIALECT ?? "sqlite",

} as const;
```

👉 Note the as const - this tells TypeScript that we don't want to accidentally be allowed to change any of these properties.

👉 Hover the CONFIG object and see that all the properties are marked `readonly`. This is TypeScript telling us that it'll complain and give us red squigglies if we try to alter this object anywhere.

👉 Note that this isn't exactly typesafe. You might be worrying about what's stopping us putting a string in our port or a number in our dbHost! The answer is: nothing!

We could improve this by using a validation library like `zod` and creating a schema for our environment variables. This would validate at runtime that what is in our .env file actually matches the types we want them to be. But, for now, this approach is sufficient.

🔎 Let's try using some of the properties we put in our `.env` file.

👉 In server.ts, import the CONFIG object from your new config.ts file.

👉 On line 9 of the code, replace the hard-coded port number with `CONFIG.port`

👉 Run npm start and notice that the port is still 3000 - it must have loaded this from config!

👉 Change the port in the .env file and hit save. Nothing happens!

💡 LESSON: environment variables are loaded only once. Specifically, they are loaded on startup when dotenv.config() is called.

👉 Close the app and rerun `npm start` to see your new port being used.

4️⃣Connecting your Minimalist Book Manager API to a PostgreSQL Database - Adding the Database Connection String to the `.env` file

To find out the database connection string to your PostgreSQL database, you will need to check the port that the database is using on your local machine.

🔎 Start PostgreSQL Server.

🔎 Open up your terminal / command prompt and run the following to utilise the PostgreSQL client.

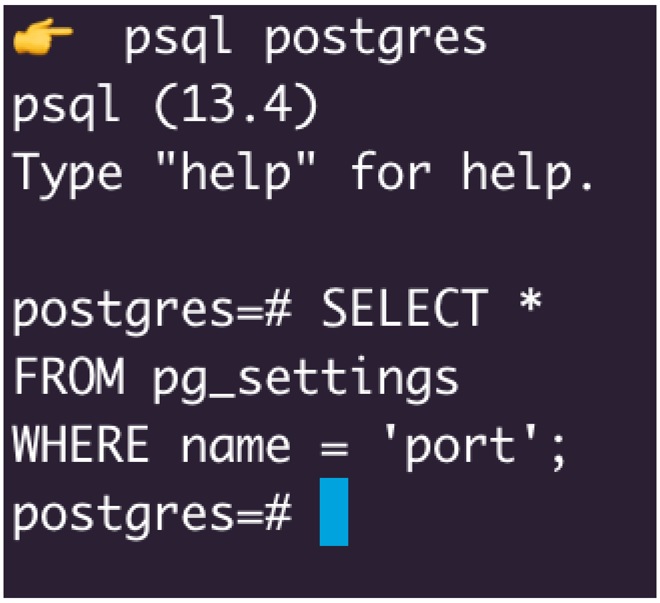
```
psql postgres
```

💡 You may choose to utilise pgAdmin instead of the terminal if you prefer.

👉 At the `postgres=#` command prompt, type `\l` to list out all the databases.

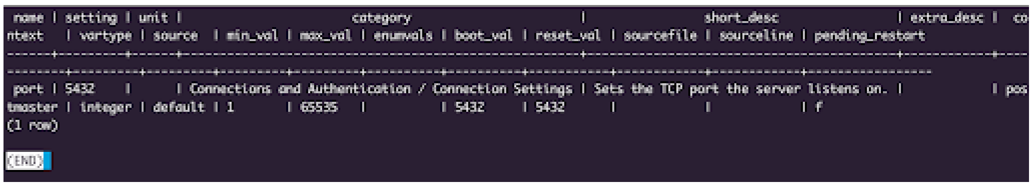
🔎 Run the following statement to check which port PostgreSQL is running on. The port will usually be `5432` , but you may have PostgreSQL running a different port.

```
SELECT * FROM pg_settings WHERE name = 'port';
```



```
👉  psql postgres
psql (13.4)
Type "help" for help.

postgres=# SELECT *
FROM pg_settings
WHERE name = 'port';
postgres=# 
```

Running the SQL statement to find out which port PostgreSQL is running on



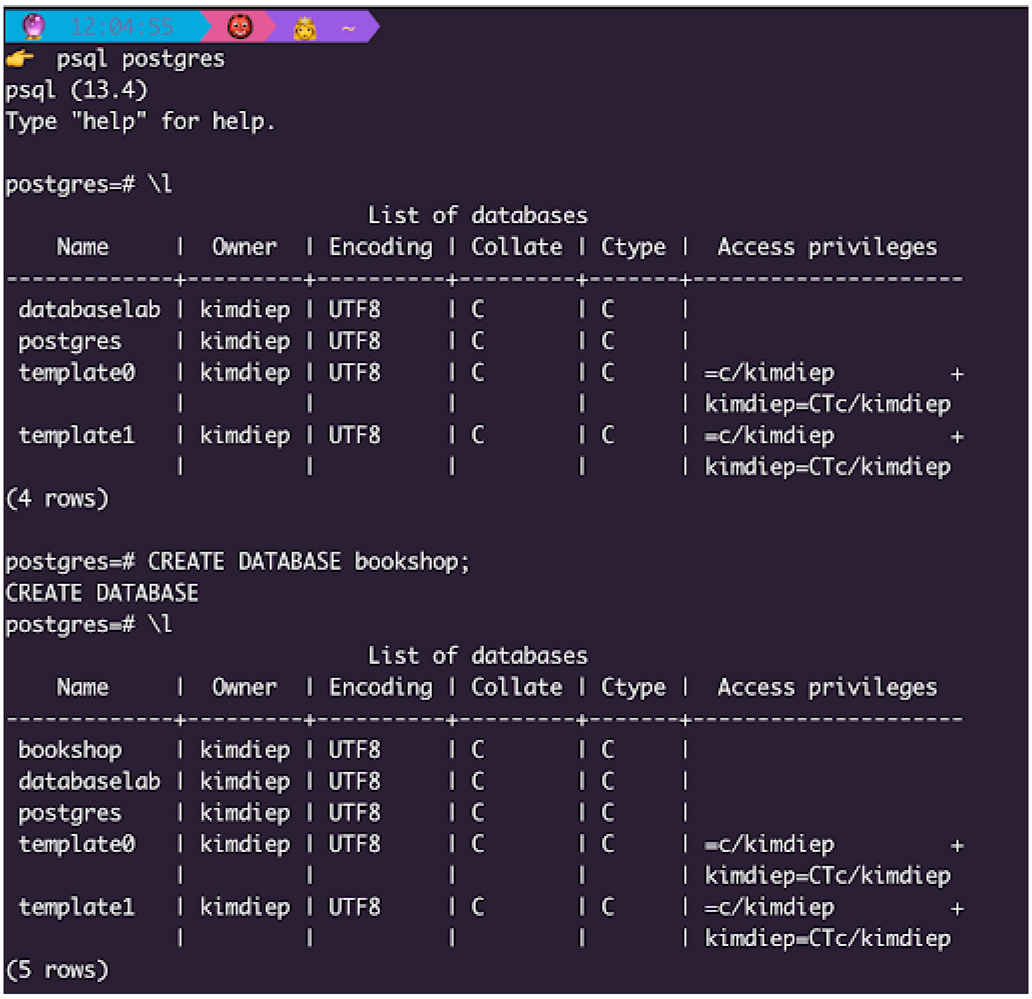
```
 name | setting | unit |                       category                       |                short_desc                | extra_desc |  co
ntext   | vartype | source  | min_val | max_val | enumvals | boot_val | reset_val | sourcefile | sourceline | pending_restart
------+---------+------+------------------------------------------------------+------------------------------------------+------------+----
--------+---------+---------+---------+---------+----------+----------+-----------+------------+------------+-----------------
 port | 5432    |      | Connections and Authentication / Connection Settings | Sets the TCP port the server listens on. |            | pos
tmaster | integer | default | 1       | 65535   |          | 5432     | 5432      |            |            | f
(1 row)

(END)
```

This is an example output from the SQL SELECT query which indicates the port that PostgreSQL is running on is `5432`. Your port may differ.

👉 Let’s create a new database called `bookshop` for our app to use. You may already have this database set up from a previous lab or assignment. If not, run the following SQL statement:

```
CREATE DATABASE bookshop;
```

👉 Type `\l` to list out all of the databases to double-check that the `bookshop` database has been created successfully. Notice the username listed in the `Owner` column against the `bookshop` database.



```
👉  psql postgres
psql (13.4)
Type "help" for help.

postgres=# \l
                              List of databases
    Name     |  Owner  | Encoding | Collate | Ctype |  Access privileges
-------------+---------+----------+---------+-------+---------------------
 databaselab | kimdiep | UTF8     | C       | C     |
 postgres    | kimdiep | UTF8     | C       | C     |
 template0   | kimdiep | UTF8     | C       | C     | =c/kimdiep         +
             |         |          |         |       | kimdiep=CTc/kimdiep
 template1   | kimdiep | UTF8     | C       | C     | =c/kimdiep         +
             |         |          |         |       | kimdiep=CTc/kimdiep
(4 rows)

postgres=# CREATE DATABASE bookshop;
CREATE DATABASE
postgres=# \l
                              List of databases
    Name     |  Owner  | Encoding | Collate | Ctype |  Access privileges
-------------+---------+----------+---------+-------+---------------------
 bookshop    | kimdiep | UTF8     | C       | C     |
 databaselab | kimdiep | UTF8     | C       | C     |
 postgres    | kimdiep | UTF8     | C       | C     |
 template0   | kimdiep | UTF8     | C       | C     | =c/kimdiep         +
             |         |          |         |       | kimdiep=CTc/kimdiep
 template1   | kimdiep | UTF8     | C       | C     | =c/kimdiep         +
             |         |          |         |       | kimdiep=CTc/kimdiep
(5 rows)
```

`CREATE DATABASE bookshop;` and `\l` to check that the database has been created successfully

👉 It would be good to create a specific user for our app to use. Create a user: (If you like you can replace the password with another.)

```
CREATE USER bookshopuser WITH PASSWORD
'super-secret-password';
```

👉 And let's give that user control over the `bookshop` database:

```
GRANT ALL PRIVILEGES ON DATABASE bookshop TO bookshopuser;
```

Normally it is good practice to be more restrictive with permissions than this. For example, our API doesn't *really* need to create/delete entire **tables**, just data in those tables, so we shouldn't really give it `ALL` privileges. However, for the sake of ease, we're being lazy in this instance 😴 Just be aware of this in the real world!

👉 Now use the `\c` command to connect to the `bookshop` database.

```
\c bookshop;
```

👉 And then create a table in the bookshop called `Books`

```
CREATE TABLE IF NOT EXISTS "Books" ("bookId" INTEGER PRIMARY
KEY, "title" VARCHAR(255) NOT NULL, "author" VARCHAR(255) NOT
NULL, "description" VARCHAR(255) NOT NULL);
```

👉 Grant your user access to the tables too:

```
GRANT ALL ON TABLE public.”Books” TO bookshopuser;
```

👉 We can now insert a book into the Books table

```
INSERT INTO "Books" VALUES (1, 'Fantastic Mr. Fox', 'Roald
Dahl', 'This book concerns a fox who is fantastic.');
```

🔎 Now let's change database settings to start connecting our API to the real database.

🔎 Open up the **database.ts** file located in the **src/database** directory. It currently looks like this:

```
import { Sequelize } from "sequelize";

// TODO: This should be external config
export const sequelize = new Sequelize("sqlite::memory:");
```

🔎 Update this file to use the properties you've set up in your **.env.dev** file:

```
import { Dialect, Sequelize } from "sequelize";

export let sequelize = new Sequelize("sqlite::memory:");

if (process.env.NODE_ENV !== 'test') {
  sequelize = new Sequelize(process.env.DB_NAME ??
'MISSING_DB_NAME_CONFIG',
          process.env.DB_USERNAME ??
'MISSING_DB_USERNAME_CONFIG',
          process.env.DB_PASSWORD ??
'MISSING_DB_PASSWORD_CONFIG', {
    host: process.env.DB_HOST ?? 'MISSING_DB_HOST_CONFIG',
    port: parseInt(process.env.DB_PORT as string) ??
"MISSING_DB_PORT_CONFIG",
    dialect: (process.env.DB_DIALECT as Dialect) ??
'postgres',
  });
}
```

🤔 Let's think a bit more about some of these changes…

🔎 By default, we use an In-Memory database. However, if we're NOT in a test environment, then we replace that connection with whatever is configured for our environment. This structure would allow us to create a test environment that works against a known, pre-seeded database.

🔎 Notice that TypeScript forces us to provide defaults, as there's no guarantee that `process.env.DB_NAME` will exist. In this case, it's sensible to provide some obvious default that will indicate exactly what the problem is if we see it pop up in an error log.

🔎 `Dialect` is a type that is part of Sequelize. We have to indicate to TypeScript that whatever we put in here ought to be one of the defined dialects that Sequelize speaks. We'll default to `postgres` for now as we're planning to connect to PostgreSQL.

👉 See here for more on `Dialect` :
https://sequelize.org/docs/v6/other-topics/dialect-specific-things/#postgresql

❗ Notice in the documentation above that the `postgres` dialect depends on a library called "pg" to speak to PostgreSQL. That's the `pg` library that we installed earlier!

👉 The final step is to alter your `.env` file. Change the settings to add the correct database name, username, password, etc.

👉 Restart your API.

🔎 Try hitting your API to GET all books - you should see books from your PostgreSQL database now instead of the in-memory SQLite database.

HELP!

If you get connection errors, you may find it easier to replace your database connection code with a URL-based connection format:

```
const connString =
"postgres://bookshopuser:super-secret-password@127.0.0.1:5432/
bookshop";

// TODO: Replace the above string with a string built from
YOUR process.env.DB_USERNAME etc variables

sequelize = new Sequelize(connString);
```

🎉 Well done! You just successfully swapped out the connection to the SQLite In-Memory Database with the connection string to the PostgreSQL Database!

The next step is to see this in action.

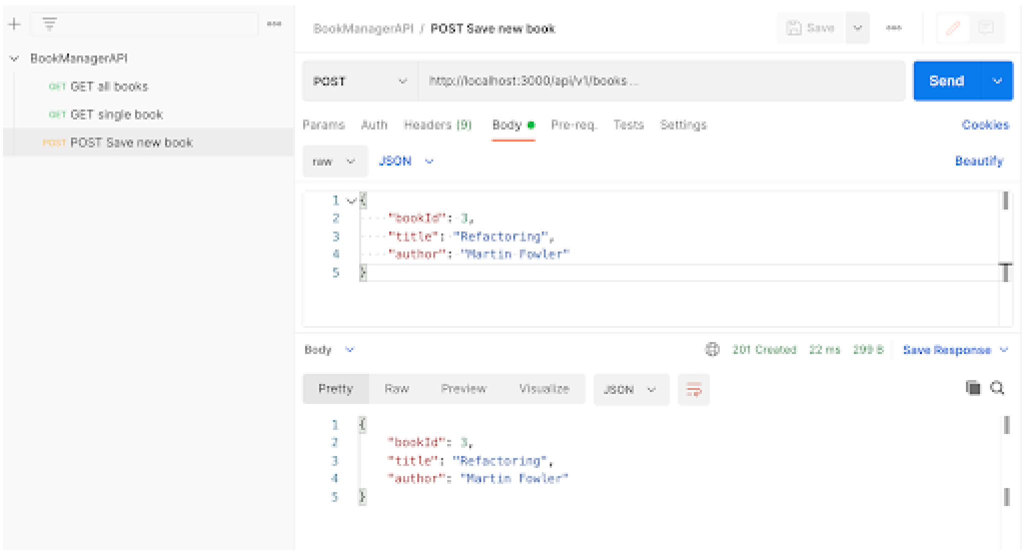
5️⃣Use the Postman Desktop Application to interact with the Minimalist Book Manager API

Since the Minimalist Book Manager API is now connected to a real database, the data will persist in database storage even when the Book Manager API shuts down.

🔎 Try adding some data into your `Book` table. Using the Postman Desktop Application, you can make a `POST` request to the http://localhost:3000/api/v1/books endpoint to add a book.

Here are some example JavaScript Object Notation (JSON) formats for some books you can post inside the HTTP request body.

```
{
    "bookId": 2,
    "title": "You Don't Know JS Yet: Get Started",
    "author": "Kyle Simpson",
    "description": "Learn JS with Kyle Simpson"
}

{
    "bookId": 3,
    "title": "Refactoring",
    "author": "Martin Fowler",
    "description": "How to refactor"
}


{
    "bookId": 4,
    "title": "The Shop Before Life",
    "author": "Neil Hughes",
    "description": "Before being born, each person must visit
the magical Shop Before Life, where they choose what kind of
person they will become down on Earth..."
}
```



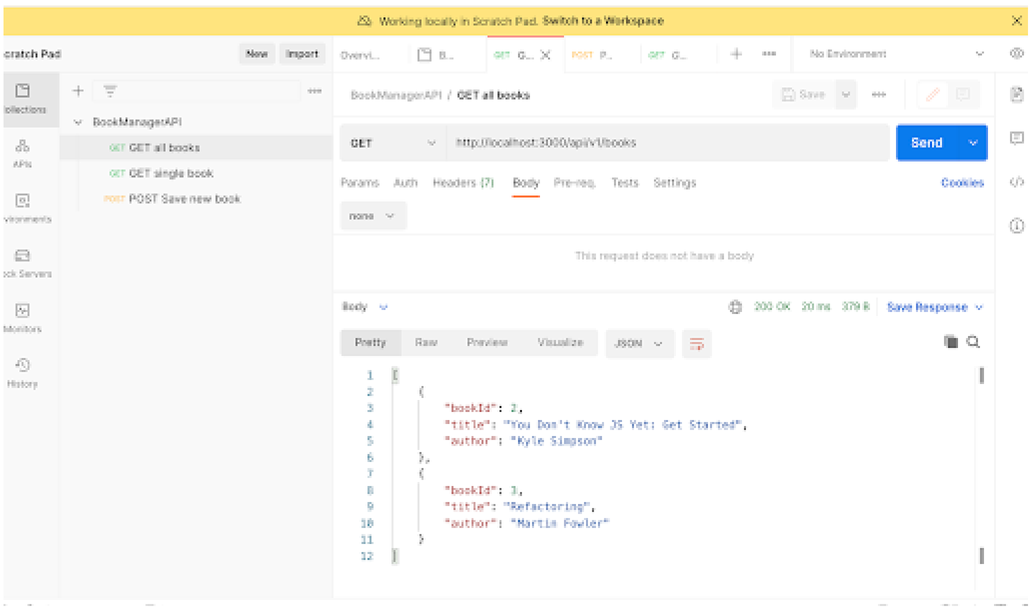
http://localhost:3000/api/v1/books...

```
{
    "bookId": 3,
    "title": "Refactoring",
    "author": "Martin Fowler"
}
```

201 Created 22 ms 299 B

```
{
    "bookId": 3,
    "title": "Refactoring",
    "author": "Martin Fowler"
}
```

🔎 Use the Postman Desktop Application to make a `GET` request to the http://localhost:3000/api/v1/books endpoint to get all books from your PostgreSQL database. You should be able to see something that looks similar to this. The books are being retrieved from the PostgreSQL database.



http://localhost:3000/api/v1/books

This request does not have a body

200 OK 20 ms 379 B

```
[
    {
        "bookId": 2,
        "title": "You Don't Know JS Yet: Get Started",
        "author": "Kyle Simpson"
    },
    {
        "bookId": 3,
        "title": "Refactoring",
        "author": "Martin Fowler"
    }
]
```

🔎 **Experiment!** Using the PostgreSQL Client - either in the terminal or through pgAdmin - add a new book into the `Books` table using an INSERT SQL statement. What do you think the GET request to the http://localhost:3000/api/v1/books endpoint will return?

💡 **Top Tip:** You need to conduct a new GET request to the endpoint.

🔎 **Experiment!** Stop your Minimalist Book Manager API application. Using the PostgreSQL Client (PostgreSQL Monitor) and ensuring you are using the `bookshop` database, run the following SQL statement. What do you expect to see? Will the books data persist or will the data disappear?

```
SELECT * FROM “Books”;
```