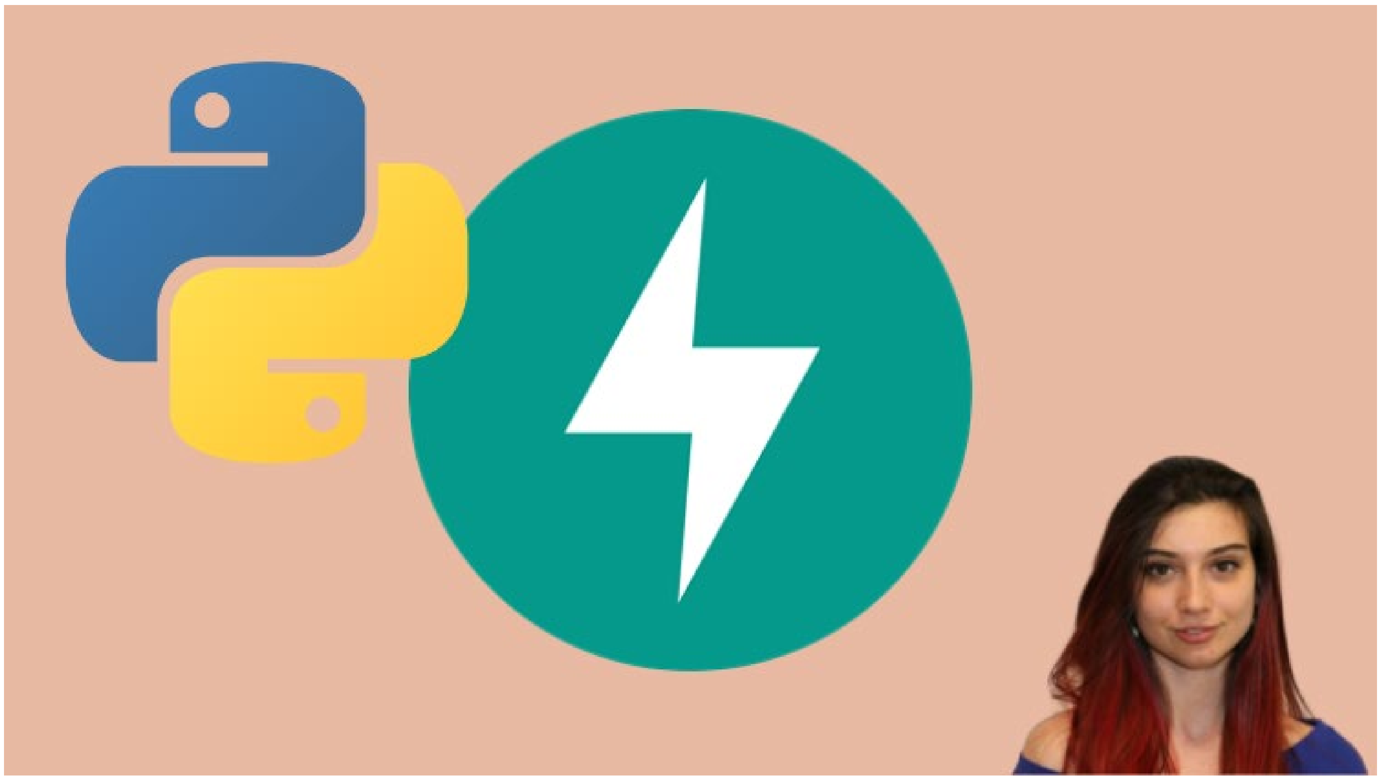
FastAPI REST – Part 2



CODE WITH FINESSE®

Ready to go?

Working with DB

TABLE OF CONTENTS

1. Installation

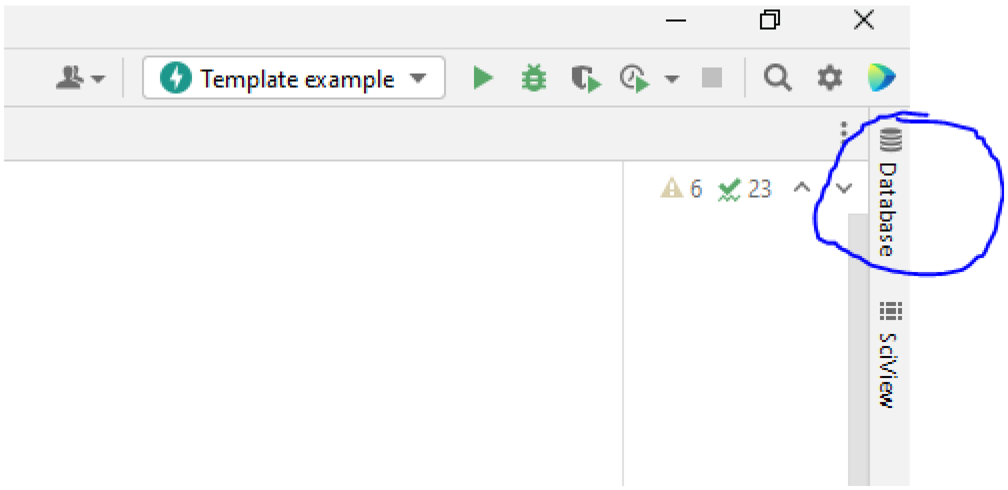
1.1 Introduction

This document will present helpful information on setting up PostgreSQL on windows and connecting the Flask application to the database. If you are using PyCharm professional, you will see how you can connect your database with the IDE and remove the need to use pgAdmin.
Also, you will see how to set up models and migrations, and we will discuss the need and usage of environment variables.

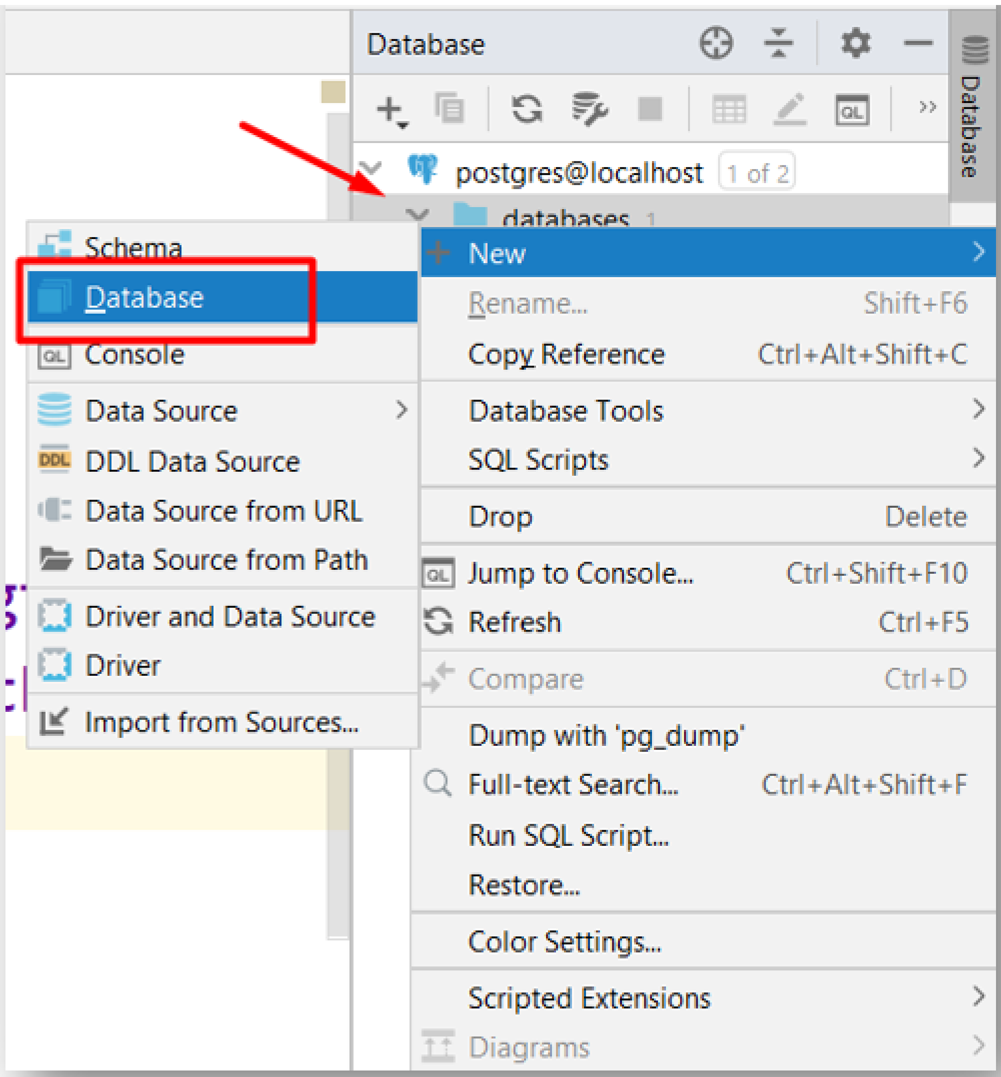
1.2 Install Postgres and DB config

If you are on windows, please follow this detailed tutorial here. It is essential to remember your password and the port that you will specify during the installation.

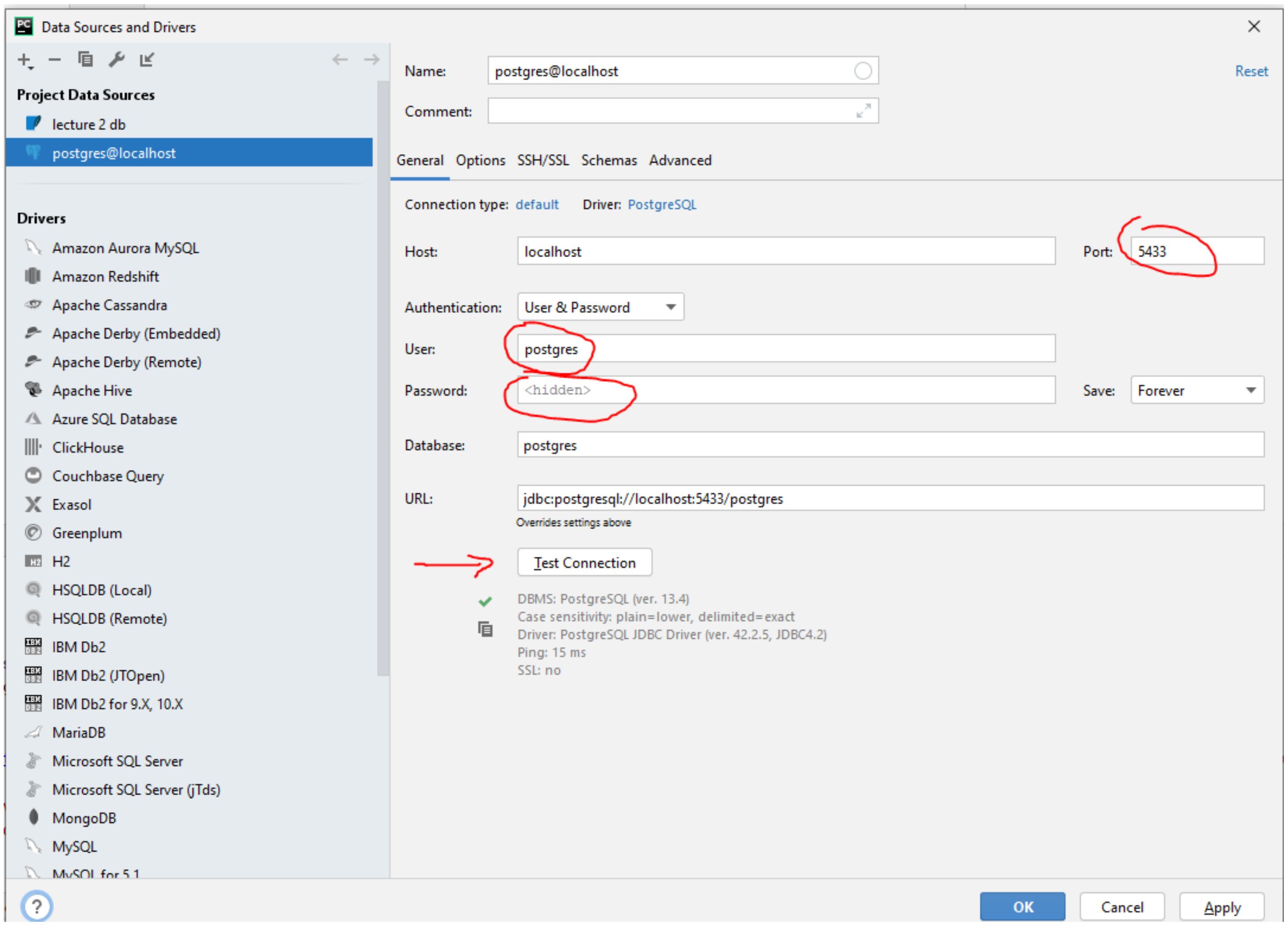
When you finish the installation, you may want to set up your PyCharm. Please note that this is working only on the professional edition.
Open your PyCharm and click Database on the right side.



Then click 'new' (the plus icon) and select a new data source **PostgreSQL**:



Next, we need to configure it.


Data Sources and Drivers

Project Data Sources
- lecture 2 db
- postgres@localhost

Drivers
- Amazon Aurora MySQL
- Amazon Redshift
- Apache Cassandra
- Apache Derby (Embedded)
- Apache Derby (Remote)
- Apache Hive
- Azure SQL Database
- ClickHouse
- Couchbase Query
- Exasol
- Greenplum
- H2
- HSQLDB (Local)
- HSQLDB (Remote)
- IBM Db2
- IBM Db2 (JTOpen)
- IBM Db2 for 9.X, 10.X
- MariaDB
- Microsoft SQL Server
- Microsoft SQL Server (jTds)
- MongoDB
- MySQL
- MySQL for 5.1

Name: postgres@localhost
Comment:

General | Options | SSH/SSL | Schemas | Advanced

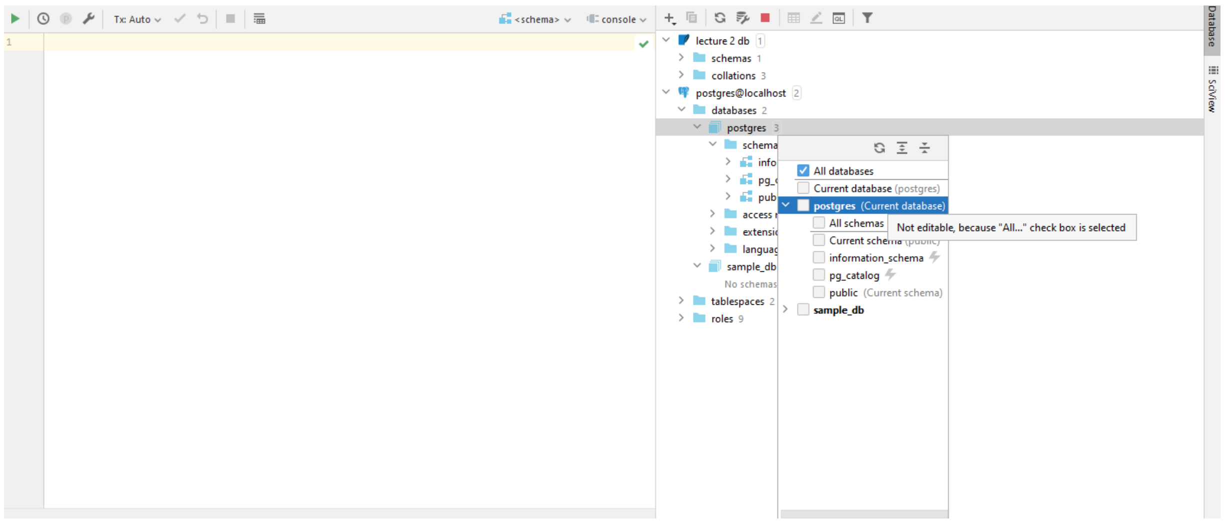
Connection type: default   Driver: PostgreSQL

Host: localhost   Port: 5433
Authentication: User & Password
User: postgres
Password: <hidden>   Save: Forever
Database: postgres
URL: jdbc:postgresql://localhost:5433/postgres
Overrides settings above

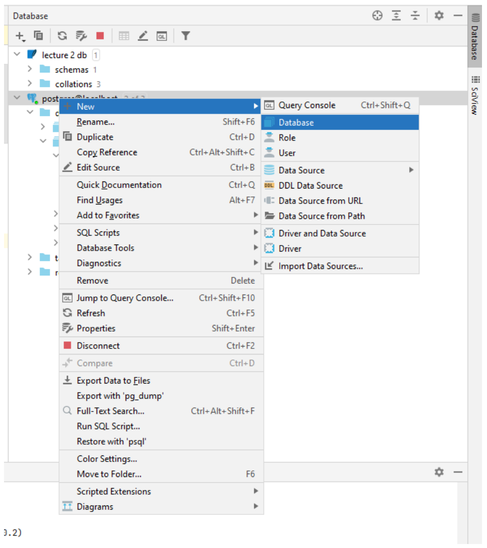
Test Connection

```
DBMS: PostgreSQL (ver. 13.4)
Case sensitivity: plain=lower, delimited=exact
Driver: PostgreSQL JDBC Driver (ver. 42.2.5, JDBC4.2)
Ping: 15 ms
SSL: no
```

You have to enter the data you configured when the installation for Postgres has been done. My port, in this case, is 5433. Your port might be the default one, 5432, if you haven't change it. The user should be Postgres. When you enter the port, user, and password you set when configuring the Postgres, you can test your connection by clicking the 'Test Connection' button. It may require installing some drivers, so install them and try again. If you see the green tick, then your PyCharm database source is configured.
If you do not have the professional edition, you can always use the alternative pgAdmin 4 (for Postgres 13). However, if you haven't specified anything different during the Postgres installation, it should be downloaded and set up by default (search in your start menu for pgAdmin).
If you do not see the databases (if you do not have any by default, you should have Postgres database), you can click on the dots and expand to see all databases and then on the next level to see all schemas like this (you should tick 'all' option):



And the last step will be to create a database. Right click on the Postgres source -> new -> database:



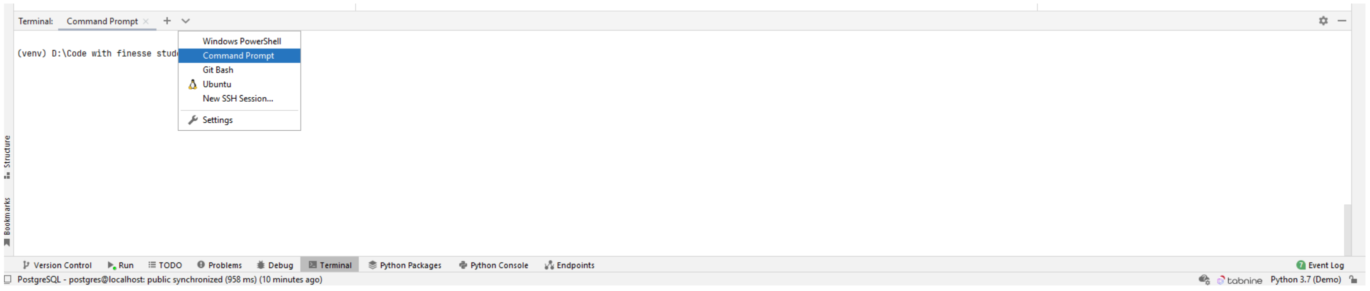
You can choose whatever name you like. For the purpose of the demo I will create the database called 'store'.

2. Setting up the App

2.1 Install all requirements

```
pip install fastapi uvicorn sqlalchemy databases asyncpg psycopg2 psycopg2-binary
```

Please, make sure you have activated your virtual environment and also choose the "Command Prompt" option from your PyCharm terminal:



```
(venv) D:\Code with finesse stud
```

PostgreSQL - postgres@localhost: public synchronized (958 ms) (10 minutes ago)

If you have configured your new project with PyCharm as FastAPI project, you do not need to install the fastapi library. The sqlalchemy and databases are required and used in our code directly as you will see in the example code below. The asyncpg psycopg2 psycopg2-binary libraries are helpers that allow us to connect with PostgreSQL. In most cases, you do not need to install psycopg2, but there are some edge cases for windows users where it is not installed, and you need to do it manually for your virtual environment.

Now you can place the following code in *main.py* file:

```
import databases
import sqlalchemy
from fastapi import FastAPI, Request

# Mine is this, please change for your credentials
# DATABASE_URL = "postgresql://postgres:ines123@localhost:5433/store"

DATABASE_URL = "postgresql://{place your db user}:{db password}@localhost:{your
port}/{db name}"


database = databases.Database(DATABASE_URL)

metadata = sqlalchemy.MetaData()

books = sqlalchemy.Table(
    "books",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column("title", sqlalchemy.String),
    sqlalchemy.Column("author", sqlalchemy.String),
)


engine = sqlalchemy.create_engine(DATABASE_URL)
metadata.create_all(engine)

app = FastAPI()


@app.on_event("startup")
async def startup():
    await database.connect()


@app.on_event("shutdown")
async def shutdown():
    await database.disconnect()


@app.get("/books/")
async def read_books():
    query = books.select()
    return await database.fetch_all(query)


@app.post("/books/")
async def create_book(request: Request):
    data = await request.json()
    query = books.insert().values(**data)
    last_record_id = await database.execute(query)
    return {"id": last_record_id}
```

The database URI used to configure the database engine, needs the database user and password we configured earlier for PyCharm and the port and the database name you have created. Please keep in mind that you need to change the values with you user, password port and db name:

DATABASE_URL = "postgresql://{your db user}:{your db password} @localhost:{port} /{db name}"
database = databases.Database(DATABASE_URL)

Here we are creating a model using the sqlalchemy.Table (*"books"* This is the argument which defines the name of the table that would be created, *metadata* is the object which we define from *sqlalchemy.MetaData()* and this allows us to use the SQLAlchemy Non-traditional mapping, all other arguments will describe the columns in our table: id, title and author.

In this case, we are creating the tables in the same Python file, but in production, you would probably want to create them with Alembic, integrated with migrations, etc. (we will see it in the next parts) Here, this section would run directly, right before starting your FastAPI application.

- Create an engine.
- Create all the tables from the metadata object.

```
engine = sqlalchemy.create_engine(DATABASE_URL)
metadata.create_all(engine)
```

Last but not least, we are creating the FastAPI object and using it as a decorator for some middleware functionality on_event "startup" and "shutdown" to connect and respectively disconnect from the database. These events are when you click the run command on PyCharm (starting the server) and when you terminate the process (ctrl + C to kill the server)

@app.get and *@app.post* are defining the request methods for URL /books/. The first one is selecting all the existing books and returns them, the second one is creating a new record using the *insert()* method. Note that because we are *awaiting* the request.json() and also each execution of the database. This is the charm of using asynchronous programming – we make it faster, but we need to be aware of the coroutines. You can refresh your knowledge and gain understanding here.

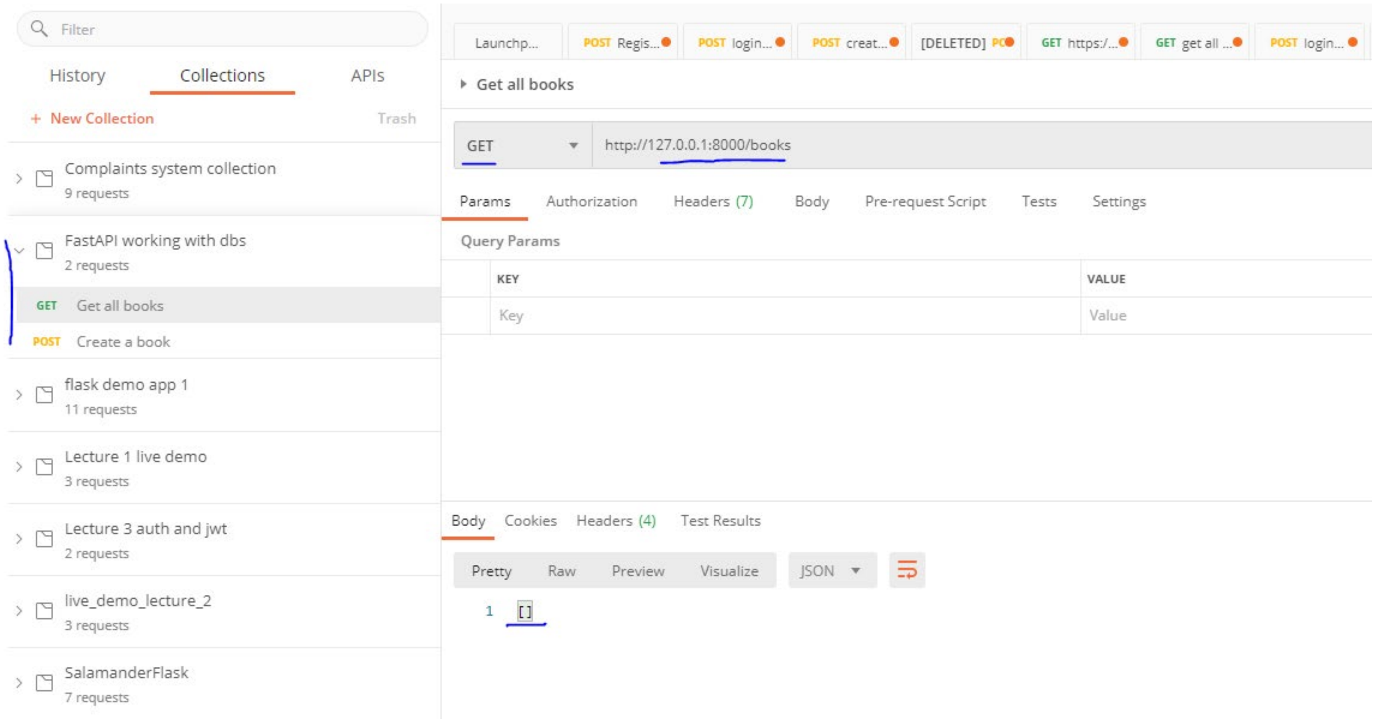
3. Test it with Postman

3.1 Set up the GET request

Create a new collection in Postman and add two requests in it.
The first one should be "Get all books". Change the method to GET and post the url http://127.0.0.1:8000/books ensure there are no leading/trailing spaces. Start your application (In PyCharm the green button, top right, or you can write in the terminal from your project root:

```
uvicorn main:app --reload
```

)
Then in Postman click the "send" button next to the URL.



```
GET http://127.0.0.1:8000/books
```

```
[]
```

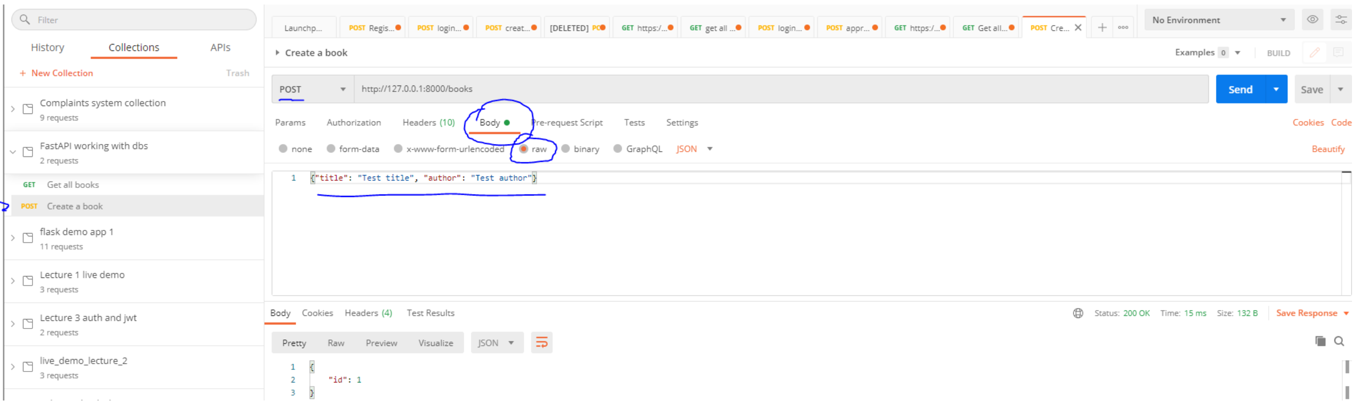
As a result, you will see empty list, because we do not have anything yet in our table.

3.2 Set up the POST request

Here it is really import not to miss a step, because you will face a lot of difficulties

1 Change the method url to POST
2 Enter again the same URL
3 Click “body” an select “raw”, then enter this JSON data (be aware that double “ are **must**)

```
{"title": "Test title", "author": "Test author"}
```



Create a book

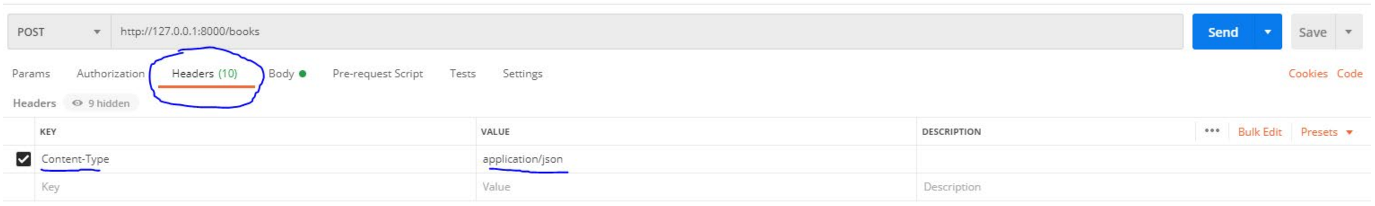
POST http://127.0.0.1:8000/books

```
{"title": "Test title", "author": "Test author"}
```

Status: 200 OK Time: 15 ms Size: 132 B

```
{
    "id": 1
}
```

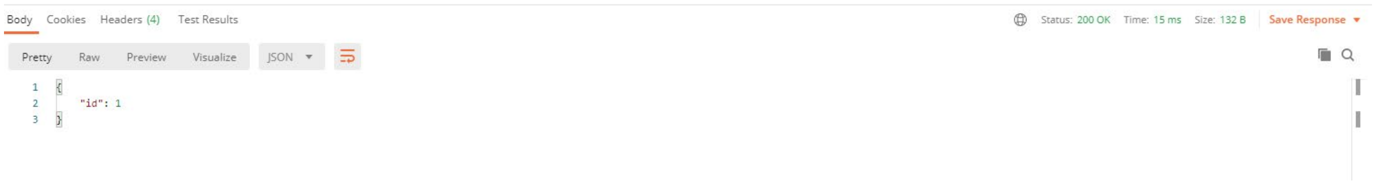
4 Click “Headers” and add the following:



POST http://127.0.0.1:8000/books

Headers (10)

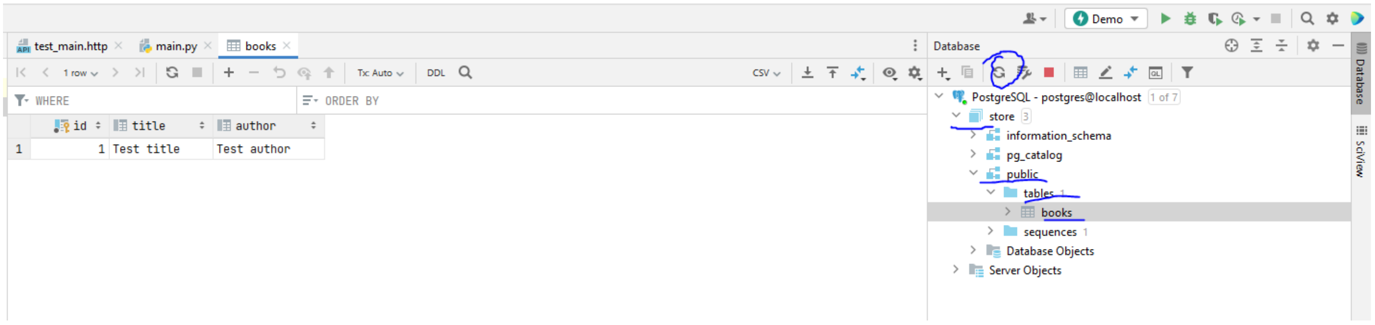
KEY	VALUE	DESCRIPTION
Content-Type	application/json	



Status: 200 OK Time: 15 ms Size: 132 B

```
{
    "id": 1
}
```

When you click “Send” it will go directly to our *create_book()* function and it will save the record in the database.
If you click on your table, you will be able to see the record. Please, refresh, if you are not able to enter.



	id	title	author
1	1	Test title	Test author

PostgreSQL - postgres@localhost
store
information_schema
pg_catalog
public
tables 1
books
sequences 1
Database Objects
Server Objects

4. Migrations

4.1 Introduction

As we have snapshots of the code state (using some version control system), we need to have a similar way to save the database state – in some cases, we may want to revert to the previous form. This, however is achieved by using migrations, which currently we do not have.

4.2 Configure alembic

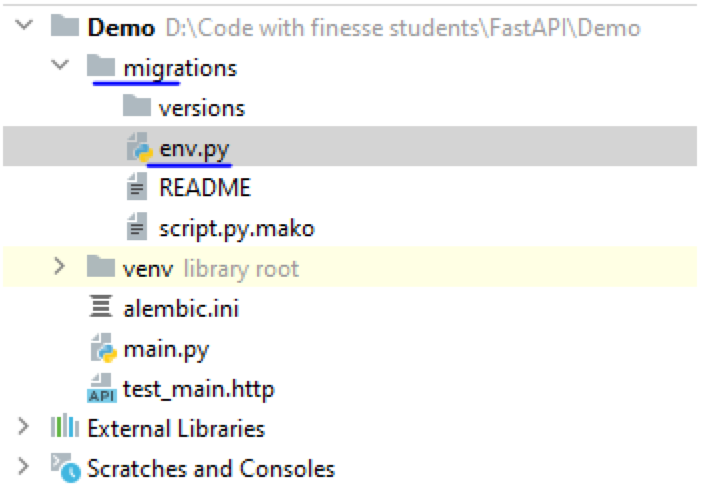
Our first step will be to install alembic as our migration manager:

```
pip install alembic
```

Then we need to create a related folders and files:

```
alembic init migrations
```

The migrations directory + respective files will be created automatically:



```
Demo D:\Code with finesse students\FastAPI\Demo
  migrations
    versions
    env.py
    README
    script.py.mako
  venv library root
  alembic.ini
  main.py
  test_main.http
External Libraries
Scratches and Consoles
```

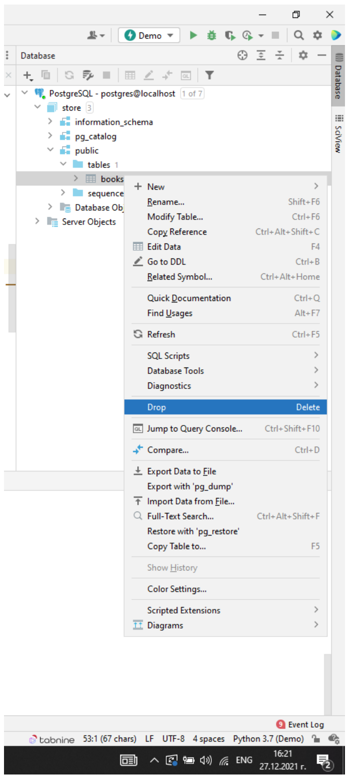
We need to configure the env.py file:

```
from main import metadata

target_metadata = metadata
```

Please, delete ***target_metadata=None*** **(if it is presented a couple of rows below)**
In the **alembic.ini** file change the
sqlalchemy.url = postgresql://{user}:{password}@localhost:{port}/{db_name}

where you have to replace them with your credentials for user, password, port and db_name (you can copy-paste the string from main.py file).

Then we need to delete the current table (so that we can be sure it is working -> right click -> drop):



Last, we need to **remove** the following lines of code in our *main.py* file:

```
engine = sqlalchemy.create_engine(DATABASE_URL)
metadata.create_all(engine)
```

4.3 Migrations

Now, having the initial setup, we need to do our first migration:

```
alembic revision --autogenerate -m "initial"
```
(The message could be anything)

You will have something like this:

```
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.ddl.postgresql] Detected sequence named 'books_id_seq' as owned by integer column 'books(id)', assuming SERIAL and omitting
```

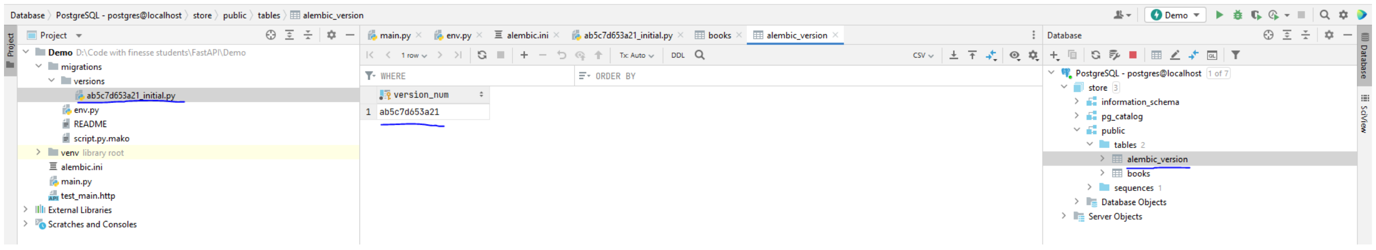
Generating D:\Code with finesse students\FastAPI\Demo\migrations\versions\b5a232ffbd2c_initial.py ... done

Then if you take a look at migrations/versions you will see the migration (the hash number of the .py will be different for you):

Then we need to upgrade to the latest version:

```
alembic upgrade head
```

Now the db has another table called “alembic_version” where the latest migration is applied for the database state:



	version_num
1	ab5c7d653a21

5. Relations

Before we get any further this could be a good refreshener/ or starting tutorial for relationships.

1.1 One-to-many

For the last part, we are going to create another model with a relationship 'Reader can have multiple books' We are going to adjust our models:

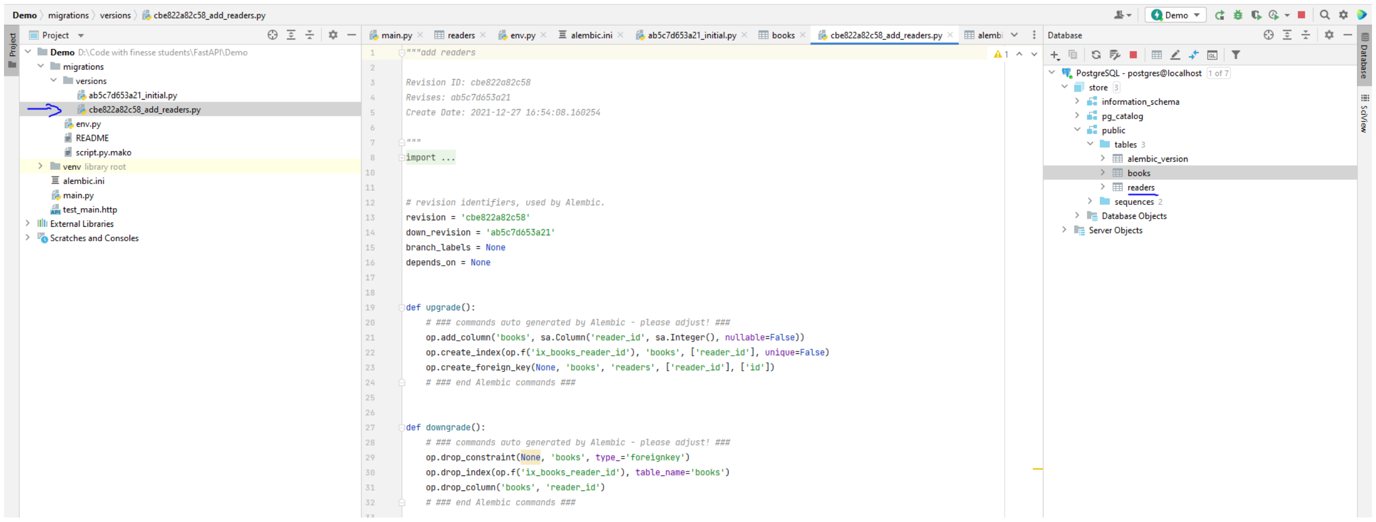
```
books = sqlalchemy.Table(
    "books",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column("title", sqlalchemy.String),
    sqlalchemy.Column("author", sqlalchemy.String),
    sqlalchemy.Column("reader_id", sqlalchemy.ForeignKey("readers.id"),
nullable=False, index=True),
)

readers = sqlalchemy.Table(
    "readers",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column("first_name", sqlalchemy.String),
    sqlalchemy.Column("last_name", sqlalchemy.String),
)
```

Here we defined a new table "readers". Also, we added a new column, using the ForeignKey from sqlalchemy:

sqlalchemy.Column("reader_id", sqlalchemy.ForeignKey("readers.id"), nullable=False, index=True)

This now means we added a column which will reference an existing row of readers table, by using the id. We need to do our migrations as well:
`alembic revision --autogenerate -m "add readers"`
and then to update the db:
`alembic upgrade head`

Now the migration is generated and the table is presented in the "store" db:



```
"""add readers

Revision ID: cbe822a82c58
Revises: ab5c7d653a21
Create Date: 2021-12-27 16:54:08.160254

"""
import ...


# revision identifiers, used by Alembic.
revision = 'cbe822a82c58'
down_revision = 'ab5c7d653a21'
branch_labels = None
depends_on = None


def upgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.add_column('books', sa.Column('reader_id', sa.Integer(), nullable=False))
    op.create_index(op.f('ix_books_reader_id'), 'books', ['reader_id'], unique=False)
    op.create_foreign_key(None, 'books', 'readers', ['reader_id'], ['id'])
    # ### end Alembic commands ###


def downgrade():
    # ### commands auto generated by Alembic - please adjust! ###
    op.drop_constraint(None, 'books', type_='foreignkey')
    op.drop_index(op.f('ix_books_reader_id'), table_name='books')
    op.drop_column('books', 'reader_id')
    # ### end Alembic commands ###
```

Now if we want to test if it is working, we need to expose another endpoint, responsible of creating the reader (because we want to create a book, we need to place existing reader id):

```
@app.post("/readers/")
async def create_reader(request: Request):
    data = await request.json()
    query = readers.insert().values(**data)
    last_record_id = await database.execute(query)
    return {"id": last_record_id}
```

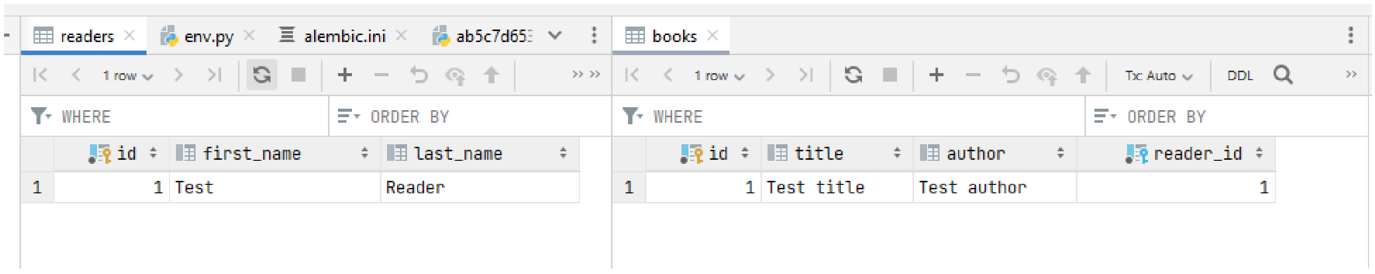
Let's go to Postman and configure another POST request to http://127.0.0.1:8000/readers raw body:

```
{"first_name": "Test", "last_name": "Reader"}
```

(Do not forget to add Content-Type header).

Now we can return to POST /books/ request and add "reader_id": 1 in the body:

```
{"title": "Test title", "author": "Test author", "reader_id": 1}
```



	id	first_name	last_name
1	1	Test	Reader

	id	title	author	reader_id
1	1	Test title	Test author	1

1.2 Many-to-many

It does not sound right – one reader to have multiple books, but the book could be to only one reader? It is more logical to have many readers to many books and many books to many readers.
Let's change the models here so we can mirror the many-to-many behavior.

First, we need to drop the tables. Then we will remove the ForeignKey row and add a new table:

```
books = sqlalchemy.Table(
    "books",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column("title", sqlalchemy.String),
    sqlalchemy.Column("author", sqlalchemy.String),
)

readers = sqlalchemy.Table(
    "readers",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column("first_name", sqlalchemy.String),
    sqlalchemy.Column("last_name", sqlalchemy.String),
)

readers_books = sqlalchemy.Table(
    "readers_books",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_key=True),
    sqlalchemy. Column("book_id", sqlalchemy.ForeignKey("books.id"),
nullable=False, index=True),
    sqlalchemy.Column("reader_id", sqlalchemy.ForeignKey("readers.id"),
nullable=False, index=True),
)
```

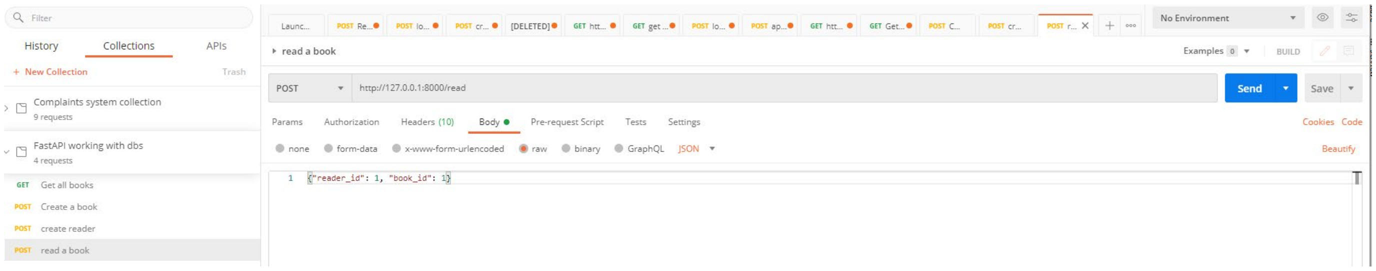
Our readers_books table is actually a junction table (with these we are able to mirror many-to-many behavior)
Then we need to do a revision and update with alembic.
Remove the "reader_id" from Postman POST request on /books/

Now we need to expose a new endpoint, which will be responsible to track who-what has read:

```
@app.post("/read/")
async def read_book(request: Request):
    data = await request.json()
    query = readers_books.insert().values(**data)
    last_record_id = await database.execute(query)
    return {"id": last_record_id}
```

We need to send requests one more time for "/books/" and "/readers/".

Last, we will create a new request:

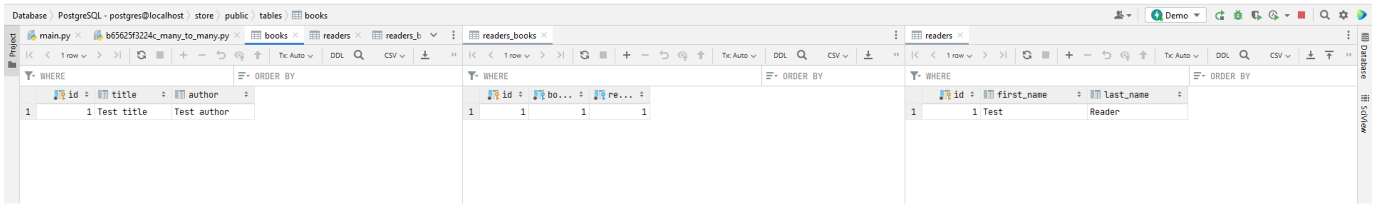


read a book

POST http://127.0.0.1:8000/read

```
{"reader_id": 1, "book_id": 1}
```

Once sent, we can see the tree tables state:



books

	id	title	author
1	1	Test title	Test author

readers_books

	id	bo…	re…
1	1	1	1

readers

	id	first_name	last_name
1	1	Test	Reader

6. Bonus – small refactor - .env

Remember how we left the connection string unprotected?

```
DATABASE_URL = "postgresql://postgres:ines123@localhost:5433/store"
```

This is considered terrible practice because if you submit this file to the Github Repo, it means everyone has access to your DB credentials if the repo is public. However, even though the repo is private, it is still dangerous to do it (submitting keys and credentials to the repo) because a breach of the repo can happen, and the credentials can leak.

One alternative to do it safely is to use environment variables. First, create a file called *.env* in your project root directory. Then assure this file is listed in the *.gitignore* file. Otherwise, it is pointless.
Now we need to install a valuable package to work with environment variables:

```
pip install python-decouple
```

Then we need to add the two variables in the *.env* file:

```
DB_USER=postgres
DB_PASSWORD=ines123
```

Please, make sure that you have replaced the values with your Postgres user and password.

Last but not least we need to use it in the code like this:

```
import databases
import sqlalchemy
from fastapi import FastAPI, Request
from decouple import config


DATABASE_URL =
f"postgresql://{config('DB_USER')}:{config('DB_PASSWORD')}@localhost:5433/store"

database = databases.Database(DATABASE_URL)


….
```

Start the application again and make sure it is working correctly (send a book create request in Postman), before submit the project to the repo.
Try to make the db name and db port as environment variables and use them as well like we did with the password and the user.