# SQLAlchemy, Postgres and migrations in Flask

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.

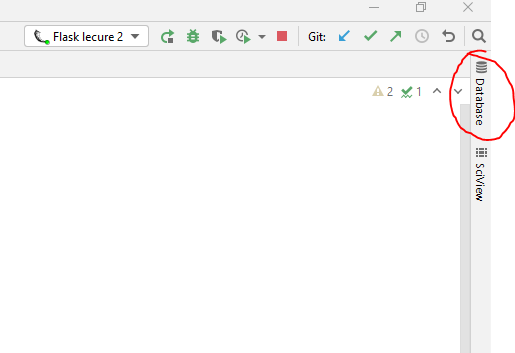
Also, you will see how to set up models and migrations, and we will discuss the need and usage of environment variables.

## Install Postgres and DB config

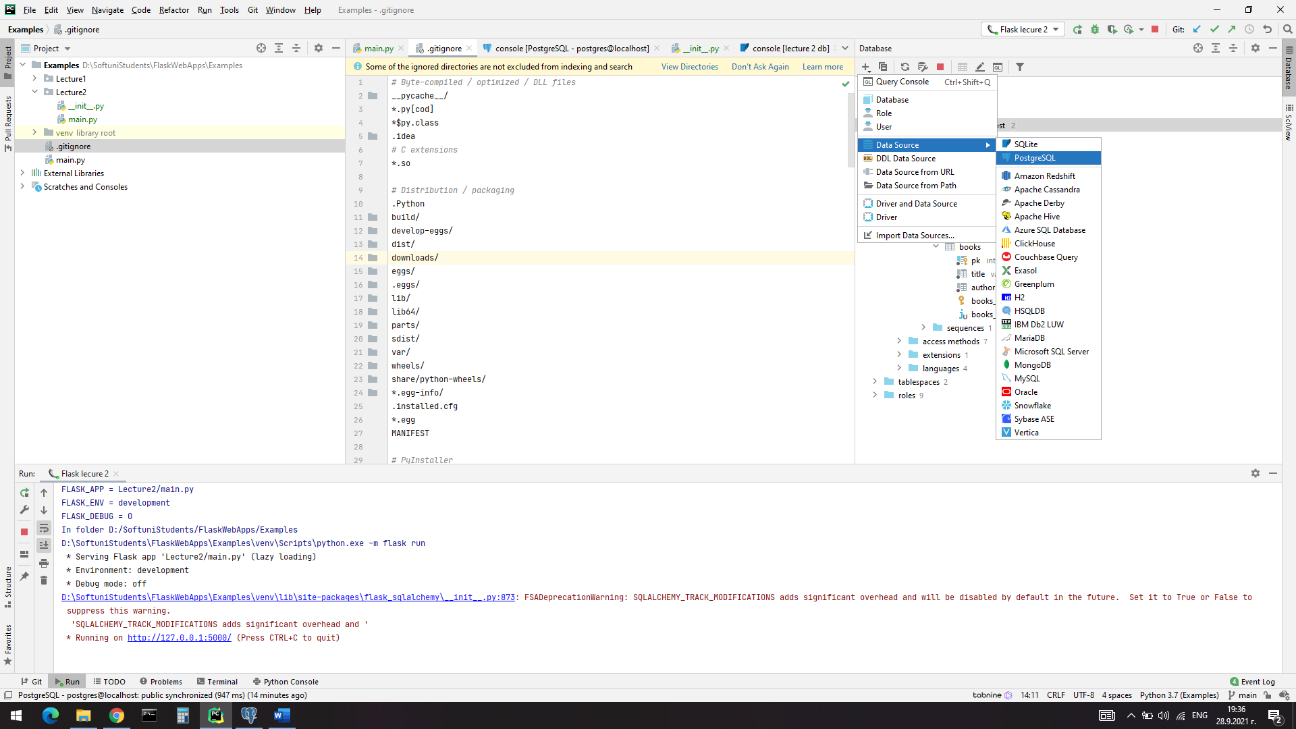
If you are on windows, please follow this detailed tutorial [here](https://www.guru99.com/download-install-postgresql.html). It is essential to remember your password and the port that you will specify during the installation.

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

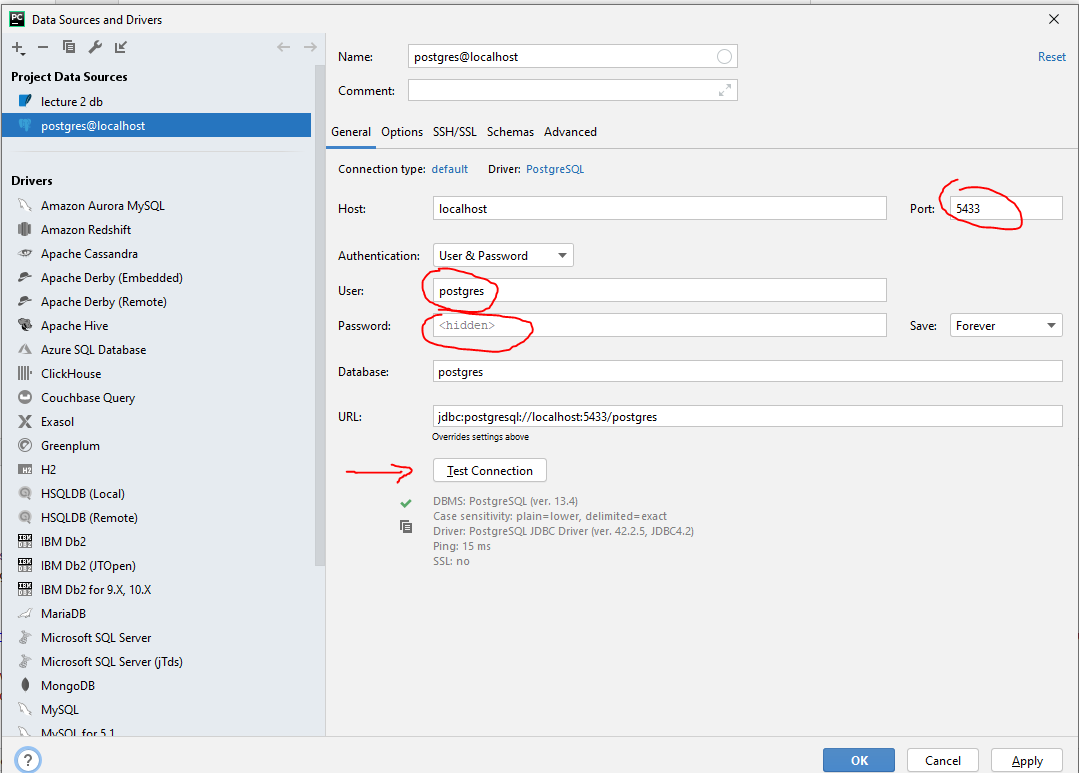
Open your PyCharm and click Database on the right side.



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



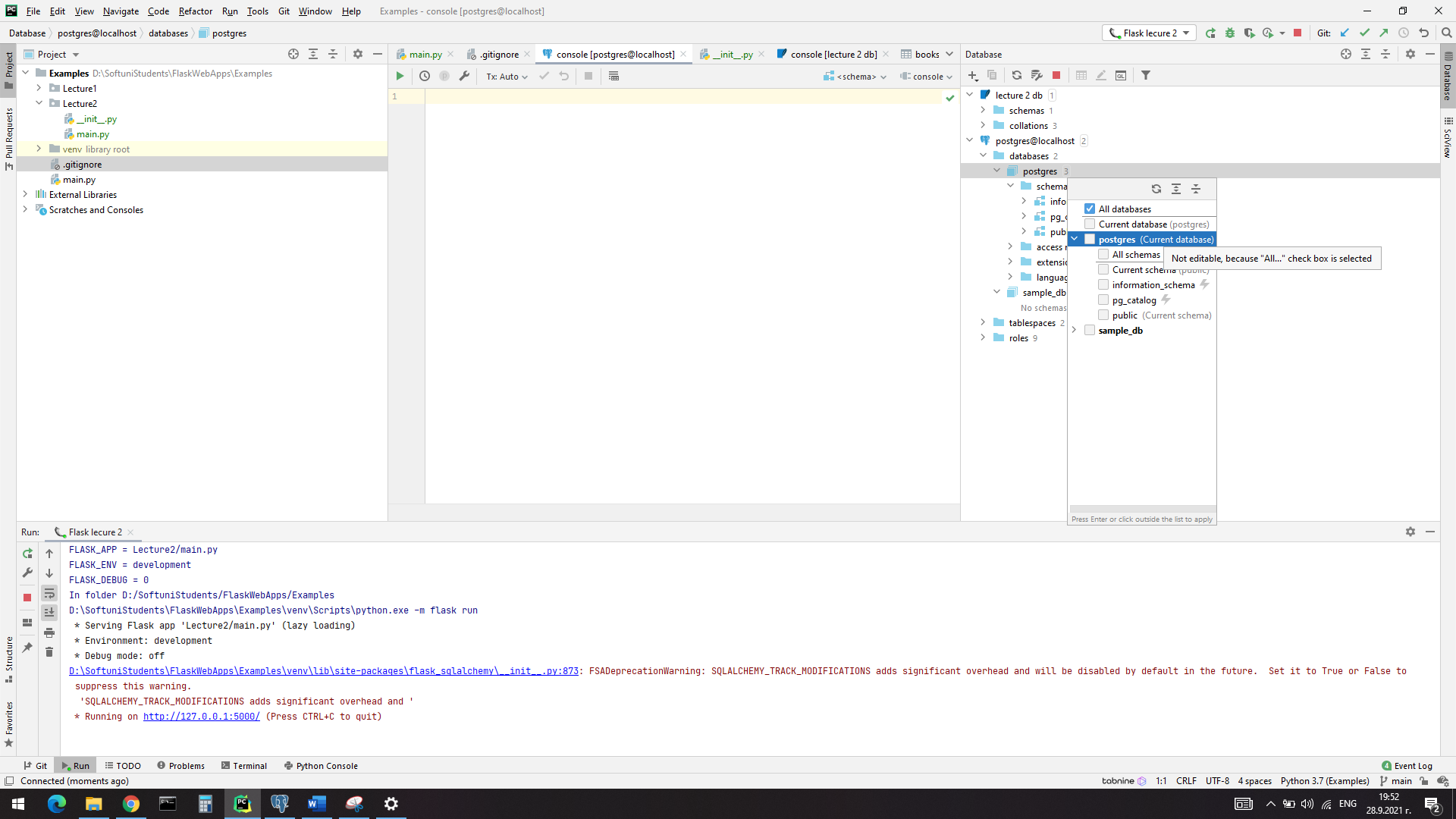
Next, we need to configure it.



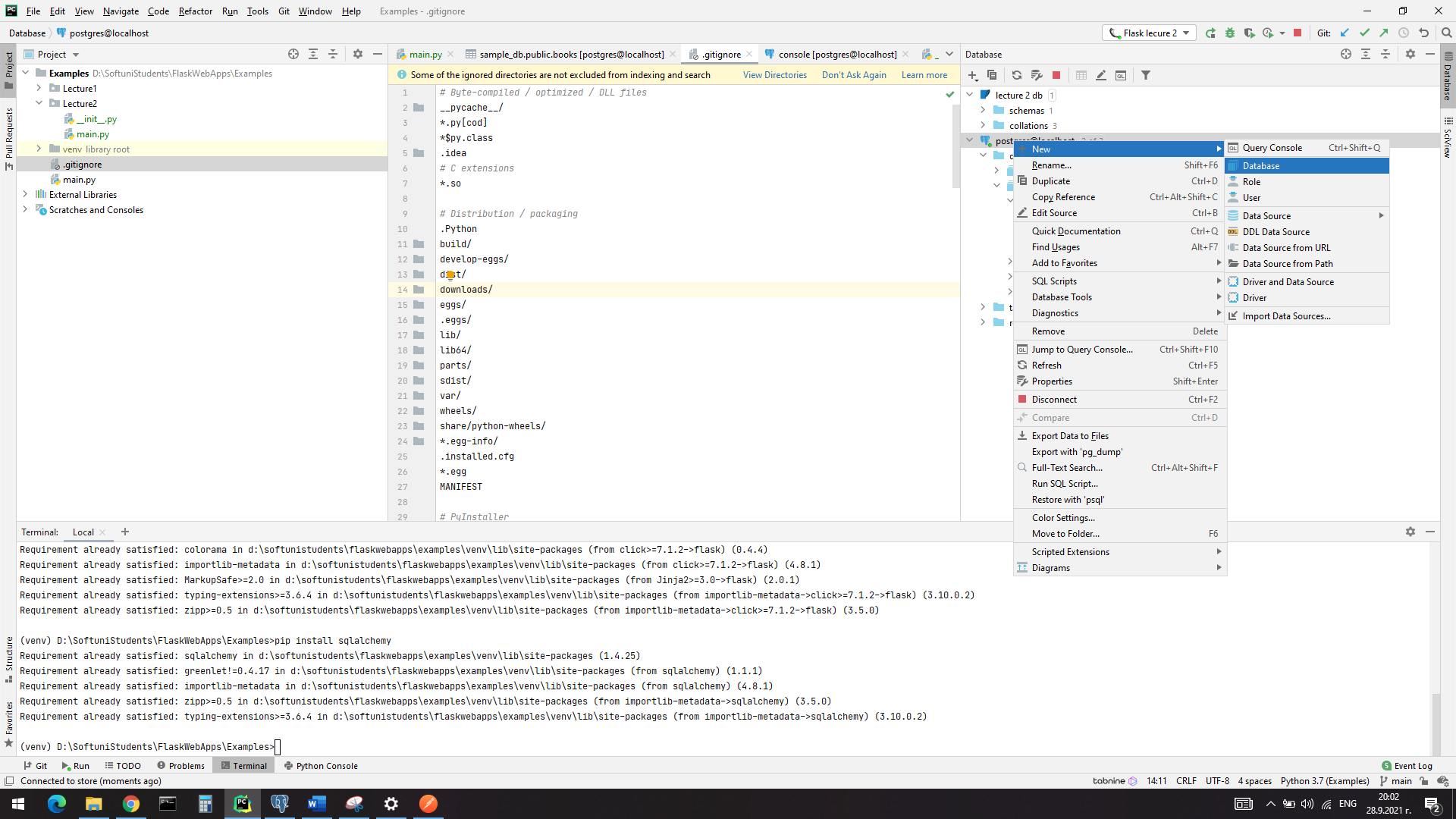
You have to enter the data you configured when the installation for Postgres has been done. My port, in this case, is 5433. Your port might be the default one, 5432, if you haven’t change it. The user should be Postgres. When you enter the port, user, and password you set when configuring the Postgres, 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’.

## Setting up the App

Now we need to install the requirements:

pip install flask flask-restful flask\_sqlalchemy psycopg2-binary

The last package gives us the chance to work with Postgres. Flask SQLAlchemy docs – [here](https://flask-sqlalchemy.palletsprojects.com/en/2.x/quickstart/#a-minimal-application).

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

**from** flask **import** Flask, request  
**from** flask\_sqlalchemy **import** SQLAlchemy  
**from** flask\_restful **import** Resource, Api  
  
app = Flask(\_\_name\_\_)  
  
app.config[**'SQLALCHEMY\_DATABASE\_URI'**] = **f'postgresql://postgres:ines123@localhost:5433/store'**db = SQLAlchemy(app)  
api = Api(app)  
  
  
**class** BookModel(db.Model):  
 \_\_tablename\_\_ = **'books'** pk = db.Column(db.Integer, primary\_key=**True**)  
 title = db.Column(db.String, nullable=**False**)  
 author = db.Column(db.String, nullable=**False**)  
  
 **def** \_\_repr\_\_(self):  
 **return f"<{**self.pk**}> {**self.title**} from {**self.author**}"  
  
 def** as\_dict(self):  
 **return** {c.name: getattr(self, c.name) **for** c **in** self.\_\_table\_\_.columns}  
  
  
**class** Books(Resource):  
 **def** post(self):  
 data = request.get\_json()  
 new\_book = BookModel(\*\*data)  
 db.session.add(new\_book)  
 db.session.commit()  
 **return** new\_book.as\_dict()  
  
  
db.create\_all()  
api.add\_resource(Books, **"/"**)  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 app.run(debug=**True**)

Here we are creating a model using the flask SQLAlchemy (by inheriting db.Model). Later on, that would be an effortless and efficient way to work with the ORM. We are setting up a single endpoint that can create books. To communicate with the database, we need to configure the connection string. This is happening by this row here:

app.config[**'SQLALCHEMY\_DATABASE\_URI'**] = **f'postgresql://postgres:ines123@localhost:5433/store'**

Keep in mind we are going to remove the password and the user from the string later on.

Other important thing is this row:

db.create\_all()

It is taking care to create all tables (in our case, we have just one ‘books’, defined by the BookModel class).

The resource class here is responsible for getting the data from the request and creating an instance object from class BookModel with this data. After that, with the session’s help, we need to add it and commit (please note if you do not commit, the changes would not be presented in the database). In the future, we will see how to apply the single responsibility principle and remove the session’s commit from the post function.

## Test it with Postman

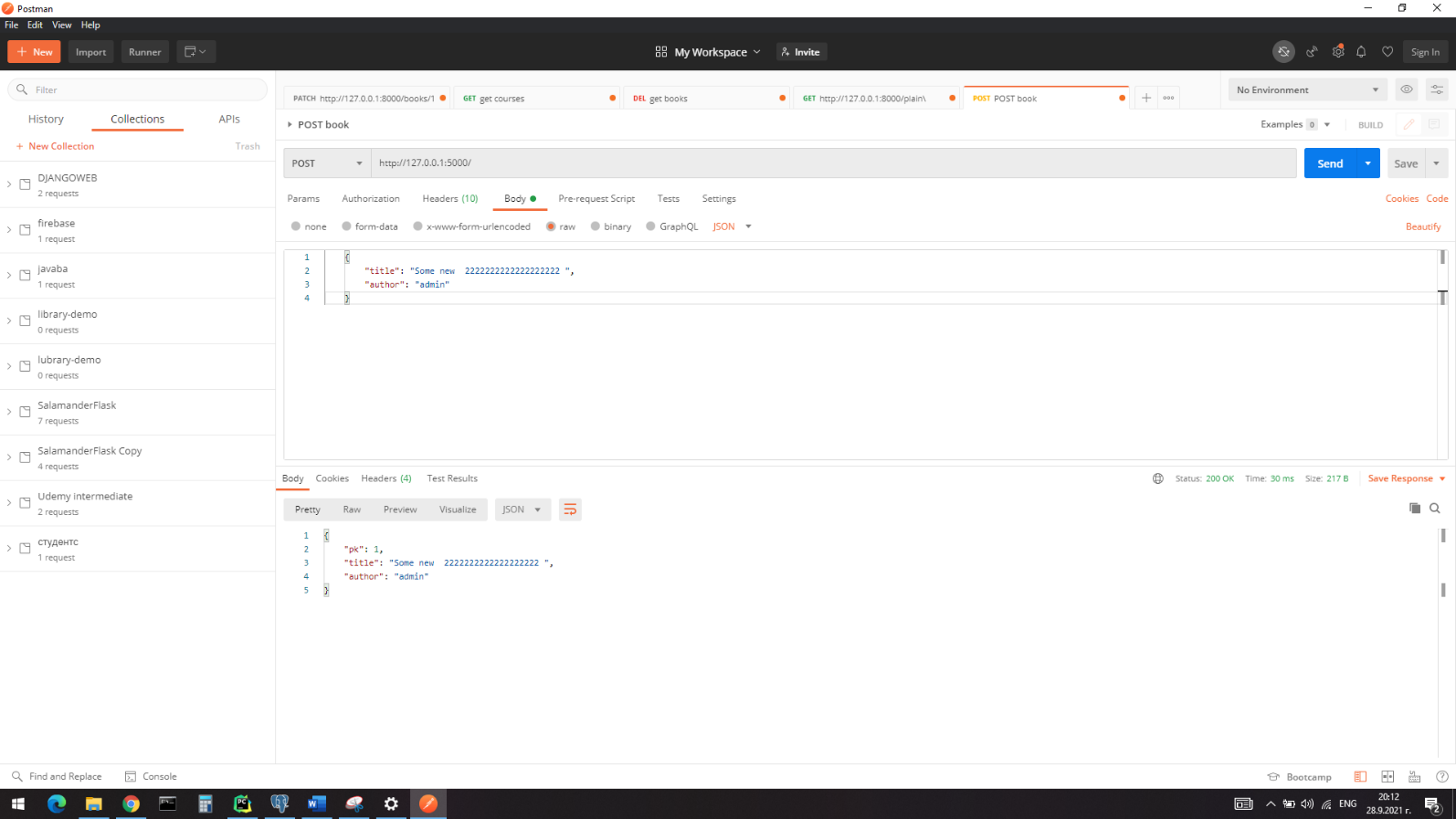
Launch the postman app and make a POST request to <http://127.0.0.1:5000/> with the following body (you should add Content-Type header (application/json) for this request so that the body is transformed to json format):

    {

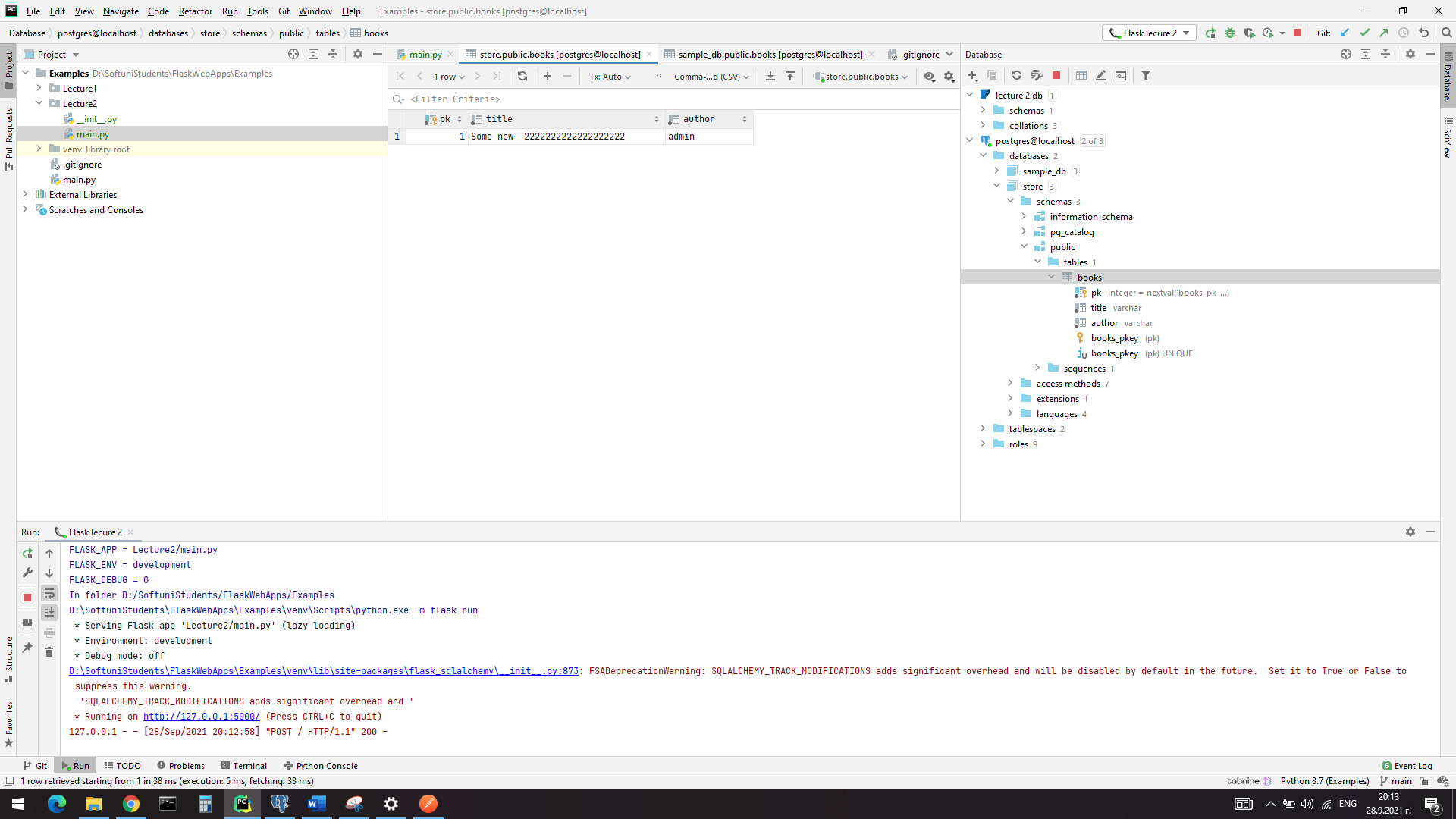
        "title": "Some new  2222222222222222222 ",

        "author": "admin"

    }

If everything is correct you should get the response for the newly created book:  


And also, you can go and check the database in PyCharm and assure there is a record:

****

## Migrations

Now is the right time to set up flask-migrate. It would be beneficial to have migrations. Otherwise, at some point, the application and the database state would become a mess and irreversible.

pip install Flask-Migrate==2.5.2

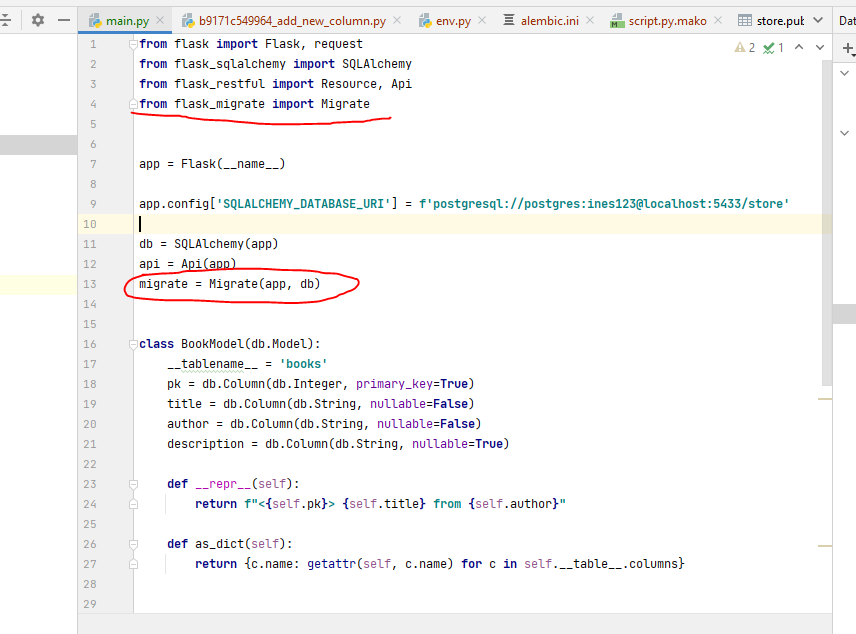
Now the fun part comes in.

If you are on windows, you should provide the "FLASK\_APP" environment variable. The following command does this:

set FLASK\_APP=./main.py

(If you are on Linux, then the commend should be export instead of set).

Then you need to import **Migrate** from **flask\_migrate** and connect the app and the database object from **SQLAlchemy**.



Then we need to initialize a migration repository with the following command:

flask db init

This will add a migrations folder to your application. The contents of this folder need to be added to version control along with your other source files.

You can then generate an initial migration:

flask db migrate -m "Initial migration."

The message could be anything in the brackets.

For demo purposes, you might notice I added a column in the BookModel class, called description. With each change (like adding, removing columns, or making it nullable or not), you must create a migration. Of course, a couple of changes could take place in a migration. So you can try adding a new column and say:

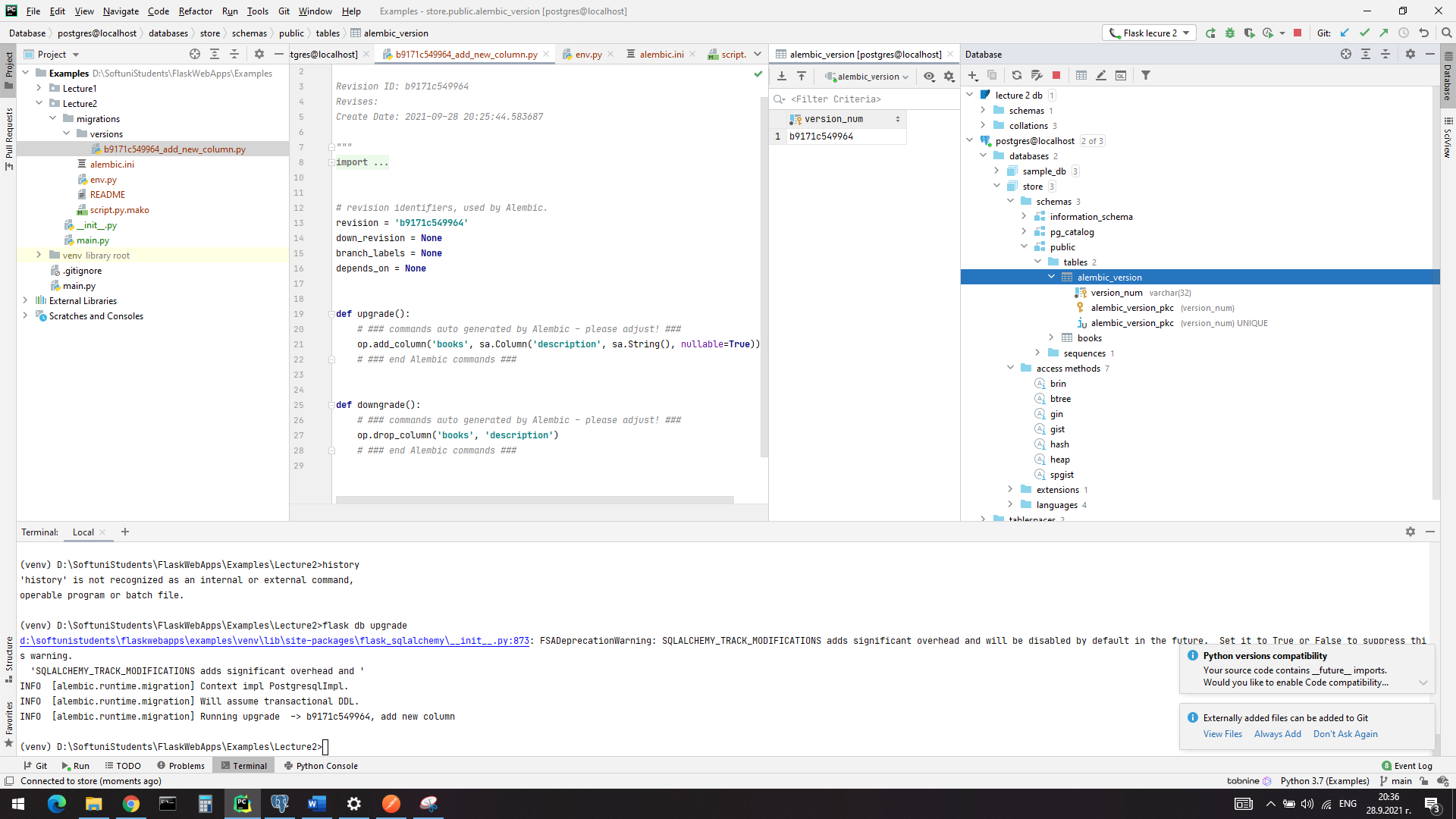
flask db migrate -m "add new column"

And then:

flask db upgrade

A couple of things happened. First, we have an official migration. Second, we have a new table in the database called alembic\_version with a single column version\_num, which keeps track of the last upgraded version of the database.

.



**NB!  
Debugging problem: If you run the commands, but no migration actually appear, please follow these steps:**1. Delete the migrations folder  
2. Delete the books table  
3. Remove db.create\_all()  
4. Repeat the three steps  
 - flask d binit  
 - flask db migrate -m “Initial migration”  
 - flask db upgrade

## Relations

Before we get any further [this](https://docs.sqlalchemy.org/en/14/orm/basic_relationships.html#one-to-one) could be a good refreshener/ or starting tutorial for relationships. For the last part today, we are going to create another model with a relationship ‘Reader can have multiple books’.  
We are going to adjust our models:

**class** BookModel(db.Model):  
 \_\_tablename\_\_ = **'books'** pk = db.Column(db.Integer, primary\_key=**True**)  
 title = db.Column(db.String, nullable=**False**)  
 author = db.Column(db.String, nullable=**False**)  
 description = db.Column(db.String, nullable=**True**)  
 some\_column = db.Column(db.String, nullable=**False**, default=**"some"**, server\_default=**"ok"**)  
 reader\_pk = db.Column(db.Integer, db.ForeignKey(**'readers.pk'**))  
 reader = db.relationship(**'ReaderModel'**)  
  
 **def** \_\_repr\_\_(self):  
 **return f"<{**self.pk**}> {**self.title**} from {**self.author**}"  
  
 def** as\_dict(self):  
 **return** {c.name: getattr(self, c.name) **for** c **in** self.\_\_table\_\_.columns}  
  
  
**class** ReaderModel(db.Model):  
 \_\_tablename\_\_ = **'readers'** pk = db.Column(db.Integer, primary\_key=**True**)  
 first\_name = db.Column(db.String, nullable=**False**)  
 last\_name = db.Column(db.String, nullable=**False**)  
 books = db.relationship(**"BookModel"**, backref=**"book"**, lazy=**'dynamic'**)

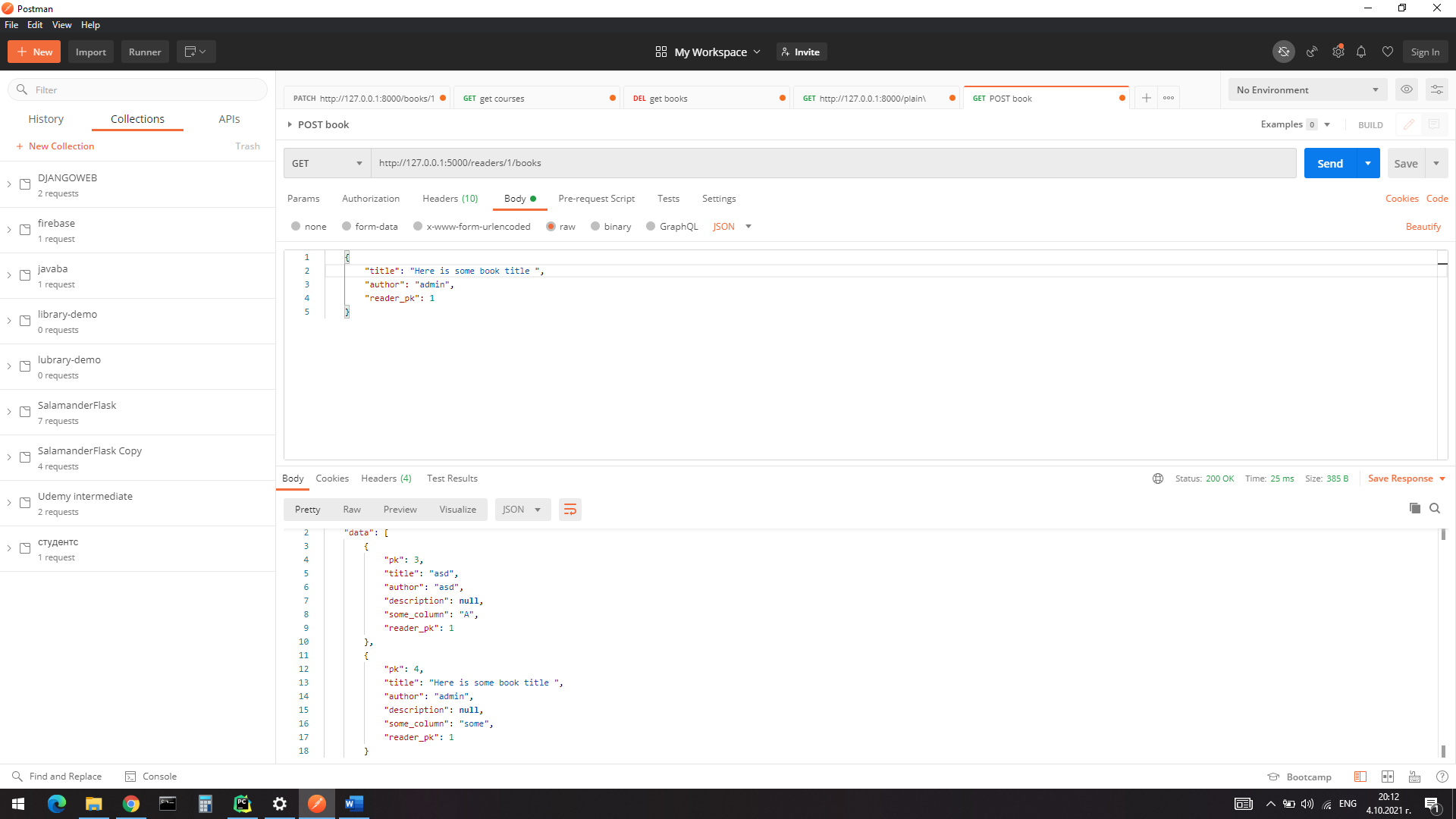
We user ForeignKey to describe the relationship to the other table and what would be the key. With db.relationship we are able to specify that we want to fetch the whole object (Reader) related to the specific row if needed. We need to migrate:  
flask db migrate -m "Create a relationship and readers table "

flask db upgrade

In the ReaderModel we specify the relationship with backref=**"book"**, lazy=**'dynamic’** It emits a SELECT statement when loading. You can see that it returns a sqlalchemy object instead of the book objects. This is basically a SELECT statement for the book class, hence, it’s still waiting for us to call a method on it. In order to return the list of books, therefore, we need to call **.all(** ) after readers. Here is an example of how you can use both to query whatever you need:

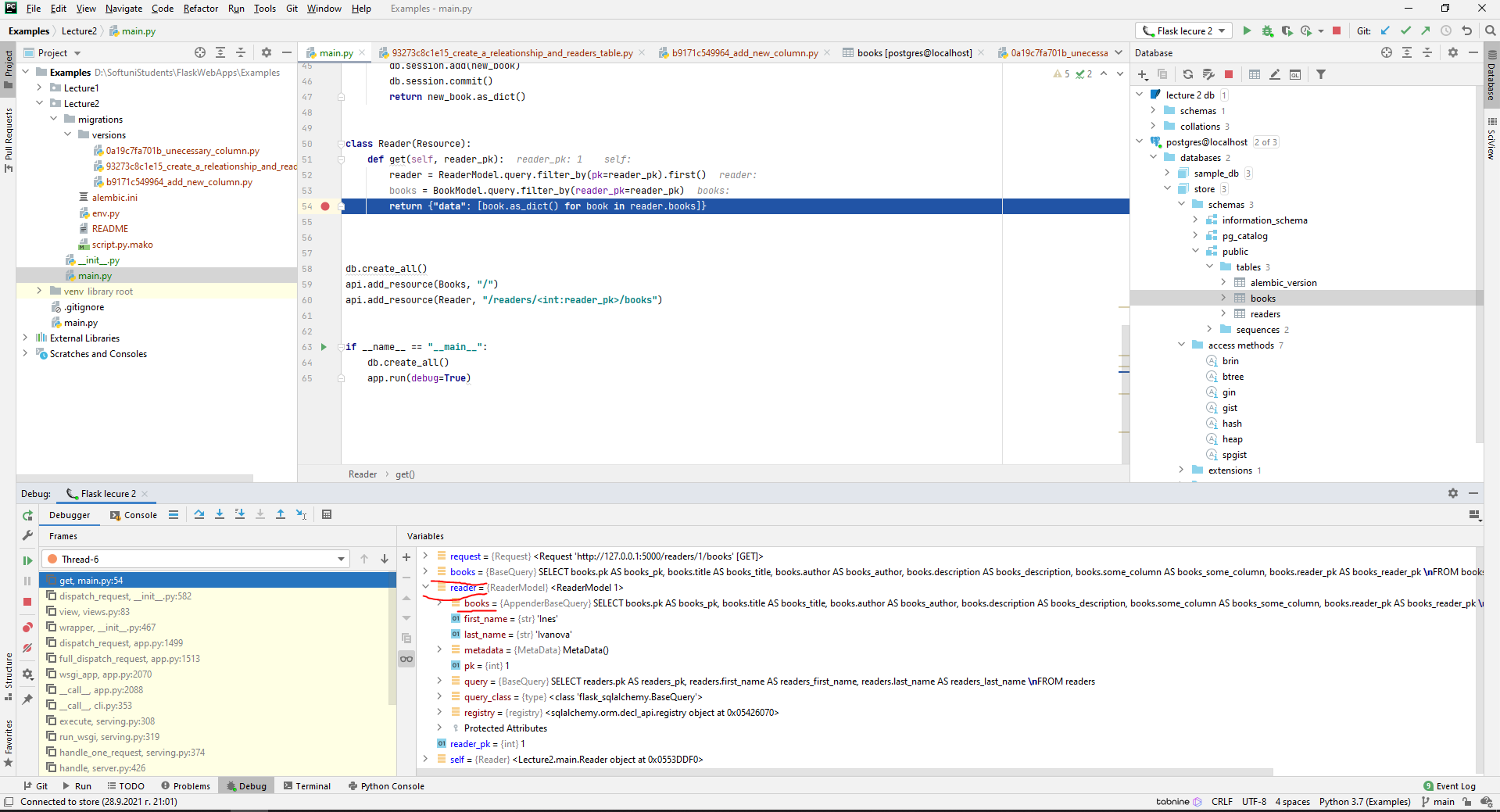
**class** Reader(Resource):  
 **def** get(self, reader\_pk):  
 reader = ReaderModel.query.filter\_by(pk=reader\_pk).first()  
 books = BookModel.query.filter\_by(reader\_pk=reader\_pk)  
 **return** {**"data"**: [book.as\_dict() **for** book **in** reader.books]}  
  
  
  
db.create\_all()  
api.add\_resource(Books, **"/"**)  
api.add\_resource(Reader, **"/readers/<int:reader\_pk>/books"**)

If you make a request with POSTMAN like this you will see the result:



Now start debugging with your Pycharm and put a breakpoint in Reader class on the return row.

You can then play with the debugger and discover what the result of the two queries is and how you can access both rows in the two tables from each one of them:



## Small refactor

Remember how we left the connection string unprotected?

app.config[**'SQLALCHEMY\_DATABASE\_URI'**] = **f'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:

**from** decouple **import** config  
**from** flask **import** Flask, request  
**from** flask\_sqlalchemy **import** SQLAlchemy  
**from** flask\_restful **import** Resource, Api  
**from** flask\_migrate **import** Migrate  
  
  
app = Flask(\_\_name\_\_)  
  
db\_user = config(**'DB\_USER'**)  
db\_password = config(**"DB\_PASSWORD"**)  
  
app.config[**'SQLALCHEMY\_DATABASE\_URI'**] = **f'postgresql://{**db\_user**}:{**db\_password**}@localhost:5433/store'**

….

Start the application again and make sure it is working correctly, 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.