Database Project Part 3

Nahtan Lichtle & Alain Riou

May 19, 2018

1 Introduction

For our project, we are using Django. As a result, we do not manipulate SQL code directly. The SQL code to generate the tables is generated automatically by Django and the requests are handled in Python. Thus this archive doesn't contain any SQL code. The script used to fill the database is described in the last but one section. As for the instructions to set up the databases, I have written them in the last section as it is more readable than it would have been in a README file.

2 Models

We started by implementing the relations in `models.py`. There are 8 models:

- `CustomUser` which inherits from Django's native `User` model ; some of the attributes use a regex for the validation, like the name or the address, others are just choices like the authorization level which can be between 0 and 4 or the privacy level which can be between 0 and 2.
- `Friendship` : `CustomUser` contains a many-to-many relationship to itself which goes through the table `Friendship` in order to store additionnal information like the status of the relationship (waiting for approval, active) or the date it started.
- `Recommendation` : same as for `Friendship`, a book recommendation happens between two users and this table stores the date of the recommendation and the associated book.
- `Category` : just an id and a name for a book category.
- `Rating` : an evaluation (from 1 to 5) of a book by a user. To avoid a user from rating the same book several time, the couple (book, user) is set as unique.
- `Review` : the review of a book, with a content and a summary. Also stores a number of likes, dislikes and reports, and a many-to-many relationship with `CustomUser` in order to store who has already voted (likes/dislikes) on this review. A user can rate a book without reviewing it but if he reviews it, he has to rate it. That is why there is a one-to-one relationship from `Review` to `Rating`.
- `Comment` : a comment on a review, whose attributes are a date, a content, the user who posted the comment and the review on which it is posted, as a foreign key.
- `Book` : regex validators for the ISBN (which is the primary key) and the cover image URL, choices for the book status and the year of publication and a custom validator to ensure the author is entitled to publish a book. The author is a foreign key to a `CustomUser` and category to `Category`, however author can be null in case of an author that is not a user on the website. In this case, a string author pseudonym can be provided.

3 Views

To each action or page on the site correspond three things :
- a view in `views.py` which handles the logic, ie makes the requests and/or handles the forms
- a template (`.html` file) which handles the display of the information

- a url in `urls.py`, which associates a name to a view and handles the urls automatically

And sometimes, when needed, a form in `forms.py`.

Each view correspond to an action and is rather short and self-explanatory. Either it is a view which displays information, so it starts by making requests to extract information from the database, then sends it to the template to be printed, or it is a form view which either displays a form, or receives the data sent by the form (using post), and if it is valid, modifies the database accordingly.

There are also some views that are not associated with a template, for instance the view which deletes a book will just verify the user is authorized to delete it and if so, delete it and redirect the user to the previous page.

The functions we used to make requests in the views are :

- `get` which returns the instance of a relation satisfying certain conditions (basically a `SELECT ... WHERE ...`), or fails if it doesn't exist or isn't unique
- in the same lines, `get_or_else_404` which is a get that returns a 404 error page in case of a failure ; but I also used `get` with a **try ... except** block catching the `ObjectDoesNotExist` exception.
- `filter` which returns all the instances of a relation satisfying certain conditions (also a `SELECT ... WHERE ...`).
- `save` which saves the object in the database (it can be either an `INSERT INTO` or an `UPDATE`)
- `delete` which is with no surprise a `DELETE`

Last but not least, there are some very convenient shortcuts. For instance, the model `CustomUser` contains an attribute books which is a many-to-many relationship towards the model `Book`. Then, given some book, we can get all the customers whose books attribute contains our book by doing `book.customuser_set`. However, it was a bit more complicated with the friendship relationship for instance, as a many-to-many relationship to oneself which stores extra attributes cannot be symmetrical, while the friendship relation we wanted was symmetrical in theory.

4 Database filling

We used the Python script `getDataBX.py` (which requires large input files that are not provided in this archive) to fill the database. This script reads some `csv` files which we have found online, and get its information in order to write another `csv` file, adapted to the constraints of our database. By executing the different fonctions in this script, we wrote files that we were able to transform into SQL tables by using the psql command `COPY`.

For books, we used a database found on `http://www2.informatik.uni-freiburg.de/c̃ziegler/BX/`, which contains a huge amount of data about books, but also ratings of these books by anonymized users all over the world. Unfortunately a considerable amount of rated books had an ISBN which doesn't exist in the table which counts the books. Therefore, we would have needed to check, for each rating, if the related book is in the right table. This would have taken way too much time to compute, which is why we don't have any default ratings in our database.

These data have been the main resources for our own building, but because of the lack of information, we had to randomize some of the attributes we needed, like dates for instance.

Therefore the functions of this script are basically giant loops on the lines of our input csv files, so they are not algorithmically really interesting.

However, an important aspect is that, because of the constraints of primary keys, foreign keys, etc. it is necessary to fill the database in a particular order. For instance, categories need to be fed to the database before books, as a book contains a category attribute.

5 Instructions to try out the project

Our project uses Django, so you must of course have installed it.

First, you need to have a database and to connect it in the section `DATABASES` in `online_library/settings.py`. Then create the tables by running `python3 manage.py makemigrations library` followed by `python3 manage.py migrate`. If there is no error so far, the tables have been created.

Then you need to fill the database. To do that, there are three `.csv` files in the folder `fill_db`, which correspond to a lot of users, books, and books categories. Connect into your database (if you are using `psql`, you would run `psql <database_name>`), then execute the following commands in the order:

```
COPY library_customuser
    (password,last_login,is_superuser,username,is_staff,is_active,date_joined,
     first_name,last_name,address,email,birthday,balance,authorization_level,privacy_level)
         FROM '<path to data_users.csv>' DELIMITER '|' NULL '';

COPY library_category (name) FROM '<path to data_categories.csv>' DELIMITER '|' NULL '';

COPY library_book
    (isbn,status,title,price,year_of_pub,image_url,category_id,author_pseudonym) FROM
    '<path to data_books.csv>' DELIMITER '|' NULL '';
```

Your database is now filled. You can access the website by running the command `python3 manage.py runserver` and going to your local URL `127.0.0.1:8000`.

I recommend you create two different users (having usernames `user1` and `user2`) using the *Sign Up* form, then execute the following commands in your terminal :

```
python3 manage.py shell
>>> from library.models import CustomUser
>>> usr = CustomUser.objects.get(username="<user1>")
>>> usr.authorization_level = 4
>>> usr.save()
```

This way, `user1` will be a moderator and `user2` will be a basic user. If you connect the two users on two different pages at the same time, you can try every possible actions.

To manually modify data, you can also set `usr.is_superuser` and `usr.is_staff` to `True` and access the Admin panel by clicking the link Admin located at the bottom-right corner of the page.