Exercise: Spring Data Intro

This document defines the exercise assignments for the "Spring Data" course @ SoftUni.

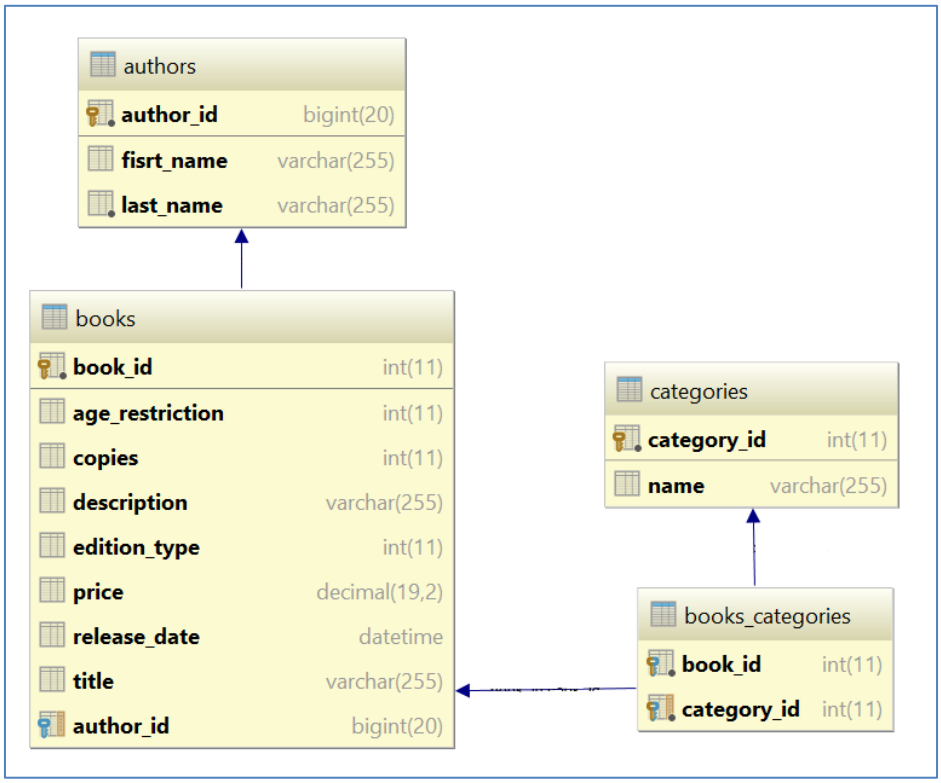
1. Bookshop System

Create database for a **Bookshop System**. A bookshop keeps **books**. A book can have **one author** and many **categories**. Each category can be placed on many books. Let's create a class for each main table.

- **Book** - id, title (between 1...50 symbols), description (optional, up to 1000 symbols), edition type (**NORMAL**, **PROMO** or **GOLD**), price, copies, release date (optional), age restriction (**MINOR, TEEN or ADULT**)
- **Author** - id, first name (optional) and last name
- **Category** - id, name

Assume everything, which is not market (optional), is mandatory.

The final schema of the database should look like that:



Seed Data into the Database

Create **`seedDatabase()`** method in the **`ConsoleRunner` class**. That method will fill records in the database.

Use the provided **files** (**categories.txt**, **authors.txt**, **books.txt**) and import the data from them.

Example of seedBooks method

`Files.readAllLines(Path.of(RESOURCE_PATH + BOOKS_FILE_NAME))`
`.forEach(row -> {`
`String[] data = row.split("\\s+");`

```
                Author author = authorService.getRandomAuthor();
                EditionType editionType = EditionType.values()[Integer.parseInt(data[0])];
                LocalDate releaseDate = LocalDate.parse(data[1],
                                                DateTimeFormatter.ofPattern("d/M/yyyy"));
                int copies = Integer.parseInt(data[2]);
                BigDecimal price = new BigDecimal(data[3]);
                AgeRestriction ageRestriction = AgeRestriction
                                                .values()[Integer.parseInt(data[4])];
                String title = Arrays.stream(data)
                        .skip(5)
                        .collect(Collectors.joining(" "));
                Set<Category> categories = categoryService.getRandomCategories();


                Book book = new Book(title, editionType, price, releaseDate,
                        ageRestriction, author, categories, copies);


                bookRepository.save(book);
            });
```

Write Queries

Write the following queries that:

1. Get all **books** after the **year 2000**. Print only their **titles**.
2. Get all **authors** with at least **one book with release date before 1990**. Print their **first name** and **last name**.
3. Get all **authors**, ordered by the **number of their books** (descending). Print their **first name**, **last name** and **book count**.
4. Get all **books** from author **George Powell**, ordered by their **release date** (descending), then by **book title** (ascending). Print the book's **title**, **release date** and **copies**.

2. User System *

Your task is to create a table **Users**. The table should contain the following fields:

- **id** – Primary Key (number in range [1, 2^{31}-1])
- **username** – Text with length between 4 and 30 symbols. Required.
- **password** – Required field. Text with length between 6 and 50 symbols. Should contain at least:
  - 1 lowercase letter
  - 1 uppercase letter
  - 1 digit
  - 1 special symbol (!, @, #, $, %, ^, &, *, (,), _, +, <, >, ?)
- **email** – Required field. Text that is considered to be in format **<user>@<host>** where:
  - **<user>** is a sequence of letters and digits, where '**.**', '**-**' and '**_**' can appear between them (they cannot appear at the beginning or at the end of the sequence).
  - **<host>** is a sequence of at least two words, separated by dots '**.**' (dots cannot appear at the beginning or at the end of the sequence)
- **registered_on** – Date and time of user registration

- **last_time_logged_in** – Date and time of the last time the user logged in
- **age** – number in range [1, 120]
- **is_deleted** – Shows whether the user is deleted or not

User Towns

User should have **born town** and **currently living** in town. The town has **name** and **country**, based on its location.

User Names

Add 2 new properties to the user - **first name** and **last name**. Also, add one more property **fullName** that would return the concatenation of **first and last name separated by a single space**. That property must be generated only when we need it (there is no need to keep it in the database).

Friends

Let's say that the **user can have many friends** that would be again other users (or in other words **many to many self-relationship**).

Albums

Previously 1 user was able to upload only 1 picture (just his/her profile picture). Now each user is capable of creating **personal albums**. Each album has **name, background color** and information whether it **is public or not**. Each picture has **title**, **caption** and **path** on the file system. An album can contain many pictures and one picture can be present in many albums. Each user can have many albums but an album can have only one owner user.

*Email Annotation

Make a validation annotation **`@Email`** that can be used on string fields. The property should check if the value of the property is valid. One email is valid if it follows the format **<user>@<host>**, where:

- **<user>** is a sequence of letters and digits, where '**.**', '**-**' and '**_**' can appear between them.
  - Examples of **valid** users: "**stephan**", "**mike03**", "**s.johnson**", "**st_steward**", "**softuni-bulgaria**", "**12345**".
  - Examples of **invalid** users: "**--123**", ".....", "**nakov_-**", "**_steve**", "**.info**".
- **<host>** is a sequence of at least two words, separated by dots '**.**'. Each word is sequence of letters and can have hyphens '**-**' between the letters.
  - Examples of hosts: "**softuni.bg**", "**software-university.com**", "**intoprogramming.info**", "**mail.softuni.org**".
  - Examples of invalid hosts: "**helloworld**", "**.unknown.soft.**", "**invalid-host-**", "**invalid-**".
- Examples of **valid emails**: info@softuni-bulgaria.org, kiki@hotmail.co.uk, no-reply@github.com, s.peterson@mail.uu.net, info-bg@software-university.software.academy.
- Examples of **invalid emails**: --123@gmail.com, …@mail.bg, info@info.info, _steve@yahoo.cn, mike@helloworld, mike@.unknown.soft., s.johnson@invalid-.

Use that annotation on the previous problems to validate any fields containing e-mail address.

Hint

Use Hibernate Validator.

*Password Annotation

Make validation annotation **`@Password`** that can be used to validate string fields. The property should check if the value of the field is valid. In the constructor, the password should receive minimum and maximum length of the

password. As optional parameters, we should be able to provide whether the password should contain lowercase letter, uppercase letter, digit or special symbol.

```
@Password(minLength = 6,
        maxLength = 40,
        containsDigit = true,
        containsLowercase = true)
private String password;
```

```
@Password(minLength = 4,
        maxLength = 30,
        containsDigit = true,
        containsLowercase = true,
        containsUppercase = true,
        containsSpecialSymbol = true,
        message = "Invalid password")
private String password;
```

Use that annotation on the previous problems to validate any fields containing password.

Get Users by Email Provider

Write program that print all usernames and emails of users by given email provider.

Example

Input	Output
gmail.com	...@gmail.com ...@gmail.com ...@gmail.com
yahoo.co.uk	...@yahoo.co.uk ...@yahoo.co.uk
abv.bg	No users found with email domain abv.bg

Remove Inactive Users

Write a program that sets the **`is_deleted`** field to true for all users, who have not been logged in after given date. Print the number of users that have been set as deleted. Then delete all users that have been marked for removal.