
PROJECT ASSIGNMENT 3

BPC-BDS

Author

Marcel Kubín

247568

Contents

1	Library management system (LMS)	3
1.1	Overview	3
1.2	Goal	3
2	JavaFX App	3
2.1	Login GUI	4
2.2	Standard user GUI	5
2.3	Librarian GUI	7
2.4	Logging	14
2.5	Dependencies	14
3	PostgreSQL database	14
3.1	Containerization	15
3.1.1	Compose file	15
3.2	PostgreSQL configuration	16
3.3	Flyway configuration	17
3.4	pgAdmin configuration	17
3.5	Database backup	17
4	Build & Run	18
4.1	Generate SSL certificates	18
4.2	Docker compose	18
4.3	JavaFX App	18
4.4	Log in credentials	19
4.5	Summary	19

1 Library management system (LMS)

1.1 Overview

The basic idea of this project is an application for library network management, providing access to selected information and functionalities for both library staff and standard users. Project consists of **JavaFX** application, **PostgreSQL** database running in **Docker** container managed by **Flyway** database-migration tool also running in **Docker** container.

1.2 Goal

Goal of the project was not only to create **JavaFX** application with **PostgreSQL** database from previous assignments, but also learn how to everything join together. And get familiar with other tools when deploying such application, for example various logging tools, version control systems like **Gitlab** for source code or **Flyway** for database, containerization options like **Docker** and also get some kind of idea about workflow and research needed for accomplishing similar tasks.

2 JavaFX App

JavaFX application is structured into multiple layers:

1. **Controllers** handles user input and also show program output to the user. They manage the interaction between graphical user interface and application logic.
2. **Service** layer acts as an intermediary between the controllers and data layer (repositories). It encapsulates the application's business logic, ensuring that controllers don't directly interact with the data layer.
3. **Data** layer consists of repositories that interact directly with the database, through **HikariCP** configuration layer. The repository implement database operations like create, read, update, delete (CRUD).
4. **Config** layer in this application is using **HikariCP**, which is high-performance JDBC connection pool and it is responsible for managing database connections in the application.
5. **API** layer is used for mapping database entities to Java objects and also other way around.
6. **Exceptions** defined to ensure easier debugging even across described multi layer design.
7. **Main** class acts as entry-point to the application and is responsible for initializing config layer and than calling **App** class.
8. **App** class is starting point for **JavaFX** application it extends **Application** class and initializes primary application **stage**.
9. **Resources** is a directory used for storing data needed by application. In this case **fxml** files, images or custom **css**.

2.1 Login GUI

Login window is the first window presented to the user after application starts. It can be used to login with already created account or to open other small window to create entirely new user account. Necessary validations on user input are placed in both windows before Sign in/Create button can be clicked. Roles that can user assign to new account are restricted to only standard user roles like basic, student or child.

Note: These roles are currently selected in code, but it would be beneficial if database role entity itself contain column identifying if role is high privileged so they can be directly queried from database.

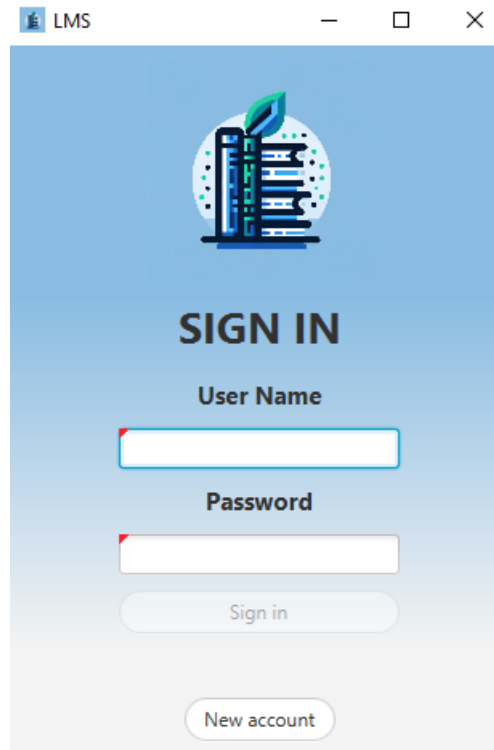


Figure 1: Login window

LMS Create account

New account

Username:

Password:

First name:

Last name:

Role:

Phone number:

Email:

Date of birth:

Cancel Create

Figure 2: Create account window

2.2 Standard user GUI

Standard user window is opened if user with one of the standard user roles is logged in. Right on the top there are buttons for closing the window and also for managing the account. In account management user can see his id, username and role. User can also change his name, phone number, email address or his password from this window. Also user can delete his account from the database (confirmation pops up before deletion), this action at the back-end also deletes user contacts and his associations to book loans.

ID	Book	Issue Date	Due Date	Returned
4	Songs of the Siren	2023-09-11	2023-10-11	true
5	Harmony of the Cosmos	2023-09-11	2023-10-11	true
6	Harmony of the Cosmos	2023-09-11	2023-10-11	true

Figure 3: Standard user window

Account

ID: 14

Username: carterp

Role: basic

First name:

Last name:

Phone number:

Email:

New password:

Figure 4: Manage account window

Further standard user window contains multiple tabs:

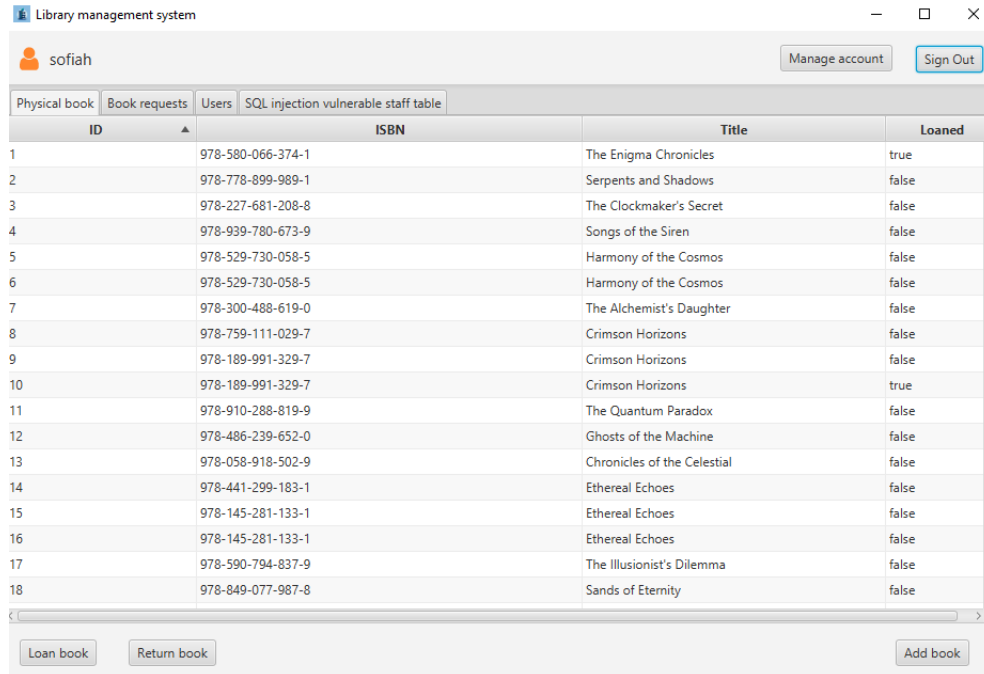
1. **Loans** tab displays users loans with basic info about them and mainly if user has

already returned loaned book.

2. **Books** tab displays all books currently registered in the system. At the bottom of this tab user can perform search based on a book title. At the database level this search operation performs similarity check using `pg_trgm` extension.
3. **Book request** tab gives user an option to create book request by submitting book ISBN and title, which is then displayed in librarian window to process it.
4. **Libraries** tab displays libraries joined in the system with their addresses so user can check where for example his book can be returned, because it can be done in any of these libraries.

2.3 Librarian GUI

Librarian window is opened when librarian account is logged in. Same as in the Standard user GUI on the top of the window user can get to the account management.

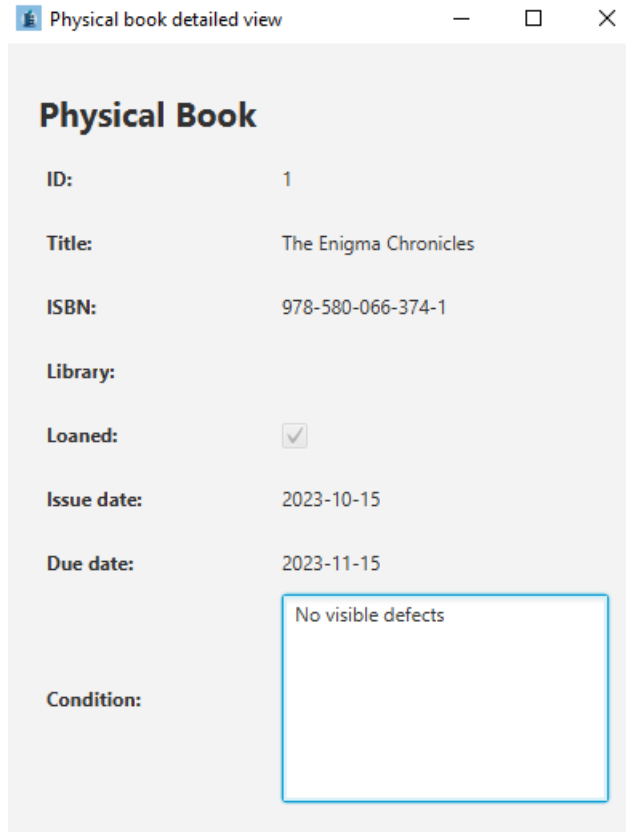


ID	ISBN	Title	Loaned
1	978-580-066-374-1	The Enigma Chronicles	true
2	978-778-899-989-1	Serpents and Shadows	false
3	978-227-681-208-8	The Clockmaker's Secret	false
4	978-939-780-673-9	Songs of the Siren	false
5	978-529-730-058-5	Harmony of the Cosmos	false
6	978-529-730-058-5	Harmony of the Cosmos	false
7	978-300-488-619-0	The Alchemist's Daughter	false
8	978-759-111-029-7	Crimson Horizons	false
9	978-189-991-329-7	Crimson Horizons	false
10	978-189-991-329-7	Crimson Horizons	true
11	978-910-288-819-9	The Quantum Paradox	false
12	978-486-239-652-0	Ghosts of the Machine	false
13	978-058-918-502-9	Chronicles of the Celestial	false
14	978-441-299-183-1	Ethereal Echoes	false
15	978-145-281-133-1	Ethereal Echoes	false
16	978-145-281-133-1	Ethereal Echoes	false
17	978-590-794-837-9	The Illusionist's Dilemma	false
18	978-849-077-987-8	Sands of Eternity	false

Figure 5: Librarian window

This window also consists of multiple tabs:

1. **Physical book** tab displays book as in Standard user window, but with a difference that they are `physicalbook` entities from database instead of `book` entities shown in Standard user window. This means that there are shown all copies of all books in system instead of just all different books. Each book in the view also has an option (by right clicking on the book) to show detailed view.



The image shows a window titled "Physical book detailed view" with standard window controls (minimize, maximize, close). Inside the window, the title "Physical Book" is displayed at the top. Below it, a form contains the following fields and values:

ID:	1
Title:	The Enigma Chronicles
ISBN:	978-580-066-374-1
Library:	
Loaned:	<input checked="" type="checkbox"/>
Issue date:	2023-10-15
Due date:	2023-11-15
Condition:	<div>No visible defects</div>

Figure 6: Physical book detailed view window

The next thing that can be done in **Physical book** tab is creating new loan of a book. This can be done clicking **Loan book** button at the bottom which opens up a new window, where librarian needs to enter user ID, physical book ID and also issue date together with due date. Issue date is preset by App to current date and due date to after month, but these dates can be changed to custom ones. Also on ID fields there are regex **validators** checking if user input is a number.

Note: *Currently IDs have to be written manually, but in real application would best option to connect application to some kind of scanner which can read users library cards and for example read codes on books so they are loaded automatically.*

Loan book

User ID:

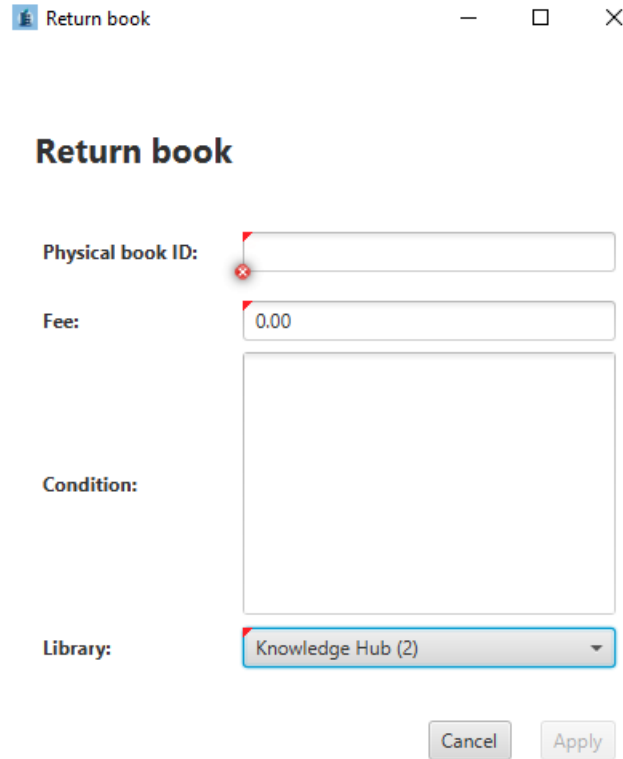
Physical book ID:

Issue date:

Due date:

Figure 7: Loan book window

When book can be loaned, it also needs to be able to be returned. This can be done by clicking **Return book** button. In the popped up window librarian enters physical book ID, fee (if user for example damaged a book; value is preset to 0.00), chooses library from choice box. Libraries are loaded directly from database to choice box to ensure if new library is added to system it can be immediately selected in choice box. Also librarian can alter the condition of the book from last loan. Condition are is automatically populated with latest condition from database when physical book ID is filled out and passes validation for being a number. There is also regex `validator` on fee text field so that only numbers with maximum two decimal places can be entered.




The image shows a software window titled "Return book" with standard window controls (minimize, maximize, close) in the top right corner. The window contains a form with the following fields:

- Physical book ID:** A text input field with a red error icon (a circle with an 'x') to its left.
- Fee:** A text input field containing the value "0.00".
- Condition:** A large, empty text area.
- Library:** A dropdown menu showing "Knowledge Hub (2)" with a downward arrow.

At the bottom right of the form are two buttons: "Cancel" and "Apply".

Figure 8: Return book window

The last functionality of **Physical book** tab is to add new a book to the system. This can be done by clicking **Add book** button. Librarian needs to enter information like book ISBN, title, publication year, current condition and so on. But there is a possibility that book is already in a system, means that librarian is adding only another copy. In this case when librarian fills out book ISBN almost all of the information about the book are automatically filled out and locked. Librarian than only needs to specify book copy specific information like condition and to which library is adding the book. Necessary **validators** are applied on all of the fields, either numeric or text based.

 Add book

Add book

ISBN:	<input type="text" value="978-580-066-374-1"/>
Title:	<input type="text" value="The Enigma Chronicles"/>
Year:	<input type="text" value="1979"/>
Evaluation:	<input type="text" value="6.35"/>
Genre:	<input type="text" value="Non-Fiction"/>
Language:	<input type="text" value="Polish"/>
Binding:	<input type="text" value="Softcover"/>
Literary period:	<input type="text" value="Victorian"/>
Library:	<div><div></div><div></div></div>
Condition:	<div></div>

Cancel

Add

Figure 9: Add book window when already known ISBN was entered

2. **Book requests** tab shows book requests submitted from Standard users. Upon right click on individual book requests Librarian can either delete (confirmation pops up before deletion) it or edit values in book request, for example correct book title.

Edit book request

Book request

ID: 1

Title: Into the Unknown

ISBN: 978-112-233-445-5

Cancel Apply

Figure 10: Book request edit window

3. **Users** tab displays all Standard users to the librarian, means librarian can not see other librarians or other accounts with higher privilege. When right clicking user librarian can either view user detailed view or edit user account, for example change his phone number, email address but also change his role from for example child to student.

User detailed view

User

ID: 2

Username: alicesmith456

First name: Alice

Last name: Smith

Role: student

Phone number: 444-234-5678

Email: gamer_girl42@example.org

Date of birth: 1985-08-23

Figure 11: User detailed view window

Edit user

User

ID: 2

Username: alicesmith456

First name: Alice

Last name: Smith

Role: student

Phone number: 444-234-5678

Email: gamer_girl42@example.org

Date of birth: 23.08.1985

Cancel Apply

Figure 12: User edit window

4. **SQL injection vulnerable staff table** is last tab in Librarian window it is not directly part of the application but was part of the assignment. This tab demonstrate SQL injection when prepared statement are not used when querying the database and user input is not properly sanitized. In the tab username and password can be entered one of the valid combinations is **arcticstar** as username and **qwerty** as password. When entered table view shows information about user:

Note: As password filed in this tab is used only classic text field instead of a password field so the password can be seen while demonstration.

Username: arcticstar Password: qwerty Show my info

ID	Username	Join date	Phone number
3	arcticstar	2020-04-27	+1-767-916-6330

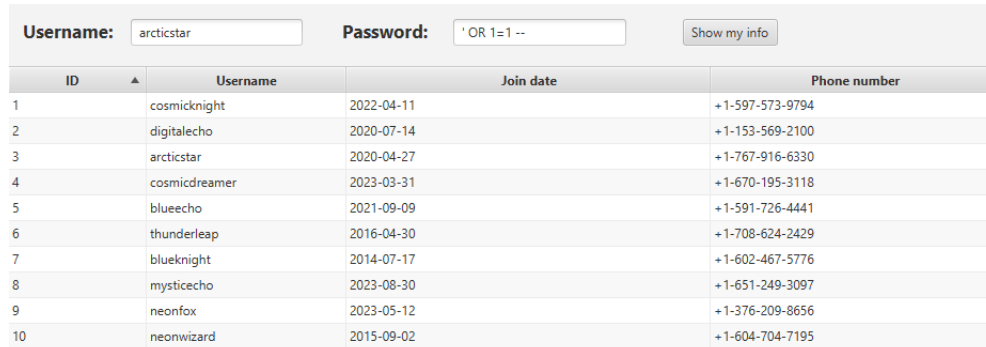
Figure 13: Valid credentials in SQL injection tab

SQL query behind this table looks just simple like this:

```
SELECT staff_id, username, join_date, phone_number
FROM bds.unsecure_staff
WHERE username = ' + username + ' AND password = ' + password + ';
```

Not using prepared statements and lack of user input validation does not prevent attacker use something like ' OR 1=1 -- as input string. Because standard statements are not precompiled like prepared statements their logic can be altered by inserting

malicious string. This exact string escapes string for password with ' than utilizes OR condition which is always true effectively making the whole WHERE clause always true for all records `1=1` and at the end `--` is placed so the rest of the query is commented out.



ID	Username	Join date	Phone number
1	cosmicknight	2022-04-11	+1-597-573-9794
2	digitalecho	2020-07-14	+1-153-569-2100
3	arcticstar	2020-04-27	+1-767-916-6330
4	cosmicdreamer	2023-03-31	+1-670-195-3118
5	blueecho	2021-09-09	+1-591-726-4441
6	thunderleap	2016-04-30	+1-708-624-2429
7	blueknight	2014-07-17	+1-602-467-5776
8	mysticecho	2023-08-30	+1-651-249-3097
9	neonfox	2023-05-12	+1-376-209-8656
10	neonwizard	2015-09-02	+1-604-704-7195

Figure 14: Dumping whole table with SQL injection

Instead of dumping all the data from the table attacker can also utilize SQL injection to DROP the whole table with query something like `' ; DROP TABLE unsecure_staff --` as input string. Only caveat at this is that user executing this DROP query has to be owner of the table. But even if user is not owner of the table this kind of SQL injection gives attacker opportunity to execute basically any query which executing database role has privileges for, e.g. attacker can for example perform whole DELETE query instead of DROP query.

Note: *Utilizing queries which makes whole WHERE condition true is not considered a good practise at penetration tests because when such a conditions accidentally became part of the for example DELETE query it can cause a lot of damage on the tested system.*

2.4 Logging

Logging in JavaFX application was made with use of `slf4j-api` and `logback-classic` Maven dependencies added in `pom.xml` file. Logging was configured in `logback.xml` file to save logs daily with `logs/logFile-%d{yyyy-MM-dd}.log` file pattern.

2.5 Dependencies

I have generated project dependencies with `mvnproject-info-reports:dependencies` command, this command produces html web page with used dependencies and their licenses. Web page can be found in `BPC-BDS-Project/App/Dependencies/dependencies.html`.

3 PostgreSQL database

Database design was main subject of previous assignments so in this paper I would like to focus more on managing the database, its containerization and so on.

3.1 Containerization

For containerization I have used **Docker** platform, where I have created `compose.yaml` file centralizing app deployment. Only the **JavaFX** application needs to be compiled and run outside of the **Docker** environment. It is possible to run even **JavaFX** application inside the container by creating own **Dockerfile**, but because application uses GUI it would require to start service like **X Server** socket on the host system so the application GUI can be passed from the container. To keep things simple I have decided not to dockerize **JavaFX** application, but only **PostgreSQL** database together with management tools like already mentioned **Flyway** or **pgAdmin**.

3.1.1 Compose file

`compose.yaml` file contains definitions for all three containers used in my **Docker** deployment. These are **PostgreSQL** database, **Flyway** database-migration tool and **pgAdmin** database administration tool. Each container is added to collective network, has mounted necessary files from host system, sometimes there is also a need to change these files or container properties at container entrypoint and if needed create **Docker** volume for its data storage.

Deploying **PostgreSQL** container:

```
bds-postgres-db:
  image: 'postgres:15-alpine'
  container_name: bds-postgres-db
  restart: unless-stopped
  ports:
    - '127.0.0.1:5432:5432'
  environment:
    #PGDATA: '${POSTGRES_DATA_DIRECTORY}'
    POSTGRES_USER: '${POSTGRES_USER}'
    POSTGRES_PASSWORD: '${POSTGRES_PASSWORD}'
  healthcheck:
    test:
      - CMD-SHELL
      - 'pg_isready -U ${POSTGRES_USER}'
    interval: 30s
    timeout: 30s
    retries: 3
  volumes:
    - 'postgres-data:/var/lib/postgresql/data'
    - 'postgres-backup:/postgres_backup'
    - './db_init:/docker-entrypoint-initdb.d/'
    - './ssl/server.key:/etc/postgres/server.key'
    - './ssl/server.crt:/etc/postgres/server.crt'
    - './ssl/root.crt:/etc/postgres/root.crt'
    - './pg_hba.conf:/etc/postgres/pg_hba.conf'
    - './db_backup.sh:/etc/postgres/db_backup.sh'
  networks:
```

```

- bds-project-net
entrypoint:
- /bin/sh
- '-c'
- >-
  chown postgres:root /etc/postgres/server.key &&
  chown postgres:root /etc/postgres/server.crt &&
  chown postgres:root /etc/postgres/root.crt &&
  chown postgres:root /etc/postgres/pg_hba.conf &&
  chown postgres:root /etc/postgres/db_backup.sh &&
  chmod 600 /etc/postgres/server.key &&
  chmod 600 /etc/postgres/server.crt &&
  chmod 600 /etc/postgres/root.crt &&
  chmod 600 /etc/postgres/pg_hba.conf &&
  chmod 700 /etc/postgres/db_backup.sh &&
  (crontab -l 2>/dev/null; echo "0 0 * * * /etc/postgres/db_backup.sh") | crontab - &&
  crond -b &&
  docker-entrypoint.sh postgres
  -c ssl=on
  -c ssl_cert_file=/etc/postgres/server.crt
  -c ssl_key_file=/etc/postgres/server.key
  -c ssl_ca_file=/etc/postgres/root.crt
  -c hba_file=/etc/postgres/pg_hba.conf
  -c log_destination='stderr'
  -c logging_collector=on
  -c log_filename='postgresql-%Y-%m-%d_%H%M%S.log'
  -c log_file_mode='0640'
  -c log_truncate_on_rotation=off
  -c log_min_messages=warning
  -c log_min_error_statement=error
  -c log_connections=on
  -c log_disconnections=on
  -c log_statement=ddl

```

3.2 PostgreSQL configuration

All of the PostgreSQL configuration and initialization is performed either directly within the `compose.yaml` file or in one database initialization script `db_init.sql` placed in `docker-entrypoint-initdb.d` directory which accommodates all scripts that are automatically run at database initialization. In this script I am creating roles for my application and other management services, creating `library_management_system` database with schema `bds` and assigning basic permissions and also creating extensions. I have assigned non-superuser privileges for my application role.

Note: Roles for database management services like *Flyway* are currently set as superusers,

but even them do not need superuser privileges and would be the best practise to lower their privileges.

3.3 Flyway configuration

Almost all of the Flyway configuration is made using `flyway.conf` file, where JDBC driver URL is specified and other information needed by Flyway to perform migrations.

`flyway.conf` file configuration:

```
flyway.url=jdbc:postgresql://bds-postgres-db:5432/library_management_
system?ssl=true&sslmode=verify-full&sslrootcert=/etc/postgres/root.crt
&sslcert=/etc/postgres/flyway.crt&sslkey=/etc/postgres/flyway.key.der
flyway.user=bds-flyway
flyway.password=gjRsWsdLj35VlgDF5JL
flyway.connectRetries=30
flyway.defaultSchema=bds
flyway.locations=filesystem:/flyway/sql
```

3.4 pgAdmin configuration

pgAdmin configuration is made mainly with `servers.json` file and also with `pgpass` file. `servers.json` file specifies which servers should be automatically added to pgAdmin interface and how should pgAdmin perform authentication to the database server. `pgpass` file contains only server, username and password if password authentication is performed.

`servers.json` file configuration:

```
{
  "Servers": {
    "1": {
      "Name": "bds-project",
      "Group": "Servers",
      "Host": "bds-postgres-db",
      "Port": 5432,
      "MaintenanceDB": "postgres",
      "Username": "postgres",
      "SSLMode": "verify-full",
      "PassFile": "/pgpassfile",
      "SSLCert": "/postgres.crt",
      "SSLKey": "/postgres.key"
    }
  }
}
```

3.5 Database backup

Database backup is performed by `db_backup.sh` script mounted into PostgreSQL container. This script uses `pg_dump` tool to extract data from database and place them in `postgres_`

backup directory which is also mounted as separated Docker volume. Command also saves log about backup where errors will be written if something wrong happens.

Used `pg_dump` command in script:

```
pg_dump -U $DB_USER $DB_NAME > $FILE_PATH".sql.bak" 2>> $FILE_PATH".bak.log"
```

Script also has registered cron job to run every midnight. This cron job is created already in entry-point of the container with:

```
(crontab -l 2>/dev/null; echo "0 0 * * * /etc/postgres/db_backup.sh") | crontab -
```

For restoring created backup I have created `db_restore.sh` script which takes generated database backup script as first positional argument. Currently it behaves in such way that it DROPS original database, create new one with same name and then load backup. This behaviour can be easily altered or the script can be written more modular, i.e. for example let user specify database name as positional argument. For restoring `psql` tool is used. Restoring database to its original state by applying database backup from first positional argument:

```
psql -U $DB_USER $DB_NAME < $1
```

4 Build & Run

First requirement is that Java 17 and Maven has to be installed. Further to start the application from scratch a few simple steps have to be performed:

4.1 Generate SSL certificates

All communication between JavaFX application and all containers is secured by SSL with not only server but also client authentication. So server self-signed certificate together with client certificates has to be generated. In order to not include private keys in my Gitlab repository I have created a simple script for Windows `KeyCertGen.cmd` and for GNU/Linux `KeyCertGen.sh` that can generate brand new set of keys/certificates and place them in correct directories. The only requirement is to have `openssl` installed and run the script directly from where is located, so that all keys/certificates end up in correct locations from where are loaded by containers and application.

4.2 Docker compose

Next step is to run `docker compose` command with `compose.yaml` file placed in Database directory.

4.3 JavaFX App

Last step is move to the App directory compile JavaFX application with `mvncleaninstall` and then run it with `java -jar .\target\bds-project-app-1.0.jar`.

4.4 Log in credentials

Each user is logged in by its username and password, password of each user is set to be their username with year when they were born. This information can be extracted from Flyway migration V3__db_data.sql, but here are some valid sets of credentials for both types of accounts:

1. **Standard user** account:

- **username:** carterp
- **password:** carterp1983

2. **Librarian** account:

- **username:** sofiah
- **password:** sofiah2002

4.5 Summary

How to start application in commands:

```
git clone <repository>
cd .\BPC-BDS-Project\
.\KeyCertGen.cmd
docker compose -f .\Database\compose.yaml up -d
cd .\App\
mvn clean install
java -jar .\target\bds-project-app-1.0.jar
```

Note: *Some commands needs to be altered when GNU/Linux is used.*