CS6106 - DATABASE MANAGEMENT SYSTEMS

Feb - May 2021

P2P(PEER-TO-PEER) BOOK RENTAL SYSTEM

KIRAN GEORGE GAURDIAN – 2019103029

NANDA KUMAR R – 2019103545

KAUSHIK NARAYAN R – 2019103536

YEAR – 2nd

SEMESTER – 4th

BATCH – Q

Table of Contents

ABSTRACT

In today's world, the sheer number of solutions to ease one's daily life is ever increasing, and with no doubt, it also includes online book rental systems in general. Therefore, we have come up with a solution in order to solve the problems of readers, particularly students.

The idea for such a solution stemmed from the fact that readers feel that certain books in their possession incapable of being re-read or stored are rendered obsolete in their eyes. While such things are common amongst the majority of people, on the other hand, there are interested parties willing to read or re-read the same book, which would have been disposed of by the former group of people, thus contributing to overall wastage and unnecessary financial expenses. A free book rental system, whose integrity depends completely on the goodwill of the community using it, would effortlessly solve the problem by giving a lot of independence in the hands of the users themselves.

The uniquely distinguishable implementation of our project is the fact that this is a system based on goodwill of the students/readers and also that this rental system is free, i.e., there are no price tags on books. This will especially help in the case where one desperately strives to possess books for the sake of learning or studying, because we believe that knowledge is free and its consumption must not come at a price.

INTRODUCTION

The project, “P2P (Peer to Peer) Book Rental System” is developed to be a full stack web application, that maintains a record of users and their locations, books, transactions, requests, and offers, along with added features which will be discussed shortly.

The following is the stack that is implemented for this project.

Front-End: ReactJS v17.0.2

Back-End: Express v4.17.1

Database Tools: PostgreSQL v13.2, pgAdmin4 v5.2 (for initial setup), PostgreSQL CLI (Command Line Interface)

Database Server: PostgreSQL Server 13

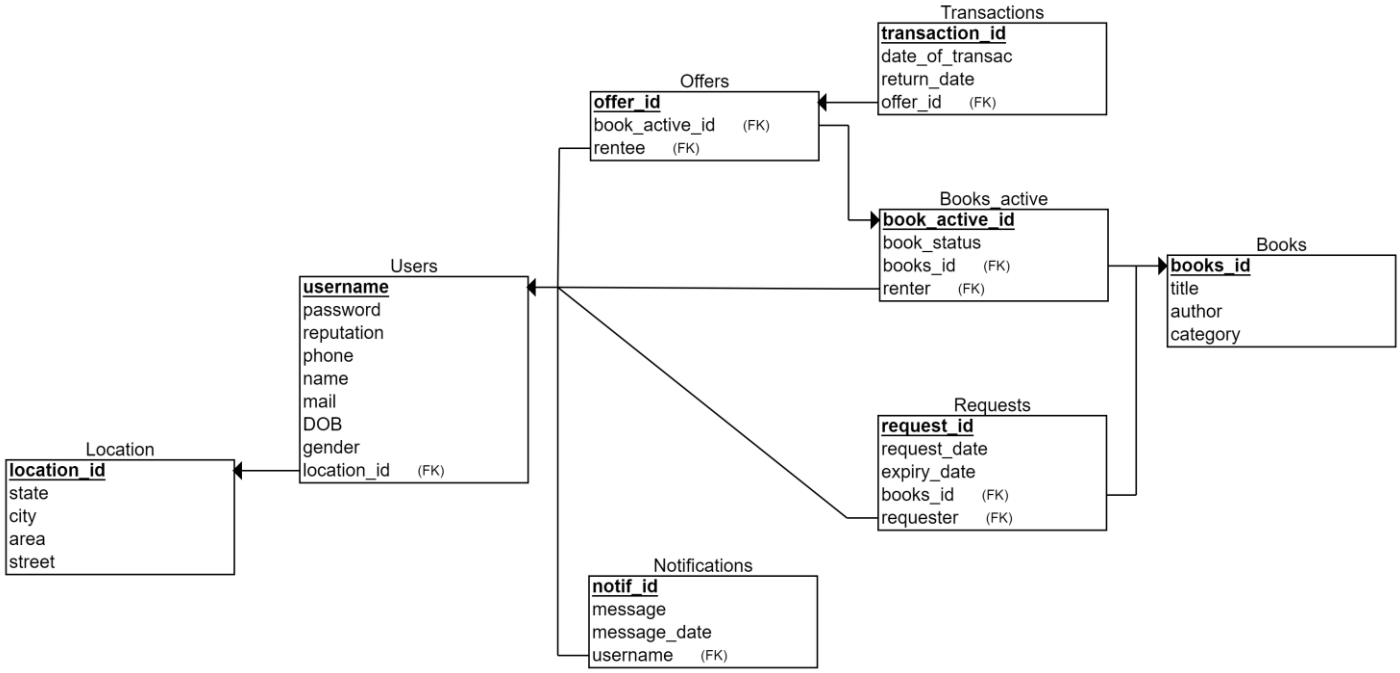
Functionalities and features of the System:

The system maintains:

- data of the users, who may be a buyer or seller or both
- data about the details of transactions that happen between the buyer and the seller
- static data of the location of the user
- data of the books that may or may not be put up for sale, including that regarding the status of the books that have been active in circulation
- a record of offers for books in circulation that have been made by buyers, which may or may not be accepted by the seller
- a record of requests made by buyers for a particular book
- The application has a **reputation system**, which is a system that formalizes the process of gathering and distributing information about a user's past behavior. With this system, depending on the number of transactions the user has participated in, the reputation of the user may increase or decrease.

Another feature is a **request system**, where a buyer can request for a particular book that may not be recorded as part of the database at that instance of time, but the request may be fulfilled at a later point in time by a seller.

Relational schema



Database Relations:

1) **Users:**

Attributes:

- username - primary key that uniquely identifies a user
- password, reputation, phone, name, mail, DOB, gender
- **location_id** – foreign key that references **location_id** attribute in the relation, **"Location"**

2) **Location:**

Attributes:

- location_id – primary key that uniquely identifies the location for a user
- state, city, area, street – non-prime attributes

3) **Books:**

Attributes:

- books_id – primary key that uniquely identifies a particular book
- title, author, category – non-prime attributes

4) **Books_active:**

Attributes:

- book_active_id – primary key that uniquely identifies an active book in circulation
- book_status - non-prime attribute
- **books_id** - foreign key that references **books_id** attribute in the relation, **"Books"**
- **renter** – foreign key that references **username** attribute in the relation, **"Users"**

5) **Transactions:**

Attributes:

- transaction_id – primary key that uniquely identifies a transaction between a buyer and a seller
- date_of_transac, return_date – non-prime attributes
- **offer_id** – foreign key that references **offer_id** attribute in the relation, **"Offers"**

6) **Offers:**

Attributes:

- offer_id - primary key that uniquely identifies an offer made by a renter for a book in circulation
- **book_active_id** – foreign key that references **book_active_id** attribute in the relation, **"Books_active"**
- **rentee** – foreign key that references **requester** attribute in the relation, **"Requests"**

7) **Requests:**

Attributes:

- request_id – primary key that uniquely identifies a request made by a buyer

- request_date, expiry_date – non-prime attributes
- **books_id** – foreign key that references **books_id** attribute in the relation, **“Books”**
- **requester** – foreign key that references **username** attribute in the relation, **“Users”**

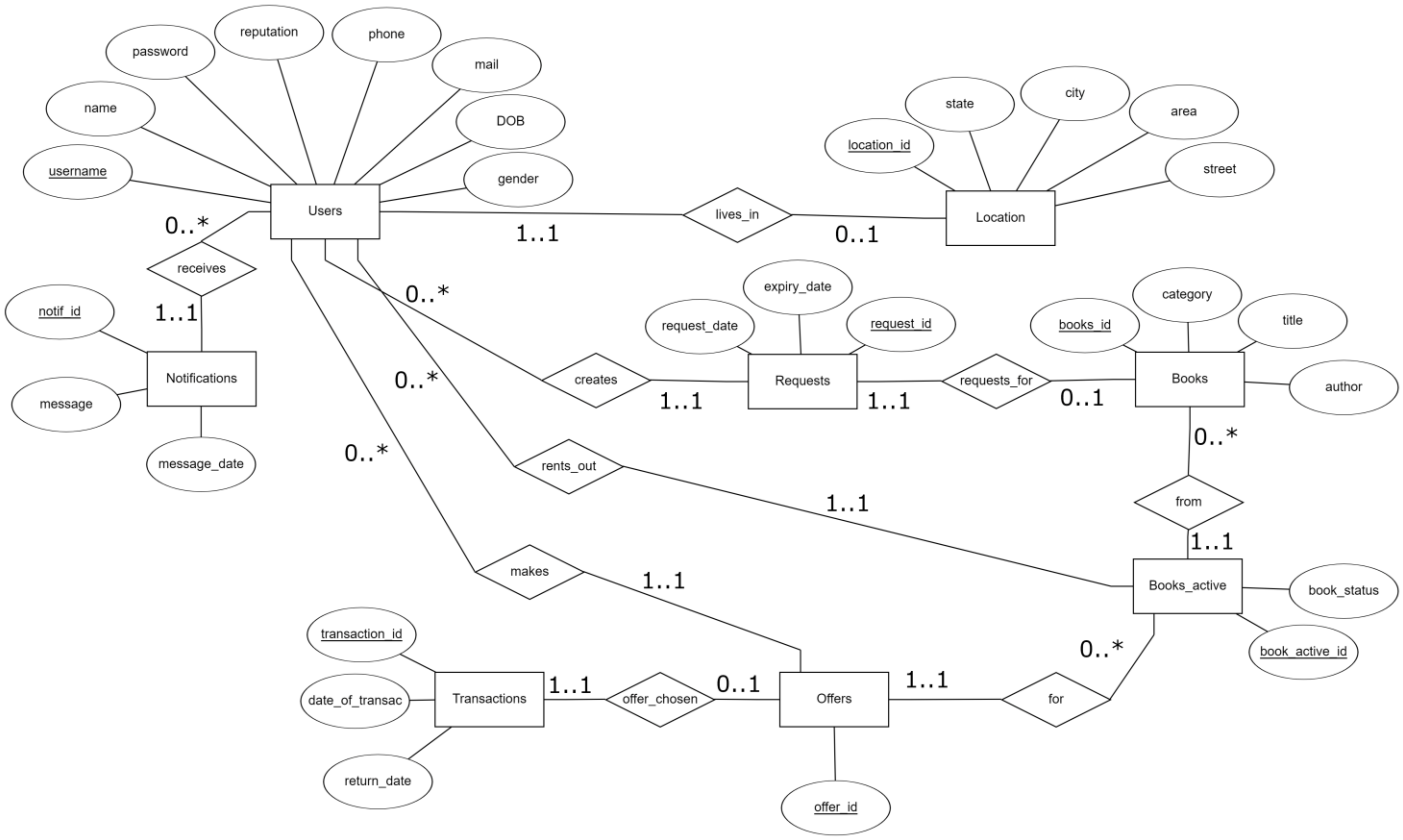
8) **Notification:**

Attributes:

- notif_id - primary key that uniquely identifies a notification meant to be viewed by the user
- **username –** foreign key that references **username** attribute in the relation, **“Users”**
- message, message_date - non-prime attributes

All the relations under the database are verified to be **Boyce-Codd Normalized.**

Entity – Relationship Diagram


password
reputation
phone
mail
name
DOB
username
gender
Users
lives_in
Location
state
city
area
location_id
street
0..*
receives
1..1
1..1
0..1
notif_id
Notifications
message
message_date
0..*
creates
Requests
request_date
expiry_date
request_id
1..1
1..1
requests_for
0..1
Books
books_id
category
title
author
0..*
rents_out
1..1
from
1..1
0..*
0..*
makes
1..1
Books_active
book_status
book_active_id
transaction_id
date_of_transac
Transactions
1..1
offer_chosen
0..1
Offers
1..1
for
0..*
return_date
offer_id

IMPLEMENTATION

Requirements: npm, PostgreSQL

Note: The front end has not been touched upon in this document.

1. POSTGRESQL DATABASE SETUP

Quick setup:

- After setting up PostgreSQL, open a terminal in the SQL_Files folder
- Connect to the psql shell and log in, then execute the following command.

  \i sql_exec.sql

```
\i db_schema_create.sql
\i rel_create.sql
\i trig_func_create.sql
\i trig_create.sql
\i proc_create.sql
```

Fig 1.1

(This is a psql shell script that combines all the below setup actions)

Manual setup:

- Database and schema setup

File: db_schema_create.sql

```
DROP DATABASE IF EXISTS p2p_books;
CREATE DATABASE p2p_books;
\c p2p_books;
CREATE SCHEMA IF NOT EXISTS p2p_books_schema;
ALTER DATABASE p2p_books SET search_path TO p2p_books_schema, public;
```

Fig 1.2

First, the database **'p2p_books'** is created from scratch, and then the database is connected to. Then the schema **'p2p_books_schema'** is created for the database. Note that this script uses a shell command to facilitate ease of setup.

To execute in psql shell: **\i db_schema_create.sql**

- Relations

File: rel_create.sql

```
CREATE SEQUENCE IF NOT EXISTS location_pk_seq START 10000;
CREATE SEQUENCE IF NOT EXISTS books_pk_seq START 20000;
CREATE SEQUENCE IF NOT EXISTS books_act_pk_seq START 30000;
CREATE SEQUENCE IF NOT EXISTS offers_pk_seq START 40000;
CREATE SEQUENCE IF NOT EXISTS requests_pk_seq START 50000;
CREATE SEQUENCE IF NOT EXISTS transacs_pk_seq START 60000;
CREATE SEQUENCE IF NOT EXISTS notif_pk_seq START 70000;

CREATE TABLE IF NOT EXISTS location (
    location_id INT NOT NULL DEFAULT nextval('location_pk_seq'),
    state VARCHAR(255) NOT NULL,
    city VARCHAR(255) NOT NULL,
    area VARCHAR(255) NOT NULL,
    street VARCHAR(255) NOT NULL,
    PRIMARY KEY (location_id)
);

CREATE TABLE IF NOT EXISTS users (
    username VARCHAR(255) NOT NULL,
    password VARCHAR(100) NOT NULL,
    name VARCHAR(255) NOT NULL,
    reputation NUMERIC(4,2) NOT NULL DEFAULT 10.00,
    phone VARCHAR(255) NOT NULL,
    mail VARCHAR(255) UNIQUE NOT NULL,
    DOB DATE NOT NULL,
    gender CHAR(1) NOT NULL CHECK (gender IN ('M', 'F', 'O')),
    location_id INT NOT NULL,
    PRIMARY KEY (username),
    FOREIGN KEY (location_id) REFERENCES location(location_id)
);
```

Fig 1.3

```sql
CREATE TABLE IF NOT EXISTS books (
    books_id INT NOT NULL DEFAULT nextval('books_pk_seq'),
    title VARCHAR(255) NOT NULL,
    author VARCHAR(255) NOT NULL,
    category VARCHAR(255) NOT NULL,
    PRIMARY KEY (books_id)
);

CREATE TABLE IF NOT EXISTS books_active (
    book_active_id INT NOT NULL DEFAULT nextval('books_act_pk_seq'),
    book_status CHAR(1) NOT NULL CHECK (book_status IN ('A', 'R', 'N')),
    books_id INT NOT NULL,
    owner VARCHAR(255) NOT NULL,
    PRIMARY KEY (book_active_id),
    FOREIGN KEY (books_id) REFERENCES books(books_id),
    FOREIGN KEY (owner) REFERENCES users(username)
);

CREATE TABLE IF NOT EXISTS offers (
    offer_id INT NOT NULL DEFAULT nextval('offers_pk_seq'),
    book_active_id INT NOT NULL,
    renter VARCHAR(255) NOT NULL,
    PRIMARY KEY (offer_id),
    FOREIGN KEY (book_active_id) REFERENCES books_active(book_active_id),
    FOREIGN KEY (renter) REFERENCES users(username)
);

CREATE TABLE IF NOT EXISTS requests (
    request_id INT NOT NULL DEFAULT nextval('requests_pk_seq'),
    request_date DATE NOT NULL,
    expiry_date DATE NOT NULL,
    books_id INT NOT NULL,
    requester VARCHAR(255) NOT NULL,
    PRIMARY KEY (request_id),
    FOREIGN KEY (books_id) REFERENCES books(books_id),
    FOREIGN KEY (requester) REFERENCES users(username)
);
```

Fig 1.4

```
CREATE TABLE IF NOT EXISTS transactions (
    transaction_id INT NOT NULL DEFAULT nextval('transacs_pk_seq'),
    date_of_transac DATE NOT NULL,
    return_date DATE NOT NULL,
    offer_id INT NOT NULL,
    PRIMARY KEY (transaction_id),
    FOREIGN KEY (offer_id) REFERENCES offers(offer_id)
);

CREATE TABLE IF NOT EXISTS notification (
    notif_id INT NOT NULL DEFAULT nextval('notif_pk_seq'),
    username VARCHAR(255) NOT NULL,
    message VARCHAR(300) NOT NULL,
    message_date DATE NOT NULL DEFAULT CURRENT_DATE,
    PRIMARY KEY(notif_id),
    FOREIGN KEY (username) REFERENCES users(username)
);
```

Fig 1.5

In this file, seven non-cycling sequences are created for use as primary key attribute values in the relations. Then eight relations are created as explained in the introduction. The fields of the relations have various constraints as seen. Fields with string data types commonly have a 255-character length limit, with exceptions such as the message field in the **'notification'** relation, and the password field in the **'users'** relation.

To execute in psql shell: **\i rel_create.sql**

- Triggers

PostgreSQL triggers execute stored trigger functions. Hence, first we create the trigger functions, and then the triggers.

File: trig_func_create.sql

```
CREATE OR REPLACE FUNCTION NewUserGreetProc()
    RETURNS TRIGGER AS $$
    BEGIN
        INSERT INTO notification (username, message)
        VALUES (NEW.username, 'Welcome to P2P Books! Share and borrow books for free!');
        RETURN NULL;
    END;
    $$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION RequestFulfilMessageProc()
    RETURNS TRIGGER AS $$

        DECLARE requester_name requests.requester%TYPE;
        DECLARE temp_bid books.books_id%TYPE;
        DECLARE req_book books.title%TYPE;
    BEGIN

        FOR temp_bid, requester_name IN (SELECT books_id, requester FROM requests) LOOP
            IF (temp_bid = NEW.books_id) AND (NEW.book_status = 'A') THEN
                SELECT b.title INTO req_book FROM books b INNER JOIN books_active ba
                ON b.books_id = ba.books_id AND ba.books_id = temp_bid;

                INSERT INTO notification (username, message)
                VALUES(requester_name, 'Your request for "' ||req_book|| '" has been fulfilled. Head to the home page to make an offer.');
            END IF;
        END LOOP;
        RETURN NULL;
    END;
    $$ LANGUAGE plpgsql;
```

Fig 1.6

```
CREATE OR REPLACE FUNCTION OwnerOfferMessageProc()
    RETURNS TRIGGER AS $$

        DECLARE owner_name books_active.owner%TYPE;
        DECLARE owner_book books.title%TYPE;
    BEGIN
        SELECT ba.owner, b.title INTO owner_name, owner_book FROM offers o INNER JOIN books_active ba
        ON o.book_active_id = ba.book_active_id AND o.book_active_id = NEW.book_active_id
        INNER JOIN books b ON ba.books_id = b.books_id;

        INSERT INTO notification (username, message)
        VALUES(owner_name, 'New offer for your book, "' ||owner_book|| '"');
        RETURN NULL;
    END;
    $$ LANGUAGE plpgsql;
```

Fig 1.7

```
CREATE OR REPLACE FUNCTION TransactOfferCleanupProc()
    RETURNS TRIGGER AS $$

        DECLARE final_bai books_active.book_active_id%TYPE;
        DECLARE final_renter users.username%TYPE;
        DECLARE temp_doi offers.offer_id%TYPE;
        DECLARE user_name offers.renter%TYPE;
        DECLARE b_title books.title%TYPE;

    BEGIN
        SELECT book_active_id, renter INTO final_bai, final_renter FROM offers
            WHERE offer_id = NEW.offer_id;

        SELECT b.title INTO b_title FROM books b INNER JOIN books_active ba ON
            ba.books_id = b.books_id AND ba.book_active_id = final_bai;

        FOR temp_doi IN (SELECT offer_id FROM offers
            WHERE book_active_id = final_bai AND renter != final_renter)
        LOOP
            SELECT renter INTO user_name FROM offers
                WHERE offer_id = temp_doi;
            INSERT INTO notification (username, message) VALUES (user_name, 'Your offer for "' ||b_title|| '" was rejected.');
            DELETE FROM offers WHERE offer_id = temp_doi;
        END LOOP;

        INSERT INTO notification (username, message) VALUES (final_renter, 'Your offer for "' ||b_title|| '" was accepted.');
        RETURN NULL;
    END;
    $$ LANGUAGE plpgsql;
```

Fig 1.8

```
CREATE OR REPLACE FUNCTION UserReputationUpdateProc()
    RETURNS TRIGGER AS $$
    BEGIN
        UPDATE users SET reputation = reputation + 2 WHERE username =
            (SELECT owner FROM books_active WHERE book_active_id =
                (SELECT book_active_id FROM offers WHERE offer_id = NEW.offer_id)
            );

        UPDATE users SET reputation = reputation - 1 WHERE username =
                (SELECT renter FROM offers WHERE offer_id = NEW.offer_id);
        RETURN NULL;
    END;
    $$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION TransactOverCleanupProc()
    RETURNS TRIGGER AS $$

        DECLARE returned_ti transactions.transaction_id%TYPE;
        DECLARE returned_oi offers.offer_id%TYPE;

    BEGIN
        IF (OLD.book_status = 'R') AND (NEW.book_status != 'R') THEN
            SELECT transaction_id INTO returned_ti FROM transactions WHERE offer_id =
                (SELECT offer_id FROM offers WHERE book_active_id = OLD.book_active_id);
            SELECT offer_id INTO returned_oi FROM offers WHERE book_active_id = OLD.book_active_id;

            DELETE FROM transactions WHERE transaction_id = returned_ti;
            DELETE FROM offers WHERE offer_id = returned_oi;
        END IF;
        RETURN NULL;
    END;
    $$ LANGUAGE plpgsql;
```

Fig 1.9

This file contains the definitions of the trigger functions used for triggers. The purposes of these functions are outlined below:

1. Greet a new user.
2. Notify a user if a book they requested for has been put in circulation.
3. Notify the owner of a book in circulation when an offer to share the book has been made.

4. When an offer for a book in circulation is accepted, delete all the rejected offers, and notify the users who made those offers, as well as the user whose offer got accepted.
5. When an offer is accepted, update the reputation score of the book's owner and borrower accordingly.
6. Once the book owner marks the book as returned, delete that accepted offer and the record of the transaction as well.

To execute in psql shell: **\i trig_func_create.sql**

File: trig_create.sql

```
DROP TRIGGER IF EXISTS NewUserGreetTrig_ai ON users;
DROP TRIGGER IF EXISTS RequestFulfilMessageTrig_aiu ON books_active;
DROP TRIGGER IF EXISTS OwnerOfferMessageTrig_ai ON offers;
DROP TRIGGER IF EXISTS TransactOfferCleanupTrig_ai ON transactions;
DROP TRIGGER IF EXISTS UserReputationUpdateTrig_ai ON transactions;
DROP TRIGGER IF EXISTS TransactOverCleanupTrig_au ON books_active;

CREATE TRIGGER NewUserGreetTrig_ai
    AFTER INSERT ON users
    FOR EACH ROW
    EXECUTE PROCEDURE NewUserGreetProc();

CREATE TRIGGER RequestFulfilMessageTrig_aiu
    AFTER INSERT OR UPDATE ON books_active
    FOR EACH ROW
    EXECUTE PROCEDURE RequestFulfilMessageProc();

CREATE TRIGGER OwnerOfferMessageTrig_ai
    AFTER INSERT ON offers
    FOR EACH ROW
    EXECUTE PROCEDURE OwnerOfferMessageProc();

CREATE TRIGGER TransactOfferCleanupTrig_ai
    AFTER INSERT ON transactions
    FOR EACH ROW
    EXECUTE PROCEDURE TransactOfferCleanupProc();

CREATE TRIGGER UserReputationUpdateTrig_ai
    AFTER INSERT ON transactions
    FOR EACH ROW
    EXECUTE PROCEDURE UserReputationUpdateProc();

CREATE TRIGGER TransactOverCleanupTrig_au
    AFTER UPDATE ON books_active
    FOR EACH ROW
    EXECUTE PROCEDURE TransactOverCleanupProc();
```

Fig 1.10

This file contains the definitions for the triggers themselves. The triggers execute the corresponding trigger functions on certain actions at certain times on the specified relations, such as after inserting or updating a table.

To execute in psql shell: **\i trig_create.sql**

- Stored procedures

File: proc_create.sql

```
CREATE OR REPLACE PROCEDURE RequestCleanProc()
AS $$
DECLARE temp_dri requests.request_id%TYPE;
DECLARE temp_dru requests.requester%TYPE;
DECLARE temp_b_title books.title%TYPE;
BEGIN
    FOR temp_dri IN (SELECT request_id FROM requests
        WHERE expiry_date < CURRENT_DATE)
    LOOP
        SELECT requester INTO temp_dru FROM requests
            WHERE request_id = temp_dri;
        SELECT b.title INTO temp_b_title FROM books b
            INNER JOIN requests r ON r.books_id = b.books_id AND r.request_id = temp_dri;
        INSERT INTO notification (username, message)
        VALUES (temp_dru, 'Your request for "' ||temp_b_title|| '" has expired.');
        DELETE FROM requests WHERE request_id = temp_dri;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Fig 1.11

This file contains the definition for a stored procedure that deletes book requests that have expired, and notifies the requester about the same.

To execute in psql shell: **\i proc_create.sql**

2. BACKEND IMPLEMENTATION

The backend portion of our site was built using ExpressJS, a Node.js web application framework.

2.0 Package setup

Open a terminal in the Server_files folder, and run the command:

npm install

This will install the dependency packages as specified in the package.json file which are required for the back end.

2.1 Middleware

These are functions that have access to the request object, the response object, and the next middleware function in the application's request-response cycle and are used to modify req and res objects.

Middleware to check for missing credentials -

If any credential is found to be missing, the server responds with a "Missing Credentials" error instead of executing the succeeding middleware function.

```
function testMissing(req, res, next) {
  const {
    username, password, name, phone, mail, dob, gender, state, city, area, street,
  } = req.body;
  if (req.path === "/register") {
    if (
      ![
        username, password, name, phone, mail, dob, gender, state, city, area, street,
      ].every(Boolean)
    ) {
      return res.json("Missing Credentials");
    }
  } else if (req.path === "/login") {
    if (![username, password].every(Boolean)) {
      return res.json("Missing Credentials");
    }
  }
  next();
}
```

Fig 2.1.1

Middleware to validate a JSON Web Token -

The token sent in the request header is verified and the username obtained is added to the request body. A response error message indicating that the token is invalid is sent in the event that the token could not be verified.

```
function validTokenTest(req, res, next) {
  const token = req.header("token");
  console.log(token);
  if (!token) {
    return res.status(403).json({ msg: "authorization denied" });
  }

  try {
    const verify = jwt.verify(token, `${process.env.jwtSecret}`);
    req.user = verify.user;
    next();
  } catch (err) {
    res.status(401).json({ msg: "Token is not valid" });
  }
}
```

Fig 2.1.2

Middleware to call RequestCleanProc procedure -

This middleware is used in all routes that deal with request components.

```
const deleteExpiredRequests = async (req, res, next) => {
  await db.query("BEGIN");
  try {
      console.log("cleaning requests...");
      await db.query("CALL RequestCleanProc()");
      await db.query("COMMIT");
      next();
  } catch (error) {
    console.error(error);
    await db.query("ROLLBACK");
  }
};
```

Fig 2.1.3

2.2 Backend routing

Routes dealing with user relation

Route to register a new user - The new username is first checked to ensure that it is unique. The email is then regex tested. When all credentials are found to be valid, the password is hashed using bcrypt and the new user is recorded into the database. A token is generated using the username and sent to the frontend as response.

```
router.post("/register", credCheck, async (req, res) => {
  await db.query("BEGIN");

  try {
    const {
      username, password, name, phone, mail, dob, gender, state, city, area, street,
    } = req.body;
    console.log(name);

    const user = await db.query(
      "SELECT * FROM users WHERE username = $1\
      UNION SELECT * FROM users WHERE mail = $2",
      [username, mail]
    );

    if (user.rows.length > 0) {
      return res.status(401).json({
        status: "failure",
        msg: "User already exists!",
      });
    }

    const re = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    if (!re.test(mail))
      return res.status(401).json({
        status: "failure",
        msg: "Invalid email",
      });

    const salt = await bcrypt.genSalt(10);
    const bcryptPassword = await bcrypt.hash(password, salt);

    const newLoc = await db.query(
      "INSERT INTO location (state, city, area, street) VALUES ($1, $2, $3, $4) RETURNING *",
      [state, city, area, street]
    );
    const newUser = await db.query(
      "INSERT INTO users (username, password, name, phone, mail, dob, gender, location_id)\
      VALUES ($1, $2, $3, $4, $5, $6, $7, $8) RETURNING *",
      [
        username, bcryptPassword, name, phone, mail, dob, gender, newLoc.rows[0].location_id,
      ]
    );

    const jwtToken = jwtGenerator(newUser.rows[0].username);

    await db.query("COMMIT");

    return res.status(201).json({ jwtToken });

  } catch (error) {
    console.error(error);
    await db.query("ROLLBACK")
  }
});
```

Fig 2.2.1

Route to log in an existing user - If the input username and password are found to be valid, a token is generated and sent to the frontend. Any subsequent request that requires the user to be authenticated should be sent with this token.

```
router.post("/login", credCheck, async (req, res) => {
  await db.query("BEGIN");
  try {
    const { username, password } = req.body;
    var user = await db.query("SELECT * FROM users WHERE username = $1", [
      username,
    ]);

    if (user.rows.length === 0)
      user = await db.query("SELECT * FROM users WHERE mail = $1", [username]);

    if (user.rows.length === 0)
      return res.status(401).json({
        status: "failure",
        msg: "User does not exist",
      });

    console.log(username);

    const validPassword = await bcrypt.compare(password, user.rows[0].password);

    if (!validPassword) {
      return res.status(401).json({
        status: "failure",
        msg: "Invalid username or password",
      });
    }
    const jwtToken = jwtGenerator(user.rows[0].username);

    await db.query("COMMIT");

    return res.json({ jwtToken });
  } catch (error) {
    console.error(error);
    await db.query("ROLLBACK");
  }
});
```

Fig 2.2.2

Route to update user details -

```
router.put("/", tokenCheck, async (req, res) => {
  await db.query("BEGIN");

  try {

    const username = req.user;
    const { name, phone, mail, state, city, area, street } = req.body;

    const re = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    if (!re.test(mail))
      return res.status(401).json({
        status: "failure",
        msg: "Invalid email"
      });

    let updatedLoc = await db.query(
      "UPDATE location SET state = $1, city = $2, area = $3, street=$4 WHERE\
        location_id = (SELECT location_id FROM users WHERE username = $5)",
      [state, city, area, street, username]
    );

    let updatedUser = await db.query(
      "UPDATE users SET name=$1, phone=$2, mail=$3 WHERE username = $4",
      [name, phone, mail, username]
    );

    await db.query("COMMIT");

    res.status(201).json({
      status: "success",
    });
  } catch (error) {
    console.error(error);
    await db.query("ROLLBACK");
  }
});
```

Fig 2.2.3

Route to get the username and location details of a user -

```
router.get("/", tokenCheck, async (req, res) => {
  try {
    const user = await db.query(
      "SELECT u.*, l.* FROM users u INNER JOIN location l ON u.location_id = l.location_id\
      AND username = $1",
      [req.user]
    );
    res.json(user.rows[0]);
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.4

"Register", "Login" and "my-details" pages

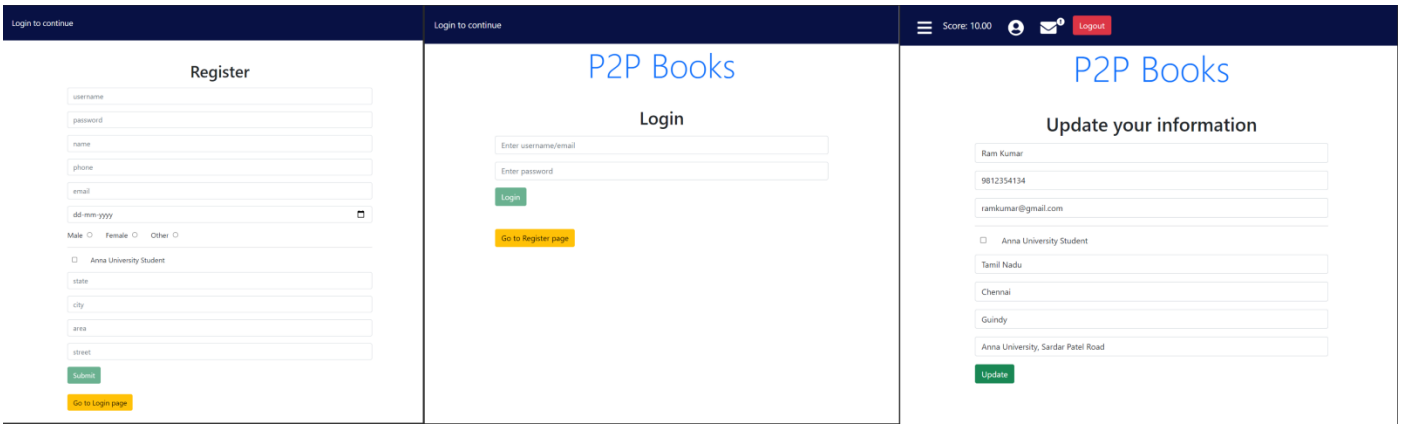


Fig 2.2.5

Routes dealing with books_active

Route to get all active books -

```
router.get("/", async (req, res) => {
  try {
    console.log("initiating get request for all active books...");
    const get_result = await db.query(
      "SELECT ba.book_active_id, ba.owner, b.* FROM books b\
      INNER JOIN books_active ba ON ba.books_id = b.books_id AND ba.book_status='A'"
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        Books: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.6

Route to get all active books of a user -

```
router.get("/profile", tokenCheck, async (req, res) => {
  try {
    console.log("initiating get request for user's active books...");
    const user = req.user;
    const get_result = await db.query(
      "SELECT t.transaction_id, t.return_date, ba.book_active_id, ba.book_status, o.renter, b.* FROM books b\
      INNER JOIN books_active ba ON ba.books_id = b.books_id AND ba.owner = $1\
      LEFT JOIN offers o ON o.book_active_id = ba.book_active_id\
      LEFT JOIN transactions t ON t.offer_id = o.offer_id",
      [user]
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        Books: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.7

Route to get borrowed books of a user -

```
router.get("/profile/borrowed", tokenCheck, async (req, res) => {
  try {
    console.log("initiating request for user's borrowed books...");
    const user = req.user;
    const get_result = await db.query(
      "SELECT t.return_date, ba.book_active_id, ba.owner, b.title, u.name, u.phone, u.mail FROM books b\
      INNER JOIN books_active ba ON ba.books_id = b.books_id AND ba.book_status='R'\
      INNER JOIN offers o ON o.book_active_id = ba.book_active_id AND o.renter = $1\
      INNER JOIN users u ON u.username = ba.owner\
      INNER JOIN transactions t ON t.offer_id = o.offer_id",
      [user]
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        Books: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.8

Route for filtering active books based on user input for author, title, category, state, city, area and street -

```
router.get("/filter", async (req, res) => {
  try {
    console.log("initiating get request for filtered active books...");
    const {
      search_title,
      search_author,
      search_category,
      search_state,
      search_city,
      search_area,
      search_street,
    } = req.query;

    var search_method = 0;
    if (search_title) search_method = search_method + 1;
    if (search_author) search_method = search_method + 2;
    if (search_category && search_category != "all")
      search_method = search_method + 4;

    var get_result;
    switch (search_method) {
      case 0:
        get_result = await db.query(
          "SELECT ba.*, b.title, b.author, b.category, u.location_id, l.state, l.city, l.area, l.street FROM books_active ba\
          INNER JOIN books b ON ba.books_id = b.books_id\
          INNER JOIN users u on u.username = ba.owner\
          INNER JOIN location l ON l.location_id = u.location_id\
          WHERE book_status='A'"
        );
        break;
      case 1:
        get_result = await db.query(
          "SELECT ba.*, b.title, b.author, b.category, u.location_id, l.state, l.city, l.area, l.street FROM books_active ba\
          INNER JOIN books b ON ba.books_id = b.books_id\
          INNER JOIN users u on u.username = ba.owner\
          INNER JOIN location l ON l.location_id = u.location_id\
          WHERE LOWER(title) ~ LOWER($1) AND book_status='A'",
          [search_title]
        );
        break;
      case 2:
        get_result = await db.query(
          "SELECT ba.*, b.title, b.author, b.category, u.location_id, l.state, l.city, l.area, l.street FROM books_active ba\
          INNER JOIN books b ON ba.books_id = b.books_id\
          INNER JOIN users u on u.username = ba.owner\
          INNER JOIN location l ON l.location_id = u.location_id\
          WHERE LOWER(author) ~ LOWER($1) AND book_status='A'",
          [search_author]
        );
        break;
      case 3:
        get_result = await db.query(
          "SELECT ba.*, b.title, b.author, b.category, u.location_id, l.state, l.city, l.area, l.street FROM books_active ba\
          INNER JOIN books b ON ba.books_id = b.books_id\
          INNER JOIN users u on u.username = ba.owner\
          INNER JOIN location l ON l.location_id = u.location_id\
          WHERE LOWER(title) ~ LOWER($1) AND LOWER(author) ~ LOWER($2) AND book_status='A'",
          [search_title, search_author]
        );
        break;
```

```
    case 4:
      get_result = await db.query(
        "SELECT ba.*, b.title, b.author, b.category, u.location_id, l.state, l.city, l.area, l.street FROM books_active ba\
        INNER JOIN books b ON ba.books_id = b.books_id\
        INNER JOIN users u on u.username = ba.owner\
        INNER JOIN location l ON l.location_id = u.location_id\
        WHERE LOWER(category) ~ LOWER($1) AND book_status='A'",
        [search_category]
      );
      break;
    case 5:
      get_result = await db.query(
        "SELECT ba.*, b.title, b.author, b.category, u.location_id, l.state, l.city, l.area, l.street FROM books_active ba\
        INNER JOIN books b ON ba.books_id = b.books_id\
        INNER JOIN users u on u.username = ba.owner\
        INNER JOIN location l ON l.location_id = u.location_id\
        WHERE LOWER(title) ~ LOWER($1) AND LOWER(category) = LOWER($2) AND book_status='A'",
        [search_title, search_category]
      );
      break;
    case 6:
      get_result = await db.query(
        "SELECT ba.*, b.title, b.author, b.category, u.location_id, l.state, l.city, l.area, l.street FROM books_active ba\
        INNER JOIN books b ON ba.books_id = b.books_id\
        INNER JOIN users u on u.username = ba.owner\
        INNER JOIN location l ON l.location_id = u.location_id\
        WHERE LOWER(author) ~ LOWER($1) AND LOWER(category) = LOWER($2) AND book_status='A'",
        [search_author, search_category]
      );
      break;
    case 7:
      get_result = await db.query(
        "SELECT ba.*, b.title, b.author, b.category, u.location_id, l.state, l.city, l.area, l.street FROM books_active ba\
        INNER JOIN books b ON ba.books_id = b.books_id\
        INNER JOIN users u on u.username = ba.owner\
        INNER JOIN location l ON l.location_id = u.location_id\
        WHERE LOWER(title) ~ LOWER($1) AND LOWER(author) ~ LOWER($2) AND LOWER(category) = LOWER($3) AND book_status='A'",
        [search_title, search_author, search_category]
      );
      break;
    default:
      throw "Bad GET request parameters";
  }
  var result_obj = { data: get_result.rows };
  var filtered_res = result_obj.data.filter(function (book) {
    return (
      (search_state === "all" ? true : book.state == search_state) &&
      (search_city === "all" ? true : book.city == search_city) &&
      (search_area === "all" ? true : book.area == search_area) &&
      (search_street === "all" ? true : book.street == search_street)
    );
  });
  console.log(filtered_res);
  res.status(201).json({
    status: "success",
    data: {
      Books: filtered_res,
    },
  });
 } catch (error) {
  console.error(error);
 }
});
```

Fig 2.2.9

Route to get categories of all currently active books -

```
router.get("/category", async (req, res) => {
  try {
    console.log("initiating get request for all active book categories...");
    const get_result = await db.query(
      "SELECT DISTINCT b.category FROM books b INNER JOIN books_active ba ON ba.books_id = b.books_id"
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        Categories: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
    res.status(400).json({
      status: "bad request",
    });
  }
});
```

Fig 2.2.10

Route to get categories and location details of all currently active books -

```
router.get("/sorting", async (req, res) => {
  try {
    const get_category = await db.query(
      "SELECT DISTINCT b.category FROM books b INNER JOIN books_active ba ON ba.books_id = b.books_id"
    );
    const get_state = await db.query(
      "SELECT DISTINCT l.state FROM location l INNER JOIN users u ON u.location_id=l.location_id\
       INNER JOIN books_active ba ON ba.owner = u.username"
    );
    const get_city = await db.query(
      "SELECT DISTINCT l.city FROM location l INNER JOIN users u ON u.location_id=l.location_id\
       INNER JOIN books_active ba ON ba.owner = u.username"
    );
    const get_area = await db.query(
      "SELECT DISTINCT l.area FROM location l INNER JOIN users u ON u.location_id=l.location_id\
       INNER JOIN books_active ba ON ba.owner = u.username"
    );
    const get_street = await db.query(
      "SELECT DISTINCT l.street FROM location l INNER JOIN users u ON u.location_id=l.location_id\
       INNER JOIN books_active ba ON ba.owner = u.username"
    );

    (get_state.rows.findIndex(x => x.state=="Tamil Nadu")) == -1 ? get_state.rows.push({ state : "Tamil Nadu"}) : console.log("State exists");
    (get_city.rows.findIndex(x => x.city=="Chennai")) == -1 ? get_city.rows.push({ city : "Chennai"}) : console.log("City exists");
    (get_area.rows.findIndex(x => x.area=="Guindy")) == -1 ? get_area.rows.push({ area : "Guindy"}) : console.log("Area exists");
    (get_street.rows.findIndex(x => x.street=="Anna University, Sardar Patel Road")) == -1 ?
      get_street.rows.push({ street : "Anna University, Sardar Patel Road"}) : console.log("Street exists");

    res.status(201).json({
      status: "success",
      data: {
        Categories: get_category.rows,
        States: get_state.rows,
        Cities: get_city.rows,
        Areas: get_area.rows,
        Streets: get_street.rows,
      },
    });
  } catch (error) {
    console.error(error);
    res.status(400).json({
      status: "bad request",
    });
  }
});
```

Fig 2.2.11

Route to put a book into circulation -

```
router.post("/", tokenCheck, async (req, res) => {
  try {
    const username = req.user;
    const { books_id } = req.body;

    const new_active_book = await db.query(
      "INSERT INTO books_active (book_status, books_id, owner)\
      VALUES ('A',$1,$2) RETURNING *",
      [books_id, username]
    );

    console.log(new_active_book.rows[0]);
    res.status(201).json({
      status: "success",
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.12

Route to update a book status -

```
router.put("/", tokenCheck, async (req, res) => {
  try {
    const { book_active_id, new_status } = req.body;
    console.log(req.body);
    const unav_book = await db.query(
      "UPDATE books_active SET book_status = $1 WHERE book_active_id = $2",
      [new_status, book_active_id]
    );
    console.log(unav_book.rows[0]);
    res.status(201).json({
      status: "success",
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.13

"Home" and "My Books" pages displaying all active books that can be rented and books that a user has rent out or borrowed respectively

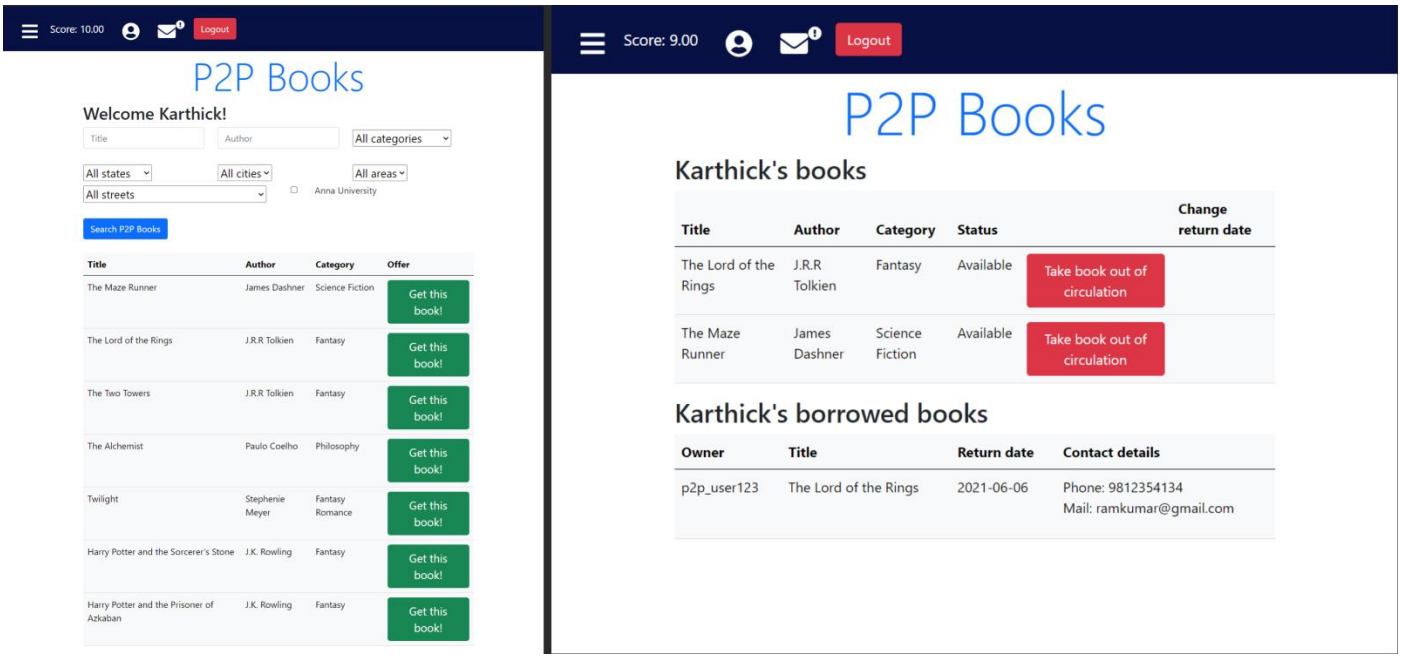


Fig 3.1

Routes dealing with offers relation

Route to get all offers made to a user sorted by book title -

```
router.get("/profile/owner", tokenCheck, async (req, res) => {
  try {
    console.log(req.user);
    const username = req.user;
    console.log("initiating get request for all offers...");
    const get_result = await db.query(
      "SELECT b.title, o.renter, u.reputation, o.offer_id FROM offers o\
      INNER JOIN books_active ba ON ba.book_active_id = o.book_active_id AND ba.owner = $1\
      INNER JOIN books b ON ba.books_id = b.books_id\
      INNER JOIN users u ON u.username=o.renter\
      WHERE o.offer_id NOT IN (SELECT t.offer_id FROM transactions t)\
      ORDER BY b.title", [username]);

    const bookwise_res = get_result.rows.reduce((acc, d) => {
      const found = acc.find(a => a.title === d.title);
      const offer = { offer_id: d.offer_id, renter: d.renter, reputation: d.reputation };
      if (!found) {
        acc.push({title:d.title, offers: [offer]})
      }
      else {
        found.offers.push(offer)
      }
      return acc;
    }, []);
    console.log(bookwise_res);

    res.status(201).json({
      status: "success",
      data: {
        Offer: bookwise_res,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.14

Route to get all offers made by a user -

```
router.get("/profile/renter", tokenCheck, async (req, res) => {
  try {
    const username = req.user;
    const get_result = await db.query(
      "SELECT b.title, ba.owner, o.offer_id from books b\
      INNER JOIN books_active ba ON b.books_id = ba.books_id\
      INNER JOIN offers o ON o.book_active_id = ba.book_active_id AND o.renter = $1\
      WHERE o.offer_id NOT IN (SELECT t.offer_id FROM transactions t)",
      [username]
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        Offer: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.15

Route to record offers - The offers relation is first checked to ensure that the user has not made an offer for the same book already.

```
router.post("/", tokenCheck, async (req, res) => {
  await db.query("BEGIN");

  try {
    const username = req.user;
    const book_active_id = req.body.book_active_id;
    const check_dup = await db.query(
      "SELECT * FROM offers WHERE book_active_id = $1 AND renter = $2\
      AND offer_id NOT IN (SELECT offer_id FROM transactions)",
      [book_active_id, username]
    );
    if (check_dup.rows.length > 0) {
      return res.status(400).json({
        status: "failure",
        msg: "You have already made this offer..."
      })
    }

    const get_result = await db.query(
      "INSERT INTO offers(book_active_id,renter) values($1,$2) RETURNING *",
      [book_active_id, username]
    );
    console.log(get_result.rows);

    await db.query("COMMIT");

    res.status(201).json({
      status: "success",
      data: {
        Offer: get_result.rows[0],
      },
    });
  } catch (error) {
    console.error(error);
    await db.query("ROLLBACK");
  }
});
```

Fig 2.2.16

Route to delete an offer -

```
router.delete("/", tokenCheck, async (req, res) => {
  try {
    const { offer_id } = req.query;
    const get_result = await db.query(
      "DELETE FROM offers WHERE offer_id = $1 RETURNING *",
      [offer_id]
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        Deleted_Offers: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});

module.exports = router;
```

Fig 2.2.17

"My offers" page displaying offer made to and by a user

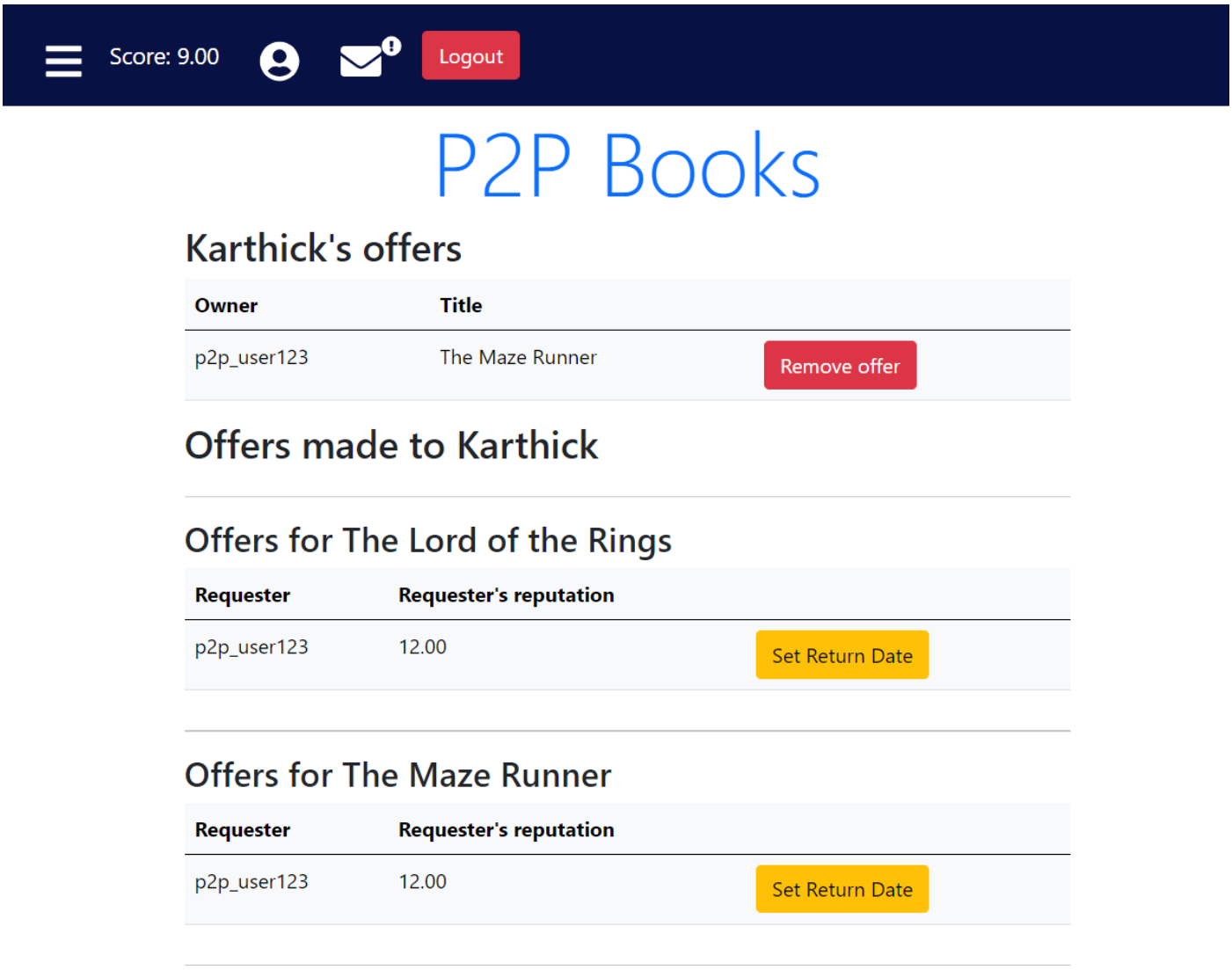


Fig 3.2

Routes dealing with requests relation

Route to get all requests -

```
router.get("/", expiredReqClean, async (req, res) => {
  try {
    console.log("initiating get request for all requests...");
    const get_result = await db.query(
      "SELECT r.request_id, r.requester, b.* FROM books b INNER JOIN requests r ON b.books_id = r.books_id"
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        reqBooks: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.18

Route to get requests made by a user -

```
router.get("/profile", tokenCheck, expiredReqClean, async (req, res) => {
  try {
    const username = req.user;
    const get_result = await db.query(
      "SELECT r.request_id, r.expiry_date, b.* from books b INNER JOIN requests r ON \
        b.books_id = r.books_id AND r.requester = $1",
      [username]
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        reqBooks: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.19

Route to record a request - The request relation is first checked to ensure that the user has not requested the same book already.

```
router.post("/", tokenCheck, expiredReqClean, async (req, res) => {
  await db.query("BEGIN");

  try {
    const username = req.user;
    const { books_id } = req.body;

    const request_check = await db.query(
      "SELECT * FROM requests WHERE books_id=$1 AND requester=$2",
      [books_id, username]
    );

    if (request_check.rows.length > 0) {
      return res.json({
        status: "failure",
      });
    }

    const new_request = await db.query(
      "INSERT INTO requests (request_date, expiry_date, books_id, requester)\
      VALUES (CURRENT_DATE, CURRENT_DATE + INTERVAL '7 day', $1, $2) RETURNING *",
      [books_id, username]
    );

    console.log(new_request.rows[0]);

    await db.query("COMMIT");

    res.status(201).json({
      status: "success",
    });
  } catch (error) {
    console.error(error);
    await db.query("ROLLBACK");
  }
});
```

Fig 2.2.20

Route to delete a request -

```
router.delete("/", tokenCheck, expiredReqClean, async (req, res) => {
  try {
    const { request_id } = req.query;
    const get_result = await db.query(
      "DELETE FROM requests WHERE request_id=$1 RETURNING *",
      [request_id]
    );
    console.log(get_result.rows[0]);
    res.status(201).json({
      status: "success",
      data: {
        Deleted_request: get_result.rows[0],
      },
    });
  } catch (error) {
    console.error(error);
  }
});
module.exports = router;
```

Fig 2.2.21

"P2P Requests" and "My Requests" pages displaying all requests made and a user's requests respectively

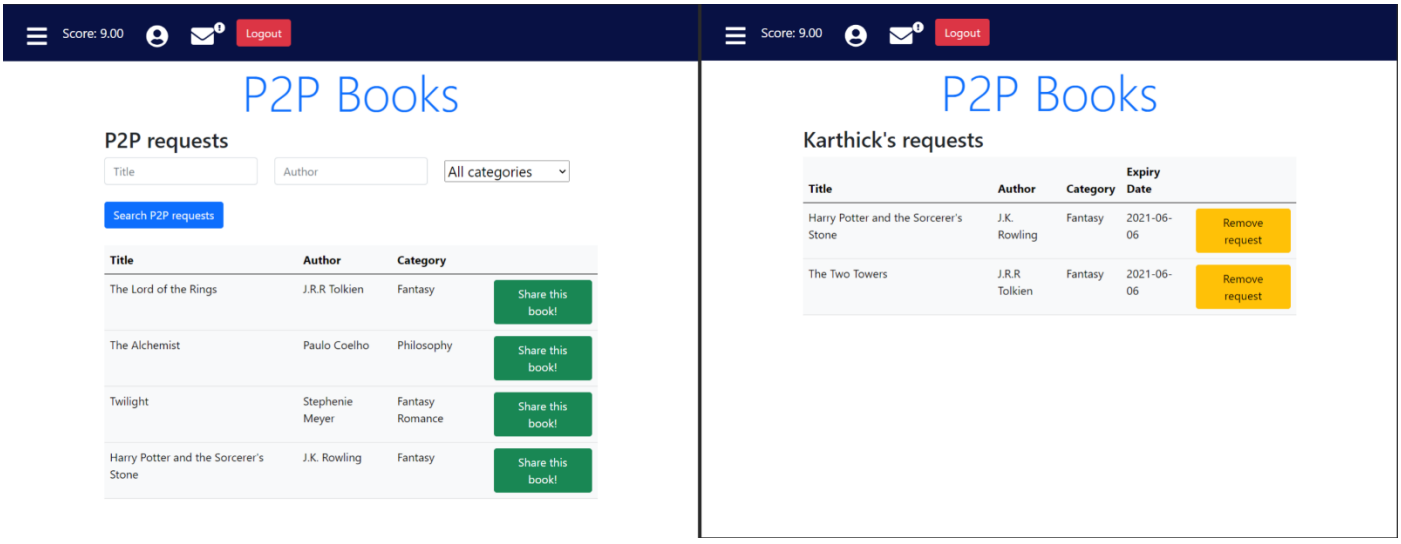


Fig 3.3

Routes dealing with books relation

Route to get all books -

```
router.get("/", tokenCheck, async (req, res) => {
  try {
    console.log(req.user);
    console.log("initiating get request for all books...");
    const get_result = await db.query("SELECT * FROM books");
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        Books: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.22

Route to filter books based on user input for author, title and category -

```
router.get("/filter", tokenCheck, async (req, res) => {
  try {
    console.log("initiating get request for filtered books...");
    const { search_title, search_author, search_category } = req.query;

    var search_method = 0;
    if (search_title) search_method = search_method + 1;
    if (search_author) search_method = search_method + 2;
    if (search_category && search_category != "all")
      search_method = search_method + 4;

    var get_result;
    switch (search_method) {
      case 0:
        get_result = await db.query("SELECT b.* FROM books b");
        break;
      case 1:
        get_result = await db.query(
          "SELECT b.* FROM books b WHERE (LOWER(b.title) ~ LOWER($1))",
          [search_title]
        );
        break;
      case 2:
        get_result = await db.query(
          "SELECT b.* FROM books b WHERE (LOWER(b.author) ~ LOWER($1))",
          [search_author]
        );
        break;
      case 3:
        get_result = await db.query(
          "SELECT b.* FROM books b WHERE (LOWER(b.title) ~ LOWER($1) AND LOWER(b.author) ~ LOWER($2))",
          [search_title, search_author]
        );
        break;
      case 4:
        get_result = await db.query(
          "SELECT b.* FROM books b WHERE (LOWER(b.category) = LOWER($1))",
          [search_category]
        );
        break;
      case 5:
        get_result = await db.query(
          "SELECT b.* FROM books b WHERE (LOWER(b.title) ~ LOWER($1) AND LOWER(b.category) = LOWER($2))",
          [search_title, search_category]
        );
        break;
      case 6:
        get_result = await db.query(
          "SELECT b.* FROM books b WHERE (LOWER(b.author) ~ LOWER($1) AND LOWER(b.category) = LOWER($2))",
          [search_author, search_category]
        );
        break;
      case 7:
        get_result = await db.query(
          "SELECT b.* FROM books b WHERE (LOWER(b.title) ~ LOWER($1) AND LOWER(b.author) ~ LOWER($2) AND LOWER(b.category) = LOWER($3))",
          [search_title, search_author, search_category]
        );
        break;
      default: throw "Bad GET request parameters";
    }
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        Books: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.23

Route to add a new book -

```
router.post("/", tokenCheck, async (req, res) => {
  await db.query("BEGIN");

  try {
    console.log(req.user);
    const { title, author, category } = req.body;
    const check_exist = await db.query(
      "SELECT * FROM books WHERE title ~ $1 AND author ~ $2",
      [title, author]
    );
    if(check_exist.rows.length > 0) {
      return res.status(400).json({
        status: "failure",
        msg: "Book already exists"
      })
    }
    const results = await db.query(
      "INSERT INTO books(title, author, category) VALUES ($1, $2, $3) RETURNING *",
      [title, author, category]
    );
    console.log(results.rows);
    await db.query("COMMIT");
    res.status(201).json({
      status: "success",
      data: {
        Book: results.rows[0],
      },
    });
  } catch (error) {
    console.error(error);
    await db.query("ROLLBACK");
  }
});
```

Fig 2.2.24

Route to get categories of all the books in the database -

```
router.get("/category", async (req, res) => {
  try {
    console.log("initiating get request for all book categories...");
    const get_result = await db.query("SELECT DISTINCT category FROM books");
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        Categories: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
    res.status(400).json({
      status: "bad request",
    });
  }
});
```

Fig 2.2.25

"Add Book" and "Share (or) Request a book" pages to add a book and search books to share or request respectively

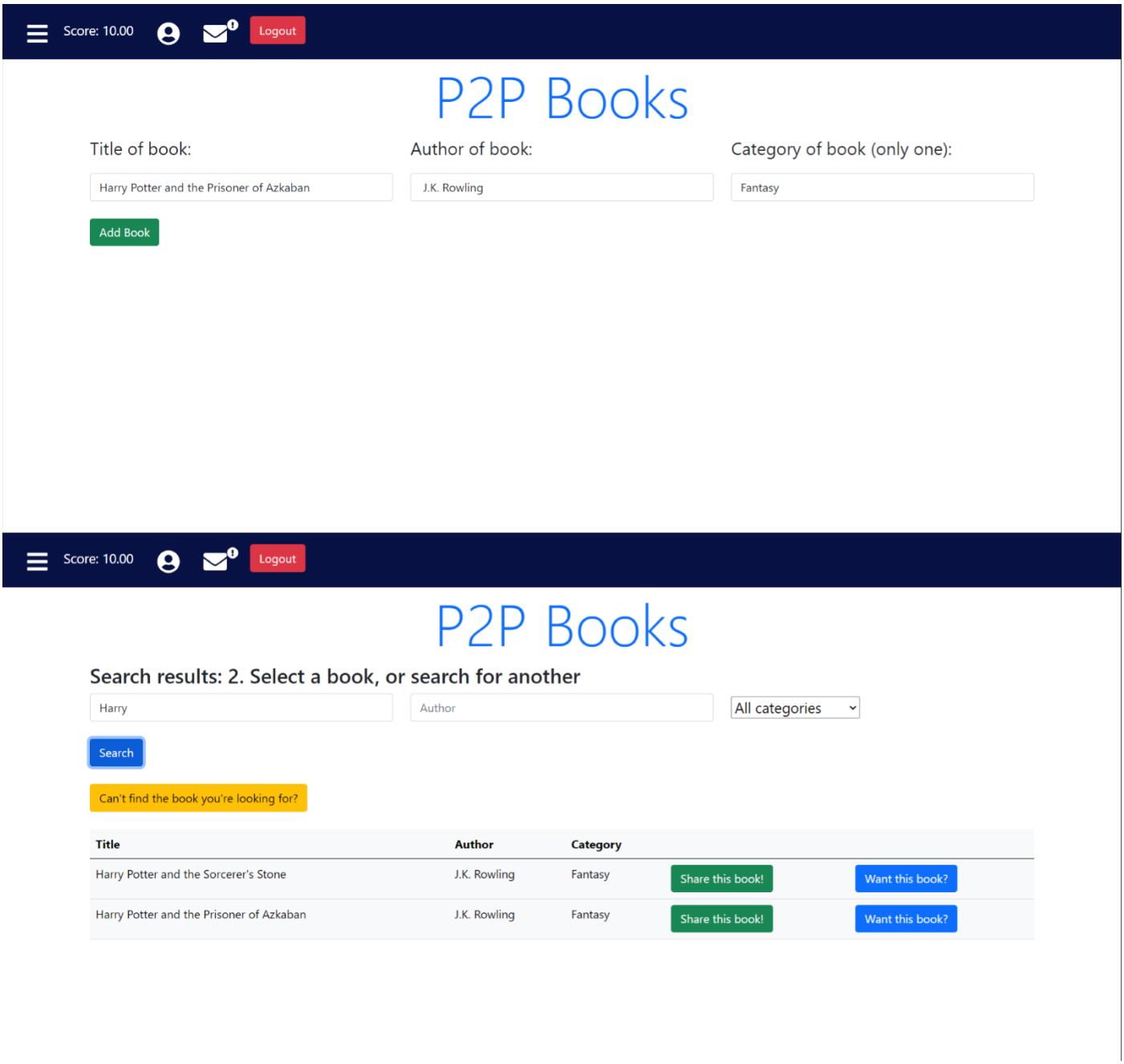


Fig 3.4

Routes dealing with transaction relation

Route to accept an offer and record a transaction -

```
router.post("/", tokenCheck, async (req, res) => {
  await db.query("BEGIN");

  try {
    const username = req.user;
    const { offer_id, return_date } = req.body;
    let date_ob = new Date();
    let curr_date = date_ob.toISOString().split("T")[0];
    const check_offers = await db.query(
      "SELECT * FROM offers WHERE offer_id = $1",
      [offer_id]
    );
    console.log(check_offers.rows);
    if(check_offers.rows.length == 0) {
      console.log("Deleted offer");
      return res.status(400).json({
        status: "failure",
        msg: "Offer does not exist anymore..."
      });
    } else {
      console.log("Offer exists");
      const get_result = await db.query(
        "INSERT INTO transactions(date_of_transac, return_date, offer_id)\
          values($1,$2,$3) RETURNING *",
        [curr_date, return_date, offer_id]
      );
      const update_book_status = await db.query(
        "UPDATE books_active SET book_status = 'R' WHERE book_active_id IN\
          (SELECT book_active_id from books_active WHERE owner = $1 AND book_active_id IN\
            (SELECT ba.book_active_id FROM books_active ba INNER JOIN offers o ON o.book_active_id=ba.book_active_id\
              AND o.offer_id = $2))",
        [username, offer_id]
      );
      console.log(get_result.rows);
      console.log(update_book_status.rows);

      await db.query("COMMIT");

      res.status(201).json({
        status: "success",
        data: {
          transaction_details: get_result.rows[0],
        },
      });
    }
  } catch (error) {
    console.error(error);
    await db.query("ROLLBACK");
  }
});
```

Fig 2.2.26

Route to get transactions of a user -

```
router.get("/", tokenCheck, async (req, res) => {
  try {
    const username = req.user;
    const get_result = await db.query(
      "SELECT t.transaction_id, t.date_of_transac, t.return_date, o.renter FROM \
      transactions t INNER JOIN offers o ON t.offer_id = o.offer_id INNER JOIN books_active ba ON\
      ba.book_active_id = o.book_active_id AND ba.owner = $1",
      [username]
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        transaction_details: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.27

Route to update the return date of a book being rented -

```
router.put("/", tokenCheck, async (req, res) => {
  try {
    const { transaction_id, return_date } = req.body;
    const get_result = await db.query(
      "UPDATE transactions SET return_date = $1 WHERE transaction_id = $2 RETURNING *",
      [return_date, transaction_id]
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        new_transaction_details: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.28

Routes dealing with notification relation

Route to get notifications of a user -

```
router.get("/", tokenCheck, async (req, res) => {
  try {
    const username = req.user;
    const get_result = await db.query(
      "SELECT * FROM notification WHERE username=$1 ORDER BY notif_id DESC",
      [username]
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        reqMessages: get_result.rows,
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.29

Route to get notification count of a user -

```
router.get("/notif-count", tokenCheck, async (req, res) => {
  try {
    const username = req.user;
    const get_result = await db.query(
      "SELECT COUNT(notif_id) FROM notification WHERE username=$1",
      [username]
    );
    console.log(get_result.rows);
    res.status(201).json({
      status: "success",
      data: {
        Count: get_result.rows[0]["count"],
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.30

Route to delete a notification -

```
router.delete("/", tokenCheck, async (req, res) => {
  try {
    const { notif_id } = req.query;
    const get_result = await db.query(
      "DELETE FROM notification WHERE notif_id=$1 RETURNING *",
      [notif_id]
    );
    console.log(get_result.rows[0]);
    res.status(201).json({
      status: "success",
      data: {
        Deleted_notif: get_result.rows[0],
      },
    });
  } catch (error) {
    console.error(error);
  }
});
```

Fig 2.2.31

"my-messages" page of a user

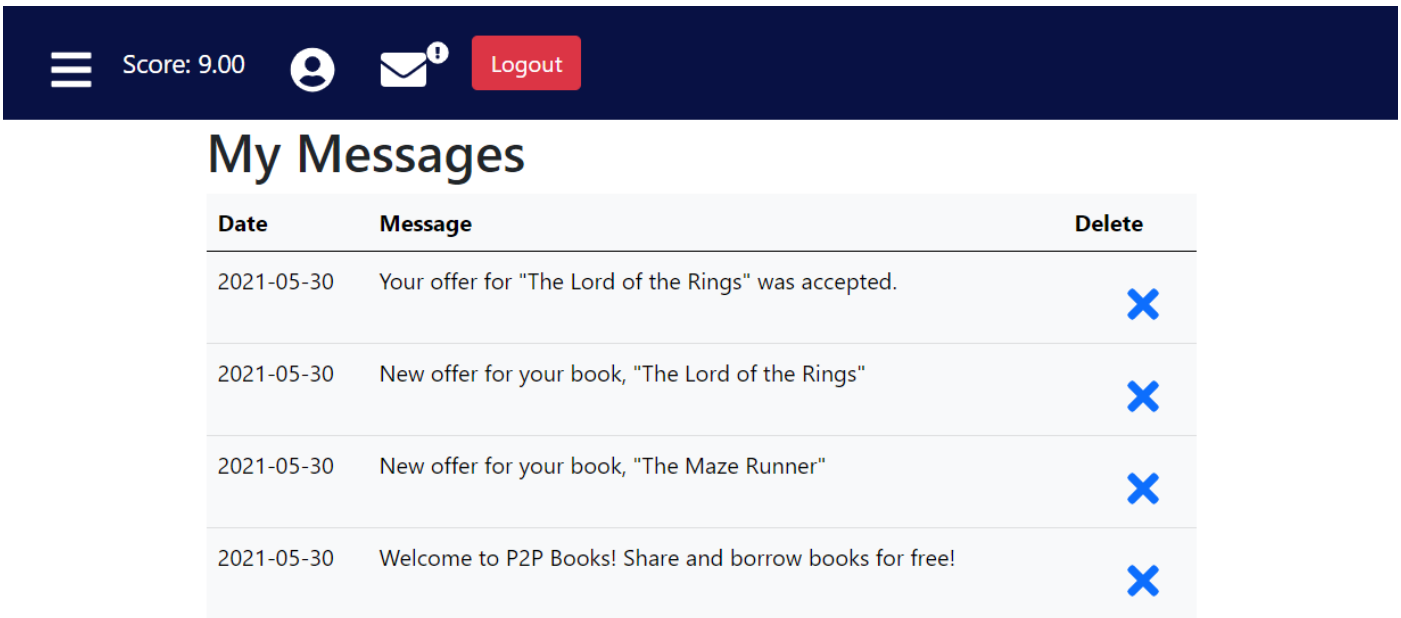


Fig 3.6

CONCLUSION

The purpose of this project was to come up with a book rental system whose integrity depends upon the goodwill of the community using it, thus giving independence in the hands of the users, and cementing the stance on the belief that knowledge is free.

The implementation of this project as explained in detail already, involves major features like a reputation system which encourages more users to put their own books into circulation rather than being a party that only sends requests or offers for books. This encourages the community as a whole to participate in putting more and more books into circulation.

We have also discussed about the basic features of the project such as adding a new book, putting a book into circulation, search filters for active books, requesting for a particular book that is not in circulation, allowing owners to accept/deny offers made by renters for a book that the owner possesses, making and deleting offers and requests, giving the owner the independence of verifying the outcome of a transaction with a renter and to set a return date for said book, and a notification system for notifying the user regarding many things such as

successful login, a messages page, and rejected/accepted offers, and fulfilled requests.

As students, we have gained firsthand experience with full stack development while working on this project, and it is evident that we have learned the basics of how the front end, back end and the database work in tandem.

We have gained deeper understanding of the concepts of efficient schema creation and querying, normalization, and atomicity of transactions happening in a database. We have found this deep dive into application development very engaging and will plan to add more features in the future.

References

Documentations:

- PostgreSQL - https://www.postgresql.org/docs/13/index.html
- ExpressJS - http://expressjs.com/en/api.html
- NodeJS - https://nodejs.org/en/docs/
- ReactJS - https://reactjs.org/docs/getting-started.html

YouTube:

- https://www.youtube.com/watch?v=J01rYl9T3BU

- https://www.youtube.com/watch?v=w7ejDZ8SWv8

GitHub: https://github.com/20kaushik02/P2P_Books