**BOOK BRIDGE**

**Introduction**

**BookBridge** is a mobile application designed to foster a culture of reading among college students by facilitating the exchange of books. Recognizing the financial barriers students face in acquiring all the books they need, this platform provides a secure and efficient way for them to borrow books from their peers.

**Key Features and Functionality**

Users must register with their name, college registration number, and email. Passwords must be at least 8 characters long and include at least one letter and one number. For enhanced security, a single sign-on (SSO) option is available, and user identity is verified via a one-time password (OTP) sent to their registered email.

Once registered, students can:

* **List and Lend:** Add books they own and are willing to lend, providing details such as title, author, and genre.
* **Search and Discover:** Browse or search for specific books.
* **Request and Exchange:** If a desired book is found, a student can view the owner's details (name and registration number) and send a loan request. The owner is notified and can accept or reject the request.
* **Secure Communication:** All communication regarding the exchange, including pickup and return arrangements, is conducted securely through the platform's in-app messaging or via the user's registered email, ensuring privacy and security.

This application creates a cooperative ecosystem where students can access educational resources without the financial burden of purchasing new books.

Tables required to store info in db:

1. User:
   1. user\_id: Primary key.
   2. name: User's full name.
   3. reg\_no: College registration number.
   4. email\_id: Unique email address.
   5. password: Hashed password.
   6. created\_at: Timestamp for when the user was created.
2. Refresh\_tokens:
   1. token: The refresh token.
   2. user\_id: Foreign key linking to the User table.
   3. expires\_at: Timestamp for token expiration.
   4. created\_at: Timestamp for token creation.
3. Books:
   1. book\_id: Primary key.
   2. book\_name: Title of the book.
   3. author\_name: Author of the book.
   4. genre: Book genre.
   5. publication\_year: Year it is published.
   6. owner\_id: Foreign key linking to the User table.
   7. availability\_status: Available (True) or Lent (False).
   8. ISBN: a unique identifier and is often a more reliable way to find a specific edition of a book.
   9. created\_at: Timestamp for when the record was created.
4. Books\_exchanged: exchange\_id, borrower\_id, lender\_id, status
   1. exchange\_id: Primary key.
   2. borrower\_id: Foreign key linking to the User table.
   3. lender\_id: Foreign key linking to the User table.
   4. book\_id: The book being exchanged.
   5. status: This should be an **enumeration** (e.g., enum) with specific values to track the loan's state, such as 'Pending', 'Accepted', 'Rejected', 'Returned', 'Overdue'.
   6. request\_date: The date the request was made.
   7. borrow\_date: The date the book was lent out.
   8. return\_date: The date the book was returned.
   9. created\_at: Timestamp for when the record was created.

SQL queries:

CREATE TABLE IF NOT EXISTS users (

user\_id SERIAL PRIMARY KEY,

user\_name VARCHAR(100) NOT NULL,

registration\_number INTEGER NOT NULL UNIQUE,

email\_id VARCHAR(100) NOT NULL UNIQUE,

password VARCHAR(300) NOT NULL,

is\_active BOOLEAN DEFAULT TRUE,

created\_at TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE IF NOT EXISTS refresh\_tokens

(

token VARCHAR(255) NOT NULL PRIMARY KEY,

user\_id INTEGER NOT NULL,

expires\_at TIMESTAMP WITHOUT TIME ZONE,

is\_valid BOOLEAN,

created\_at TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

CONSTRAINT refresh\_tokens\_user\_id\_fkey FOREIGN KEY (user\_id)

REFERENCES public.users (user\_id)

ON UPDATE NO ACTION

ON DELETE CASCADE

);

CREATE TYPE book\_availability\_status AS ENUM (

'Available',

'Lent',

'Lost'

);

CREATE TYPE exchange\_status AS ENUM (

'Pending',

'Accepted',

'Rejected',

'Returned',

'Overdue'

);

CREATE TABLE IF NOT EXISTS books (

book\_id SERIAL PRIMARY KEY,

book\_name VARCHAR(255) NOT NULL,

author\_name VARCHAR(255) NOT NULL,

publication\_year INTEGER CHECK (publication\_year > 0),

isbn VARCHAR(20) UNIQUE,

created\_at TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE IF NOT EXISTS book\_genres (

book\_id INTEGER REFERENCES books(book\_id),

genre VARCHAR(100),

created\_at TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE IF NOT EXISTS books\_users (

id SERIAL PRIMARY KEY,

book\_id INTEGER NOT NULL,

owner\_id INTEGER NOT NULL,

availability\_status book\_availability\_status NOT NULL DEFAULT ‘Available’,

created\_at TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

CONSTRAINT books\_users\_owner\_id\_fkey FOREIGN KEY (owner\_id)

REFERENCES public.users (user\_id)

ON UPDATE NO ACTION

ON DELETE CASCADE,

CONSTRAINT books\_users\_book\_id\_fkey FOREIGN KEY (book\_id)

REFERENCES public.books (book\_id)

ON UPDATE NO ACTION

ON DELETE CASCADE

);

CREATE TABLE IF NOT EXISTS books\_exchanged (

exchange\_id SERIAL PRIMARY KEY,

borrower\_id INTEGER NOT NULL,

lender\_id INTEGER NOT NULL,

book\_id INTEGER NOT NULL,

request\_date TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

borrow\_date TIMESTAMP WITH TIME ZONE,

returned\_date TIMESTAMP WITH TIME ZONE CHECK (borrow\_date > request\_date),

due\_date TIMESTAMP WITH TIME ZONE,

status exchange\_status NOT NULL DEFAULT ‘Pending’,

created\_at TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT\_TIMESTAMP,

CONSTRAINT books\_exchanged\_borrower\_id\_fkey FOREIGN KEY (borrower\_id)

REFERENCES public.users (user\_id)

ON UPDATE NO ACTION

ON DELETE CASCADE,

CONSTRAINT books\_exchanged\_lender\_id\_fkey FOREIGN KEY (lender\_id)

REFERENCES public.users (user\_id)

ON UPDATE NO ACTION

ON DELETE CASCADE,

CONSTRAINT books\_exchanged\_book\_id\_fkey FOREIGN KEY (book\_id)

REFERENCES public.books (book\_id)

ON UPDATE NO ACTION

ON DELETE CASCADE

);

Best Practices to Avoid Over-Indexing

* ✅ Index only columns used in WHERE, JOIN, or ORDER BY.
* ✅ Avoid indexing columns that rarely appear in queries.
* ✅ Use EXPLAIN ANALYZE to confirm which indexes are actually used.
* ✅ Periodically audit your indexes with:

SELECT \* FROM pg\_indexes WHERE tablename = 'your\_table';

CREATE INDEX idx\_books\_book\_name ON books (book\_name);

CREATE INDEX idx\_books\_author\_name ON books (author\_name);

CREATE INDEX idx\_book\_genres\_genre ON book\_genres (genre);

CREATE INDEX idx\_book\_genres\_book\_id ON book\_genres (book\_id);

Relationships:

1. **Users ↔ Books** = **Many-to-Many**

* You nailed it: multiple users can own the same book title (e.g., *Ikigai*), and each user can own multiple books.
* This is modeled via the books\_users table.

2. **Users ↔ Refresh Tokens** = **One-to-Many**

* One user can have many tokens (active or expired).
* Each token belongs to one user.

3. **Users ↔ Exchanges** = **Two One-to-Many Relationships**

* A user can **borrow** many books → borrower\_id
* A user can **lend** many books → lender\_id

So, books\_exchanged links two users and one book per transaction.

BUSINESS RULES:

* 1. If a user would like to delete his/her account. Change ‘is\_active’ column for that user to False (Soft-delete).
  2. If a user logout or password change (tokens revoke), set ‘is\_valid’ to Flase.
  3. If a book is marked lent, then it should not be added to exchange table. (meaning again can't be lent) => APPLICATION LOGIC
  4. If a book is lost, it should not be requested and added to exchange table as well. => APPLICATION LOGIC
  5. Can add more than one genre for a book. Use ‘+’ sign to let the user add more genres and send it as list to update table in backend (this is ease the filtering or searching).
  6. Can add or leave as it is for publication year and ISBN (not required fields but for filters).
     + To only show books with a known publication year, use `SELECT \* FROM books WHERE publication\_year IS NOT NULL;`
     + Filter by year but allow fallback, use `SELECT \* FROM books WHERE COALESCE(publication\_year, 0) >= 2010;`. This treats NULL as 0, so only books from 2010 onward are shown.
     + While searching by ISBN, SELECT \* FROM books WHERE isbn IS NOT NULL AND isbn = '9780143441234';
     + Sort with NULLs Last: SELECT \* FROM books ORDER BY publication\_year ASC NULLS LAST;
     + Or in API or Frontend,

if book['publication\_year'] is not None:

# show year

else:

# show "Year not available"

* 1. Should not list 'lost' books for the users in List of Books Page. Use `SELECT \* FROM books\_users WHERE availability\_status != 'Lost';`
  2. Should not request books of inactive users. (If any book that is owned only by invalid users should not be shown in list of books page for other users to request).
     + SELECT bu.\* FROM books\_users bu

JOIN users u ON bu.owner\_id = u.user\_id

WHERE u.is\_active = TRUE AND bu.availability\_status != 'Lost';

* 1. In books\_exchanged table, request\_date < borrow\_date and borrow\_date should be early than returned date and due date. But users might return after due date or on due date or before due date so no constraint on it.
  2. In books\_exchanged table, once a book is returned (status = 'returned') no more alterations to those entries.  Application logic: disable updates if status = 'Returned'
  3. Based on due dates, the requested books can be in green, yellow and red color. Red color indicating that the borrowed book is overdue.
  4. A user shouldn’t request the same book multiple times while a request is pending.

SELECT \* FROM books\_exchanged WHERE borrower\_id = ? AND book\_id = ? AND status = 'Pending';

* 1. Restrict how many books a user can borrow at once. Max = 3.
  2. **Auto-Update Overdue Status**
* If due\_date < CURRENT\_DATE and status != 'Returned', mark as 'Overdue'.

**Option**: Use a scheduled job or trigger to update status daily.

* Use pg\_corn to UPDATE books\_exchanged SET status = 'Overdue'

WHERE due\_date < CURRENT\_DATE AND status != 'Returned';

* To do so, we can use node-corn in node.js.

**Schedule jobs via postgresql corn:**

using PostgreSQL, the cleanest way to automate this is by using the [pg\_cron](https://github.com/citusdata/pg_cron) extension, which allows you to schedule SQL jobs like a cron task inside the database itself.

**🕒 Step-by-Step: Auto-Update Overdue Status Using pg\_cron**

**✅ 1. Install pg\_cron Extension**

If you're using PostgreSQL 13+ and your setup supports extensions:

CREATE EXTENSION pg\_cron;

If it says the extension is missing, you may need to install it via your package manager or enable it in your PostgreSQL configuration.

**✅ 2. Schedule the Daily Update Job**

This query will mark books as 'Overdue' if their due\_date has passed and they haven’t been returned:

SELECT cron.schedule(

'update\_overdue\_status',

'0 0 \* \* \*', -- runs daily at midnight

$$UPDATE books\_exchanged

SET status = 'Overdue'

WHERE due\_date < CURRENT\_DATE

AND status != 'Returned';$$

);

You can change the time to suit your needs (e.g., '0 6 \* \* \*' for 6 AM daily).

**✅ 3. Verify Scheduled Jobs**

To check your scheduled jobs:

SELECT \* FROM cron.job;

This will show the job name, schedule, and status.

**🧠 Bonus Tip: Add Logging**

If you want to track how many rows were updated, you can wrap the query in a function and log the count to a table.

**Referential actions** in SQL—rules that define what happens to **child rows** when a **parent row** is updated or deleted. They’re essential for maintaining **data integrity** in relational databases.

**🔗 Referential Actions in Foreign Keys**

When you define a foreign key, you can specify what should happen to the referencing row (child) if the referenced row (parent) is **deleted** or **updated**.

**✅ ON DELETE CASCADE**

* **Meaning**: If a parent row is deleted, all child rows referencing it are automatically deleted.
* **Use Case**: If a user deletes their account, you might want to delete all their books, tokens, exchanges, etc.

FOREIGN KEY (user\_id) REFERENCES users(user\_id) ON DELETE CASCADE

**✅ ON UPDATE NO ACTION**

* **Meaning**: If the parent row’s key is updated, the database **does nothing** to the child rows.
* **Use Case**: You rarely update primary keys, so this is safe and common.

FOREIGN KEY (user\_id) REFERENCES users(user\_id) ON UPDATE NO ACTION

**🧠 Other Referential Actions**

| **Action** | **What It Does** |
| --- | --- |
| CASCADE | Propagates the change (delete or update) to child rows |
| SET NULL | Sets the foreign key in child rows to NULL |
| SET DEFAULT | Sets the foreign key in child rows to its default value |
| RESTRICT | Prevents the action if any child rows exist |
| NO ACTION | Same as RESTRICT, but enforcement may be deferred depending on DB engine |

**🧪 Examples**

**1. ON DELETE SET NULL**

FOREIGN KEY (owner\_id) REFERENCES users(user\_id) ON DELETE SET NULL

If a user is deleted, their books remain but owner\_id becomes NULL.

**2. ON DELETE RESTRICT**

FOREIGN KEY (book\_id) REFERENCES books(book\_id) ON DELETE RESTRICT

You **cannot delete** a book if it’s still referenced in books\_exchanged.

**✅ Best Practices**

* Use CASCADE when child data is meaningless without the parent.
* Use SET NULL when child data can exist without the parent.
* Use RESTRICT or NO ACTION when you want to **prevent accidental deletes**.
* Avoid updating primary keys—stick with ON UPDATE NO ACTION.