

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:
 - a. user_id: Primary key.
 - b. name: User's full name.
 - c. reg_no: College registration number.
 - d. email_id: Unique email address.
 - e. password: Hashed password.
 - f. created_at: Timestamp for when the user was created.

2. Refresh_tokens:

- a. token: The refresh token.
- b. user_id: Foreign key linking to the User table.
- c. expires_at: Timestamp for token expiration.
- d. created_at: Timestamp for token creation.

3. Books:

- a. book_id: Primary key.
- b. book_name: Title of the book.
- c. author_name: Author of the book.
- d. genre: Book genre.
- e. publication_year: Year it is published.
- f. owner_id: Foreign key linking to the User table.
- g. availability_status: Available (True) or Lent (False).
- h. ISBN: a unique identifier and is often a more reliable way to find a specific edition of a book.
- i. created_at: Timestamp for when the record was created.

4. Books_exchanged: exchange_id, borrower_id, lender_id, status

- a. exchange_id: Primary key.
- b. borrower_id: Foreign key linking to the User table.
- c. lender_id: Foreign key linking to the User table.
- d. book_id: The book being exchanged.
- e. 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'.
- f. request_date: The date the request was made.
- g. borrow_date: The date the book was lent out.
- h. return_date: The date the book was returned.
- i. 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 False.
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"
7. Should not list 'lost' books for the users in List of Books Page. Use `SELECT * FROM books_users WHERE availability_status != 'Lost';`
8. 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';
9. 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.

10. In books_exchanged table, once a book is returned (status = 'returned') no more alterations to those entries. ? Application logic: disable updates if status = 'Returned'
 11. Based on due dates, the requested books can be in green, yellow and red color. Red color indicating that the borrowed book is overdue.
 12. 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';`
 13. Restrict how many books a user can borrow at once. Max = 3.
 14. **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_cron` to `UPDATE books_exchanged SET status = 'Overdue' WHERE due_date < CURRENT_DATE AND status != 'Returned';`
 - To do so, we can use `node-cron` in `node.js`.
-

Schedule jobs via postgresql cron:

using PostgreSQL, the cleanest way to automate this is by using the [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.