BOOK BRIDGE

XT5089 MOBILE AND PERVASIVE COMPUTING

S. No.	REGISTRATION NUMBER	NAME	PROGRAMME & BRANCH
1	2021239006	HARINI S	M. Sc. (Computer Science)
2	2021242003	DHAYA VARSHINI SK	M. Sc. (Information Technology)
3	2021242007	HARI PRIYA K	M. Sc. (Information Technology)

INTRODUCTION

BookBridge is a student-centric platform designed to democratize access to books through peer-to-peer exchange. In academic environments where affordability and accessibility often hinder reading books, BookBridge empowers students to lend, borrow, and discover books within their campus or community—without relying on costly intermediaries.

Built with a robust backend architecture and secure transactional logic, BookBridge ensures data integrity, user safety, and seamless book tracking. The platform supports real-time availability status, exchange history, and genre-based discovery, making it intuitive for users to find what they need and contribute what they have. With features like refresh token–based authentication, modular service layers, and scalable schema design, BookBridge is engineered for maintainability and growth.

Beyond technology, BookBridge fosters a culture of collaboration, sustainability, and shared learning. Whether you're a student looking for a textbook, or sharing your favorite novel, BookBridge bridges the gap between need and generosity—one book at a time.

CODE

Features Available:

1. **REGISTRATION:** Allows new users to sign up by providing essential details like name, registration number, email, and password. Ensures secure onboarding and uniqueness through validations and database constraints.
2. **LOGIN:** Authenticates users using their registered credentials. Generates secure access and refresh tokens to manage session validity and protect user data.
3. **FORGOT PASSWORD:** Enables users to reset their password if forgotten. Typically involves email verification-based recovery to ensure account security.
4. **VIEW BOOKS LIST:** Displays all books available in the system, including metadata like title, author, publication year, and genre.
5. **VIEW OWNER DETAILS:** Shows the profile of the users who owns a specific book. Includes name, registration number, availability status, and email ID.
6. **REQUEST BOOK:** Allows a user to initiate a borrowing request for a selected book. Tracks request status (Pending, Accepted, Rejected, etc.) and logs timestamps for request, borrow, return, and due dates.
7. **VIEW LIST OF REQUESTED BOOKS:** Displays all books the user has requested, along with their current status. Helps users manage active, past, and pending exchanges
8. **VIEW INCOMING REQUESTS:** For book owners, this feature lists incoming borrowing requests from other users. Owners can accept, reject, or update the status of each request.
9. **VIEW & UPDATE PROFILE:** Lets users view and edit their personal information such as name, email, password, and activity status. Promotes personalization and account management.
10. **VIEW & UPDATE MY BOOKS:** Allows users to manage the books they've added—remove lost books, or add new ones. Ensures accurate tracking of shared resources.

CODE – VIEW BOOK LIST & VIEW OWNER DETAILS:

1. **app.js:**

```
import cors from "cors";
import dotenv from "dotenv";
import express from "express";

import { sequelize } from "./db/sequelize.js"
import models from "./models/index.js";
dotenv.config();
const app = express();
const FRONTEND_URL = process.env.FRONTEND_URL;
```

```
sequelize
  .authenticate()
  .then(() => {
    console.log("Database connected!");
    return sequelize.sync({ alter: true });
  })
  .then(() => {
    console.log("All models synced!");
    app.listen(PORT, () => {
      console.log(`Server running on port ${PORT}`);
    });
  })
  .catch((err) => {
    console.error("DB Error:", err);
});
sequelize.models = models;

app.use(express.urlencoded({ extended: true }));
app.use(express.json());
app.use(
  cors({
    origin: FRONTEND_URL,
    credentials: true,
  })
);

import { userAuthRoutes } from "./authentication/userAuthRoutes.js";
app.use("/userAuth", userAuthRoutes);
import { booksRoutes } from "./routers/bookRoutes.js";
app.use("/books", booksRoutes);
import { exchangeRoutes } from "./routers/bookExchangeRoutes.js";
app.use("/exchanges", exchangeRoutes);
import { userRoutes } from "./routers/userRoutes.js";
app.use("/user",userRoutes )
export default app;
```

2. **models/index.js:**

```
import User from "./user.js";
import RefreshToken from "./refreshToken.js";
import Book from "./book.js";
```

```
import Genre  from "./Genre.js";
import BookGenre from "./bookGenre.js";
import BookUser from "./bookUser.js";
import BookExchange from "./bookExchange.js";

const models = { User, RefreshToken, Book, Genre, BookGenre, BookUser, BookExchange };
export default models;
```

3. **db/sequelize.js:**

```
import dotenv from "dotenv";
import {Sequelize} from "sequelize";
dotenv.config({ path: "./.env" });

const sequelize = new Sequelize(
  process.env.POSTGRESQL_DB_NAME,
  process.env.POSTGRESQL_DB_USERNAME,
  process.env.POSTGRESQL_DB_PASSWORD,
  {
    host: process.env.POSTGRESQL_DB_HOST,
    dialect: "postgres",
    logging: false,
  }
);
export { sequelize };
```

4. **db/transactionHandler.js:**

```
import { sequelize } from "./sequelize.js";
async function withTransaction(callback) {
  const t = await sequelize.transaction();
  try {
    console.log("Transaction started.");
    const result = await callback(t);
    await t.commit();
    console.log("Transaction committed.");
    return result;
  } catch (error) {
    await t.rollback();
    console.error("Transaction rolled back due to error:", error.message);
    throw error;
  }
```

```
}
export { withTransaction };
```

5. **router.js:**

```
import express from "express";
import { authenticateToken } from "../middlewares/verifyToken.js";
const booksRoutes = express.Router();
import {
  getAllBooks,
  fetchBookDetails,
} from "../controllers/bookControllers.js";

booksRoutes.get("/getAllBooks", authenticateToken, getAllBooks);
booksRoutes.get(
  "/fetchBookDetails/:bookId",
  authenticateToken,
  fetchBookDetails
);
export { booksRoutes };
```

6. **controller.js:**

```
import {
  getAllBooksService,
  getBookByBookIdService,
} from "../services/bookServices.js";

async function getAllBooks(req, res, next) {
  try {
    const books = await getAllBooksService();
    if (!books || books.length === 0) {
      return res.status(404).json({ message: "No books found" });
    }
    res.status(200).json(books);
  } catch (error) {
    next(error);
  }
}

async function fetchBookDetails(req, res, next) {
```

```
  try {
    const bookId = req.params.bookId;
    const book = await getBookByBookIdService(bookId);
    if (!book) {
      return res.status(404).json({ message: "Book not found" });
    }
    res.status(200).json(book);
  } catch (error) {
    next(error);
  }
}
export { getAllBooks, fetchBookDetails };
```

7. **service.js:**

```
import { withTransaction } from "../db/transactionHandler.js";
import {
  getAllBooks,
  getBookByBookId
} from "../models/bookModels.js";

async function getAllBooksService() {
  return withTransaction(async (transaction) => {
    const books = await getAllBooks(transaction);
    return books;
  });
}

async function getBookByBookIdService(id) {
  return withTransaction(async (transaction) => {
    const book = await getBookByBookId(id, transaction);
    return book;
  });
}
export { getAllBooksService, getBookByBookIdService };
```

8. **model.js:**

```
import { Op } from "sequelize";
import Book from "./book.js";
import User from "./user.js";
import BookUser from "./bookUser.js";
```

```
import BookExchanged from "./bookExchange.js";

async function getAllBooks(transaction) {
  try {
    const books = await Book.findAll({ transaction });
    if (books.length === 0) {
      return null;
    }
    return books;
  } catch (error) {
    console.error("Sequelize error during book fetching:", error.message);
    throw error;
  }
}

async function getBookByBookId(id, transaction) {
  try {
    const book = await Book.findByPk(id, {
      transaction,
      include: [
        {
          model: User,
          through: {
            model: BookUser,
            attributes: ["count", "available_count", "availability_status"],
          },
          attributes: ["user_id", "user_name", "email_id", "registration_number"], // customize
as needed
        },
      ],
    });
    if (!book) {
      return null;
    }
    return book;
  } catch (error) {
    console.error(
      "Sequelize error while fetching book with owners:",
      error.message
    );
    throw error;
```

```
  }
}
export {  getAllBooks,  getBookByBookId };
```

9. **ORM CODES:**

```
import { DataTypes } from "sequelize";
import { sequelize } from "../db/sequelize.js";


const User = sequelize.define(
  "User",
  {
    user_id: {
      type: DataTypes.INTEGER,
      autoIncrement: true,
      primaryKey: true,
      allowNull: false,
    },
    user_name: {
      type: DataTypes.STRING(100),
      allowNull: false,
    },
    registration_number: {
      type: DataTypes.INTEGER,
      unique: true,
      allowNull: false,
    },
    role: {
      type: DataTypes.ENUM(
        "admin",
        "student"
      ),
      defaultValue: "student",
      allowNull: false,
    },
    email_id: {
      type: DataTypes.STRING(100),
      unique: true,
      allowNull: false,
    },
    password: {
```

```
      type: DataTypes.STRING(300),
      allowNull: false,
    },
    is_active: {
      type: DataTypes.BOOLEAN,
      defaultValue: true,
    },
  },
  {
    tableName: "users",
    timestamps: true,
    underscored: true,
  }
);

const Book = sequelize.define(
  "Book",
  {
    book_id: {
      type: DataTypes.INTEGER,
      autoIncrement: true,
      primaryKey: true,
      allowNull: false,
    },
    book_name: {
      type: DataTypes.STRING(255),
      allowNull: false,
    },
    author_name: {
      type: DataTypes.STRING(255),
      allowNull: false,
    },
    publication_year: {
      type: DataTypes.INTEGER,
      validate: {
        min: 1,
      },
    },
    isbn: {
      type: DataTypes.STRING(20),
      unique: true,
```

```
      allowNull: true,
    },
  },
  {
    tableName: "books",
    timestamps: true,
    underscored: true,
    indexes: [
      {
        name: "idx_books_author_name",
        fields: ["author_name"],
      },
      {
        name: "idx_books_book_name",
        fields: ["book_name"],
      },
    ],
  }
);

const BookExchange = sequelize.define(
  "BookExchange",
  {
    borrower_id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      allowNull: false,
    },
    book_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      primaryKey: true,
      references: {
        model: Book,
        key: "book_id",
      },
      onUpdate: "NO ACTION",
      onDelete: "CASCADE",
    },
    lender_id: {
      type: DataTypes.INTEGER,
```

```
    allowNull: false,
   },
   status: {
    type: DataTypes.ENUM(
     "Pending",
     "Accepted",
     "Borrowed",
     "Rejected",
     "Returned",
     "Overdue"
    ),
    defaultValue: "Pending",
    allowNull: false,
   },
   request_date: {
    type: DataTypes.DATE,
    defaultValue: DataTypes.NOW,
   },
   borrow_date: {
    type: DataTypes.DATE,
   },
   returned_date: {
    type: DataTypes.DATE,
   },
   due_date: {
    type: DataTypes.DATE,
   },
  },
  {
   tableName: "books_exchanged",
   timestamps: true,
   underscored: true,
   validate: {
    preventSelfLending() {
     if (this.lender_id === this.borrower_id) {
      throw new Error("The borrower_id and lender_id must be different. A user cannot
borrow a book from themselves.");
     }
    },
   },
});
```

```
User.hasMany(BookExchange, { as: "BorrowedBooks", foreignKey: "borrower_id" });
User.hasMany(BookExchange, { as: "LentBooks", foreignKey: "lender_id" });

BookExchange.belongsTo(User, { foreignKey: "borrower_id" });
BookExchange.belongsTo(User, { foreignKey: "lender_id" });
BookExchange.belongsTo(Book, { foreignKey: "book_id" });

const BookUser = sequelize.define(
  "BookUser",
  {
    book_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      primaryKey: true,
      references: {
        model: Book,
        key: "book_id",
      },
      onUpdate: "NO ACTION",
      onDelete: "CASCADE",
    },
    owner_id: {
      type: DataTypes.INTEGER,
      allowNull: false,
      primaryKey: true,
      references: {
        model: User,
        key: "user_id",
      },
      onUpdate: "NO ACTION",
      onDelete: "CASCADE",
    },
    count: {
      type: DataTypes.INTEGER,
      allowNull: false,
      defaultValue: 1,
      validate: {
        min: 1,
      },
    },
    available_count: {
```

```
      type: DataTypes.INTEGER,
      allowNull: false,
      defaultValue: 1,
      validate: {
        min: 0,
      },
    },
    availability_status: {
      type: DataTypes.ENUM("Available", "Lent", "Lost"),
      allowNull: false,
      defaultValue: "Available",
    },
    is_deleted: {
      type: DataTypes.BOOLEAN,
      defaultValue: false,
    },
  },
  {
    tableName: "books_users",
    timestamps: true,
    underscored: true,
  }
);

User.belongsToMany(Book, {
  through: BookUser,
  foreignKey: "owner_id",
  otherKey: "book_id",
});

Book.belongsToMany(User, {
  through: BookUser,
  foreignKey: "book_id",
  otherKey: "owner_id",
});

export { User, Book, BookExchange, BookUser };
```

SCREENSHOTS

CONCLUSION

REFERENCES

1. https://docs.flutter.dev/
2. https://nodejs.org/en/learn/getting-started/introduction-to-nodejs
3. https://sequelize.org/