BookExchange Platform

- ALHATE DIPAK BAPU .
- SOFTWARE ENGINEERING
- M.TECH BITS PILANI
- Roll No. 2023TM93604

- Full Stack Application

Contents

Web Application Design Document

1. Project Overview

1.1 Purpose & Objectives

Scope of this document is design and detail documentation on architectural design of Book Exchange Application Platform.

This web application is designed as a **Book Exchange Application** where users can:

- sign up,

- log in, and

- access various book-related services.

The goal is to create a secure, scalable platform that allows users to exchange books and build connections/networking with people from your region.

1.2 Scope

- Core Features:
    - User Authentication
        - (Sign up, Login, Forgot Password. Token Authentication & Authorisation)
    - Book Exchange Functionality
    - REST API Integration (Flask API for database operations)
    - UI Pages for front end
- Excluded Features (for now):
    - Social media integrations
    - Advanced search and filtering features for books
    - REST API Authorisation using token
    - Modularity of code directory structure
    - Messaging and transaction management

1.3 Target Audience

- General users looking to exchange or purchase books from nearest locations

2. Functional Requirements

2.1 User Stories

2.1.1 User Story 1: User Authentication

As a user, I want to securely register, log in, and manage my account, So that I can access and use the book exchange platform.

Acceptance Criteria:

- The platform must allow users to register with a valid email and password.
- Passwords must be stored securely using encryption.
- Users should be able to reset their password via a password recovery system.
- Users should be able to log out from their account.

2.1.2 User Story 2: Book Listing

As a user, I want to list books that I want to exchange or lend, So that others can browse and request the books I offer.

Acceptance Criteria:

- Users should be able to add a book to their list by providing details such as title, author, genre, condition, and availability status.
- Each book listing must have a unique ID associated with a user's profile.
- Users should be able to edit or delete book listings at any time.
- The book listing must be displayed in the user's profile and searchable by others.

2.1.3 User Story 3: Book Search

As a user, I want to search for books based on criteria such as title, author, genre, and location,

So that I can easily find books that interest me.

Acceptance Criteria:

- The platform must provide a search bar where users can enter keywords like title, author, or genre.
- The platform should allow users to filter search results by availability status, genre, and location.
- Users must be able to view detailed information about a book (title, author, condition, etc.) when clicking on a search result.
- The search results should be paginated or load incrementally to handle large datasets.

2.1.4 User Story 4: Exchange Requests

As a user, I want to send and receive book exchange requests, So that I can initiate a transaction to exchange books with others.

Acceptance Criteria:

- Users must be able to send an exchange request to another user for a specific book.

- The request must include the option to negotiate terms, such as delivery method and exchange duration.
- The recipient of the request should be able to accept, reject, or modify the request.
- Both parties should receive notifications about the status of the exchange request (pending, accepted, rejected, modified).
- The platform should track ongoing exchanges in the user's transaction history.

2.1.5 User Story 5: Transaction Management

As a user, I want to manage my book exchanges, so that I can track the status of all my exchange transactions.

Acceptance Criteria:

- Users must be able to view a history of their exchange requests, including pending, accepted, and completed exchanges.
- The transaction management interface should allow users to cancel pending exchanges.
- Users should receive notifications when a transaction status changes (e.g., request accepted, book delivered).
- Transaction history should be available to users on their profile page.

2.2 Features & Functionality

- **Authentication:** User sign-up, login, forgot password, token-based authentication (JWT).
- **Book Management:** CRUD operations for book listings & book searching, user profile details, and auth system database.
- **Dashboard:** User-specific dashboard showing their books and exchange status.
- **API Integration:** Flask API endpoints to perform database operations for books and users.

2.3 User Roles & Permissions

- **Admin:** Full access to manage users and books. (Not implemented)
- **Registered User:** Access to exchange books, view books.

3. Non-Functional Requirements

3.1 Performance

- The application should support up to 500 concurrent users with minimal load times.

Note: It is kept at 500 for concurrent users as it's a new platform.

3.2 Security

- JWT for secure authentication.
- Password hashing.
- Two-factor authentication for password reset.

3.3 Scalability

- Horizontal scaling for increased user load, with database read replicas for efficient data access.

3.4 Usability

- Simple, intuitive UI designed for ease of use across devices.

3.5 Reliability & Availability

- 99.9% uptime expected, with automatic failover for server downtime.

3.6 Maintainability

- Well-documented code following Python and React best practices.

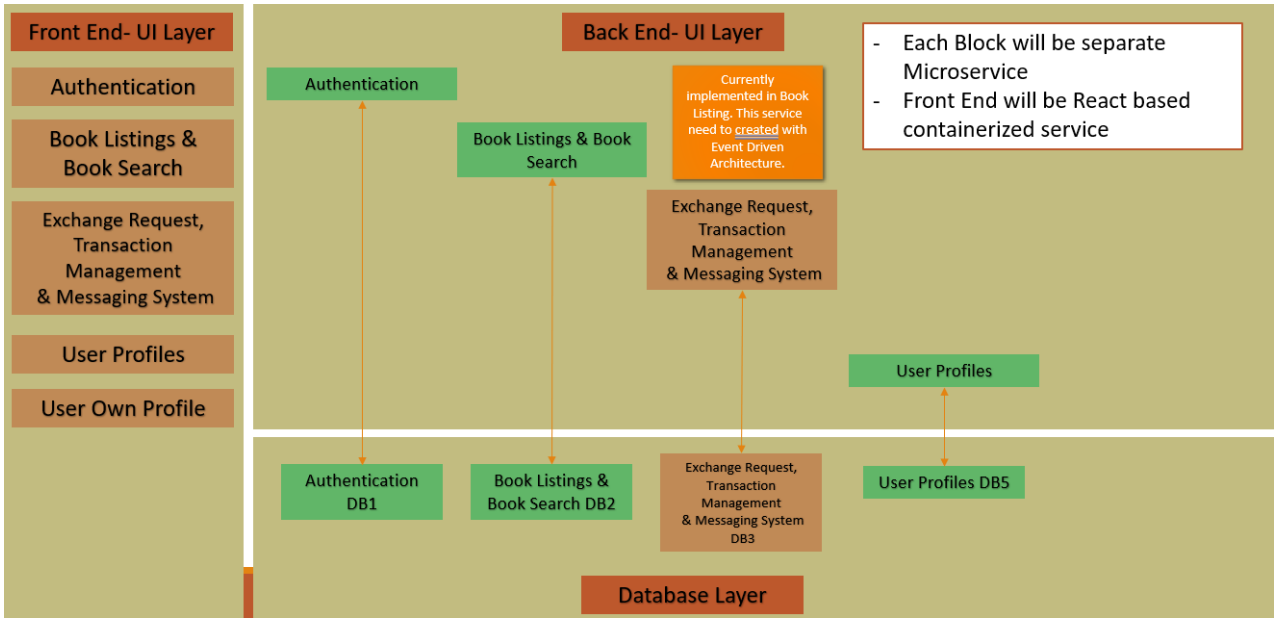
3.7 Compliance

- GDPR (General Data Protection Regulation) -compliant data handling, especially in user authentication.

4. 4. Architecture Overview

4.1 System Architecture

- Microservice architecture with containerized services for frontend (React) and backend (Flask).



4.2 Frontend Architecture

- **Technology Stack:** React, React Router (for navigation), Axios for client side request to backend.
- **Component Structure:** Reusable components for book listings, user profile, etc.
- **Routing:** React Router for navigation between pages.

4.3 Backend Architecture

- **Technology Stack:** Flask API with PostgreSQL for database management.
- **API Structure:** RESTful API exposing endpoints for user authentication and book management.
- **Authentication:** JWT tokens for secure access control.

4.4 Database Design

- **Schema for each data model:**
    - Users table: (id, name, email, password_hash).

  - Token table: (id, user_id, token, refresh_token, otp, otp_expires, created_by, expires_at).
  - Books table: (id, title, author, genre, location, listed_by, borrow_period, request_status, request_by).
  - User Profile table: (id, username, email, age, location, profilepic)

- **Relationships:**
  - Users has a one-to-many relationship with Books.
  - Books has a one-to-one relationship with Users.
  - Token has one-to-many relationship with Users.

4.5 Data Flow

- User actions (e.g., login, book exchange requests) trigger Flask API calls, which interact with the Flask SQL database.

---

5. UI/UX Design

5.1 Wireframes/Mockups

Refer slide deck for mock up.

5.2 Navigation Flow

- Home → Login/Sign-up → Dashboard → Browse Books → Book Exchange

5.3 Design Guidelines

- Colour Scheme: Soft and dark blue and white tones for a clean interface.
- Typography: Sans-serif fonts for readability.

5.4 Responsive Design

- Design optimized for desktop, tablet, and mobile.

---

6. API Design

6.1 API Endpoints

Flask API is used for backend. Following are the details of APIs.

User Profile Data Model & Microservice:

Route	Method	Description	Parameters	Request Body	Response Codes	Response Example
/user-profile	GET	Get user profile by email	Query Parameter: email (string, required)	N/A	200: Success, 400: Missing email, 404: User not found	{ "username": "JohnDoe", "email": "john.doe@example.com", "location": "New York" }
/user-profile	POST	Create or update user profile	Request Body: username (string), email (string), age (integer), location (string), profilePic (string, optional)	{ "username": "JaneDoe", "email": "jane.doe@example.com", "age": 28, "location": "Los Angeles", "profilePic": "https://example.com/profile-pic.jpg" }	201: Created/Updated, 400: Invalid input, 500: Server error	{ "message": "User profile created/updated successfully", "user": { "username": "JaneDoe", "email": "jane.doe@example.com", "location": "Los Angeles", "profilePic": "https://example.com/profile-pic.jpg" } }
/user-profile	DELETE	Delete user profile by email	Query Parameter: email (string, required)	N/A	200: Success, 400: Missing email, 404: User not found	{

Route	Method	Description	Parameters	Request Body	Response Codes	Response Example
						"message": "User with email john.doe@example.com deleted successfully" }
/user-profiles	GET	Get all user profiles	N/A	N/A	200: Success, 404: No profiles, 500: Server error	[{ "email": "john.doe@example.com", "location": "New York", "profilePic": "https://via.placeholder.com/50" }, { "email": "jane.doe@example.com", "location": "Los Angeles", "profilePic": "https://via.placeholder.com/50" }]
/user-profiles	DELETE	Delete all user profiles	N/A	N/A	200: Success, 500: Server error	{ "message": "Deleted 3 users from the database" }

Users & Token Data Model & Microservice:

Route	Method	Description	Parameters	Request Body	Response Codes	Response Example
/signup	POST	User signup	None	{"email": "string", "password": "string", "name": "string"}	201: User created 400: Missing fields 409: User already exists	{"message": "User created successfully"}
/login	POST	User login	None	{"email": "string",	200: Login successful 401: Invalid credentials	{"token": "jwt_token", "refresh_token": "jwt_token"}

Route	Method	Description	Parameters	Request Body	Response Codes	Response Example
				"password": "string"}		
/send-otp	POST	Send OTP for password reset	None	{"email": "string"}	200: OTP sent 404: User not found 500: Failed to send OTP	{"message": "OTP sent successfully", "otp": "123456"}
/reset-password	POST	Reset user password	None	{"email": "string", "otp": "string", "password": "string"}	200: Password reset successful 400: Invalid OTP 404: User not found	{"message": "Password has been reset successfully"}
/user/<int:user_id>	DELETE	Delete user by ID	user_id: int (in path)	None	200: User deleted 404: User not found	{"message": "User and associated tokens deleted successfully"}
/logout	POST	User logout	None	None	200: Logout successful 400: Token not found 500: Server error	{"message": "Logout successful"}

Book Listing Data Model & Microservice:

Route	Method	Description	Parameters	Request Body	Response Codes	Response Example
/books	POST	Add a new book	None	JSON: { "title": "book1", "author": "author1", ... }	201 Created, 400 Bad Req.	{"message": "Book added successfully"} or {"message": "Missing fields"}

Route	Method	Description	Parameters	Request Body	Response Codes	Response Example
/books	GET	Get all books	None	None	200 OK	[{"title": "book1", "author": "author1", ...}]
/books/search	GET	Get books by listed_by	listed_by as query param	None	200 OK, 404 Not Found	[{"title": "book1", "author": "author1", ...}] or {"message": "No books found matching the provided criteria"}
/raise-request	POST	Send email for raising request	None	JSON: { "to": "recipient@example.com", "subject": "subject", "message": "content" }	200 OK, 400 Bad Req.	{"success": True, "message": "Request email sent successfully!"} or {"error": "Invalid data received"}
/add-book	POST	Add a book and return its details	None	JSON: { "title": "book1", "author": "author1", ... }	201 Created, 400 Bad Req.	{"message": "Success", "id": 1, "title": "book1", ...}
/update-book	POST	Update the request_by field of a book	None	JSON: { "listed_by": "email@example.com", "title": "book1", "request_by": "requester1" }	200 OK, 404 Not Found	{"message": "Book request has been updated successfully!"} or {"message": "Book not found!"}
/books/request-update	PATCH	Update the request_status field of a book	None	JSON: { "listed_by": "email@example.com", "title": "book1", "request_status": "requested" }	200 OK, 400 Bad Req., 404 Not Found	{"message": "Updated status to requested"} or {"message": "Book not found"}
/delete-book	DELETE	Delete a book by listed_by and title	listed_by, title as query params	None	200 OK, 400 Bad Req., 404 Not Found	{"message": "Deleted successfully"} or {"message": "Book not found"}

6.2 Authentication & Authorization

- JWT tokens are used to authenticate users across login endpoints.

Note: Due to time constraints, its not implemented on all endpoints.

6.3 Rate Limiting & Throttling

- Rate limiting set to 100 requests per minute per IP. It should be implemented on server side.

Note: This is not yet implemented.

6.4 Error Handling

- Standard 400, 401, 403, and 500 error codes with detailed error messages.

6.5 Versioning

- API versioning using /v1/ in URLs.

---

7. Implementation Demo Deck

Design Document.pptx

8. Deployment & Hosting

7.1 Environment Setup

- Dev, Staging, and Production environments configured using Docker.

7.2 Hosting Infrastructure

- AWS EC2 instances running Docker containers.

7.3 Containerization & Orchestration

- Docker containers for both the React frontend and Flask backend micorservices.

7.4 CI/CD Pipeline

- GitHub Actions for continuous integration and deployment.

7.5 Load Balancing & Caching

- AWS Elastic Load Balancer (ELB) for distributing traffic.
- Redis for caching frequently accessed data.

7.6 Monitoring & Logging

- AWS CloudWatch for logging and monitoring performance.

7.7 Backup & Recovery

- Daily backups of the flask SQL database to AWS S3.

---

9. Testing Plan

8.1 Unit Testing

- PyTest for testing Flask API endpoints.

8.2 Integration Testing

- Postman integration testing for API endpoints.

8.3 End-to-End Testing

- Selenium tests for full user journeys (from sign-up to book exchange).

8.4 Performance Testing

- Apache JMeter for load testing under different user loads.

8.5 Security Testing

- Regular penetration testing and vulnerability scans using OWASP ZAP.

---

10. Security Considerations

9.1 Authentication & Authorization

- JWT-based authentication with token expiration and refresh strategies.

9.2 Data Protection

- SSL/TLS for all communications.
- AES-256 encryption for sensitive user data stored in the database.

9.3 Secure Data Storage

- Secure cookies for session management.

9.4 Common Vulnerabilities

- Protection against SQL injection, CSRF, and XSS vulnerabilities using Flask extensions and best practices.

9.5 Logging & Monitoring

- Login attempts and error logs are securely stored and monitored via CloudWatch.

---

11. Project Timeline & Milestones

10.1 Development Timeline

- Phase 1: MVP Development (2 months)
    - 3 User stories implementation: Done with basic implementation. It needs further enhancement tasks.
- Phase 2: Beta Testing (1 month)
- Phase 3: Full Launch (1 month)

10.2 Key Milestones

- MVP Completion: End of Month 2
- Beta Testing: End of Month 3
- Full Launch: End of Month 4

10.3 Resource Allocation

- Developer Team: 2 Full Stack Developers, 1 UI/UX Designer, 1 DevOps Engineer.

---

12. Risk Management

11.1 Risk Identification

- Scaling challenges under high user load.

11.2 Mitigation Strategies

- Test scalability with load testing.

11.3 Contingency Plans

- Implement horizontal scaling using AWS auto-scaling groups.

13. 12. Future Enhancements

12.1 Roadmap for New Features

- Social media integration for sharing book lists.
- Advanced filtering options for book search.

12.2 Scalability Considerations

- Plan for implementing global CDN and multi-region hosting to support international users.