SCALABLE SERVICES

ASSIGNMENT

LIBRARY MANAGEMENT SYSTEM

PREPARED BY

- Arwa Arif - 2023mt93215
- Aryan Kaushik - 2023mt93174
- Priya lekha - 2023mt93265
- Simran Khinvasara - 2023mt93328

WORK CONTRIBUTION BY GROUP MEMBERS

Name	Work Done
Arwa Arif - 2023mt93215	Created comprehensive project documentation detailing the system's architecture, functionalities, and usage. Prepared an in-depth description of the microservices (user, book, borrowing, and review).
Aryan Kaushik - 2023mt93174	Recorded the first demo video showcasing the overall functionality of the Library Management System.
Priya lekha - 2023mt93265	Deployed the application and orchestrated the services using Docker. Created the second demo video focusing on the system's security features and deployment process.
Simran Khinvasara - 2023mt93328	Designed and developed four microservices for the Library Management System. Implemented RESTful APIs using Node.js and Mongoose for seamless database interactions.

GITHUB REPOSITORIES

- User Service: https://github.com/simrankhinvasaraBITS/User-Microservice
- Book Service: https://github.com/simrankhinvasaraBITS/Book-Microservice
- Borrowing Service: https://github.com/simrankhinvasaraBITS/Borrowing-Microservice
- Return Service: https://github.com/simrankhinvasaraBITS/Review-Microservice
- Frontend (Angular): https://github.com/simrankhinvasaraBITS/Library-Management-App

DEMO VIDEOS

- First Video explaining microservices, the database and the communication between the microservices: https://drive.google.com/file/d/1p9QtJanNeWarJ8YI4nWHy464lFPaixRF/view?usp=sharing.
- Second Video to show a demo about security and deployment:: https://drive.google.com/file/d/1B0_MC-RUcRE0uY6MUJVE4-zNpgr2K0Rr/view?usp=sharing

LIBRARY MANAGEMENT SYSTEM

The Library Management System is designed to manage library resources efficiently, allowing users to search for books, borrow them, and receive notifications for due dates. This system is divided into four microservices to allow independent development, scaling, and maintenance.

In this project, we will implement an Library Management System using a **Microservices architecture**. This approach divides the application into four distinct services, each handling specific responsibilities and functioning independently. This microservices design allows for scalability, flexibility, and ease of maintenance, as each component can be developed, deployed, and updated separately without affecting the overall application.

The Library Management Application is divided into the following four microservices, each responsible for a specific part of the library's functionality:

1. User Service:

- Manages user accounts, including registration, login, and profile updates.
- Enforces security through authentication (e.g., OAuth 2.0 or JWT) and ensures access control.

2. Book Service:

- Handles book inventory, including adding new books and updating book details.
- Provides search and filtering capabilities, allowing users the users to easily find books.

3. Borrowing Service:

- Manages book borrowing and returning, storing details such as borrow dates, return dates, and the current status of each book.

4. Review Service:

- Allow users to create, update, delete, and view their reviews on specific books.
- Provide aggregated rating and review data for each book.

TECH STACK

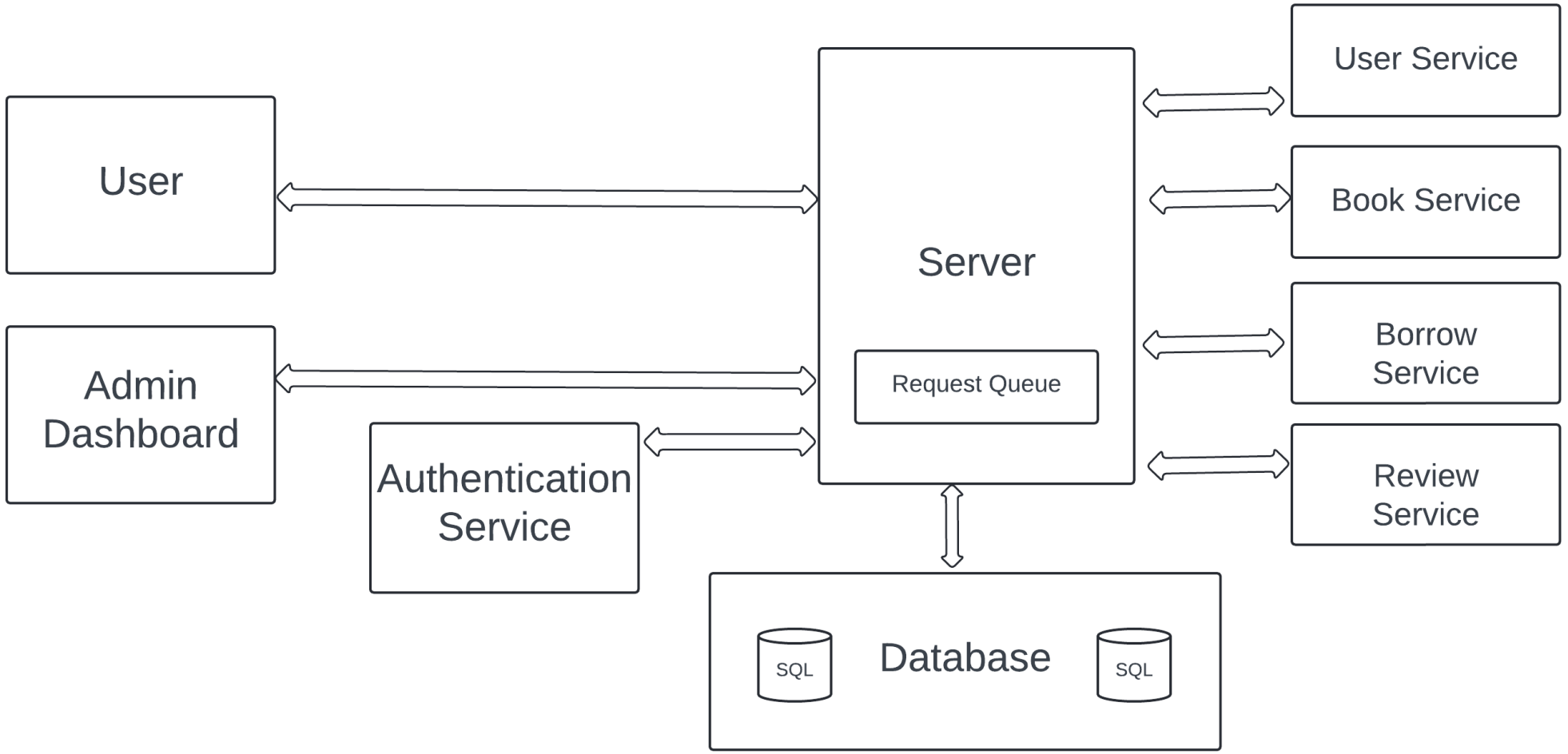
- **Programming Language**: Node.js and Angular
- **Database**: MongoDB
- **Inter-Service Communication**: REST API (synchronous) and RabbitMQ/Kafka (asynchronous for notifications)
- **Security**: OAuth 2.0 or JWT tokens
- **Containerization and Orchestration**: Docker and Kubernetes

APPLICATION ARCHITECTURE

The architecture of the Library Management System (LMS) is designed using a microservices-based approach. This approach provides modularity, scalability, and resilience by dividing the core functionalities of the system into independent, loosely-coupled services that can be developed, deployed, and scaled independently.

Each microservice in this architecture performs a specific function related to the library's operations, such as managing users, handling book inventory, processing borrowing and returns, and sending notifications. Each service has its own database, allowing for isolation and efficient scaling based on demand.

ARCHITECTURE DIAGRAM



OVERVIEW OF ARCHITECTURE

The Library Management Application architecture consists of four primary microservices:

1. User Service
2. Book Service
3. Borrowing Service
4. Notification Service

These services communicate with each other through both synchronous (REST) and asynchronous (message queues) communication, depending on the nature of the interaction.

MongoDB is used as the NoSQL database for storing user, book, borrowing, and return information.

The architecture also includes a gateway and optional authentication layer (OAuth or JWT tokens), as well as containerization and orchestration via **Docker** and **Kubernetes** for deployment. Each service can be scaled independently, and the system can handle high traffic by distributing the load among multiple instances of each microservice.

DETAILED ARCHITECTURE COMPONENTS

1. User Service

a. **Responsibilities**: Manages user registration, authentication, and profile information. It also handles role-based access control to define different permissions for library staff and patrons.

b. **Endpoints**:
   - POST /register: Register a new user (e.g., librarians and patrons).
   - POST /login: Authenticate user credentials and issue an access token (JWT).
   - GET /profile/{user_id}: Retrieve profile information for a user.

c. **Database**: MongoDB is used as the NoSQL database for storing user, book, borrowing, and return information. Each user has an entry in the User table with attributes like user_id, name, email, password, and role.

d. **Security**: Optional use of OAuth 2.0 or JWT for securing endpoints and managing authentication.

e. **Communication**: The User Service is mainly self-contained but interacts with the Borrowing Service to fetch user profiles or verify roles as needed.

2. Book Service

a. **Responsibilities**: Manages all operations related to books in the library, including adding new books, updating details, checking availability, and handling search functionalities.

b. **Endpoints**:

- POST /books: Add a new book to the system.
- GET /books: Retrieve all books or apply filters to search by title, author, or genre.
- GET /books/{book_id}: Get details about a specific book.
- PATCH /books/{book_id}/status: Update the availability status of a book.

c. **Database**: The Book table includes fields like book_id, title, author, genre, and availability.

d. **Communication**: The Book Service is frequently accessed by the Borrowing Service to check the availability of books during the borrowing and returning process.

3. Borrowing Service

a. **Responsibilities**: Manages the process of borrowing and returning books. It keeps track of borrow dates, return dates, and overdue books, maintaining the borrowing history for each user.

b. **Endpoints**:

- POST /borrow: Record a new borrowing transaction.
- POST /return: Record the return of a borrowed book.
- GET /borrowing/{user_id}: Retrieve a user's borrowing history.

c. **Database**: The BorrowRecord table holds details for each transaction, with fields like record_id, user_id, book_id, borrow_date, and return_date.

d. **Communication**: Communicates with the Book Service to check book availability and update status when books are borrowed or returned.

4. Review Service

a. **Responsibilities**: Allow users to leave reviews and ratings for books, providing valuable feedback.

b. **Endpoints**:

- POST /reviews: Submit a new review and rating for a book.
  - Request parameters: user_id, book_id, rating, review_text.
- GET /reviews/{book_id}: Retrieve all reviews for a specific book.
- GET /reviews/average/{book_id}: Retrieve the average rating for a specific book.
- PATCH /reviews/{review_id}: Update a review (only if the user is the author of the review).
- DELETE /reviews/{review_id}: Delete a review (only if the user is the author of the review).

c. **Database**: The Review table holds details for each ratings, with fields like review_id, user_id, book_id and rating.

d. **Communication**: The Book Service can call the Review Service to retrieve reviews or average ratings for display on the book details page. The Book Service communicates with the Review Service via REST API to retrieve reviews and ratings when displaying book details.

SUPPORTING COMPONENTS IN ARCHITECTURE

1. API Gateway

The API Gateway serves as a single entry point to the application, routing client requests to the appropriate microservices. It provides centralized access control and security, including rate limiting and request validation.

- Routes incoming requests to the appropriate microservice endpoint.
- Handles token validation, ensuring that each request is authenticated.
- Aggregates data if needed (e.g., combining user profile and borrowing history in a single response).

2. Authentication and Security Layer (OAuth/JWT)

Manages user authentication and secures inter-service communication. By issuing tokens (JWT or OAuth 2.0), it ensures only authorized users can access certain endpoints and perform actions according to their roles.

After a user logs in, they receive an access token which must be included in requests to protected endpoints.

Role-Based Access Control (RBAC): Allows for fine-grained control, permitting patrons to borrow books and view profiles, while librarians have elevated permissions for managing inventory.

3. Message Broker (RabbitMQ/Kafka)

Facilitates asynchronous communication between services, specifically enabling the Review Service to send ratings without blocking other operations.

The Borrowing Service publishes messages to the queue (e.g., when a book is borrowed).

4. Databases (Database per Service)

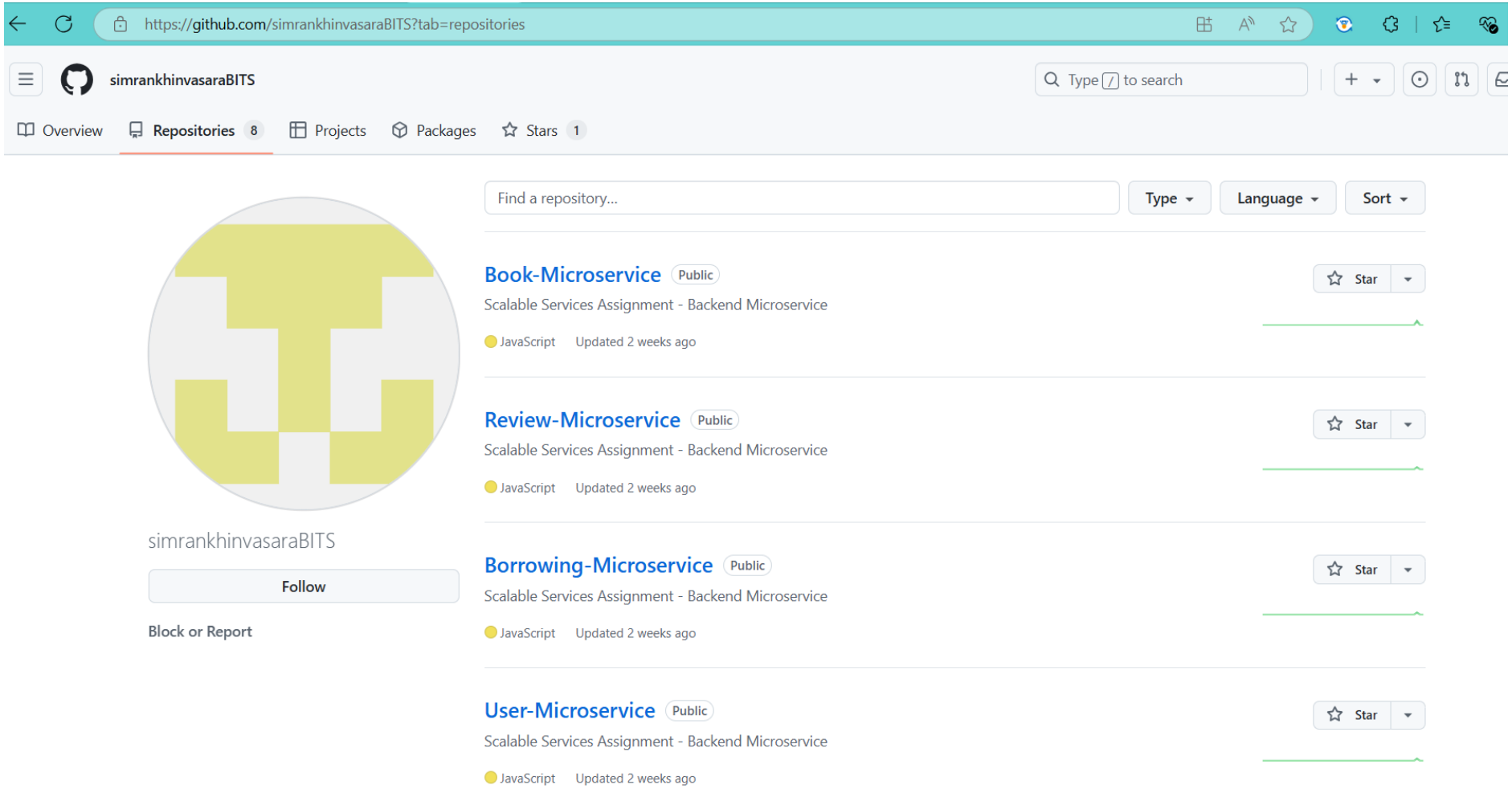
Each microservice has its own dedicated database to store data relevant to its function, enhancing data isolation, security, and scalability.

- User Service Database (User data)
- Book Service Database (Book inventory data)
- Borrowing Service Database (Borrowing records)
- Review Service Database (Review records)

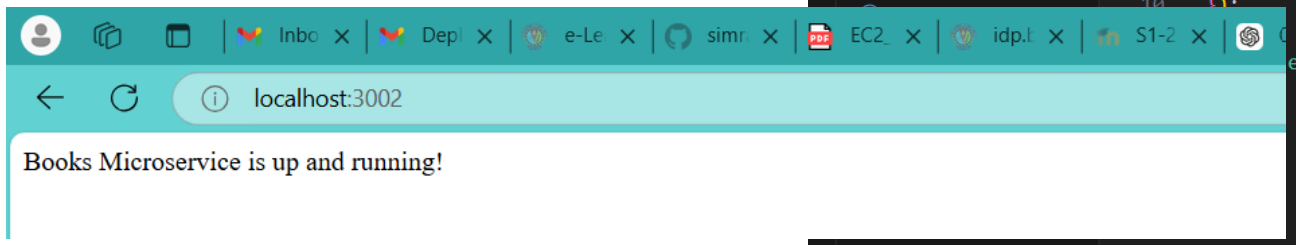
Data independence allows each service to scale and be managed individually, reducing risks of cross-service data conflicts.

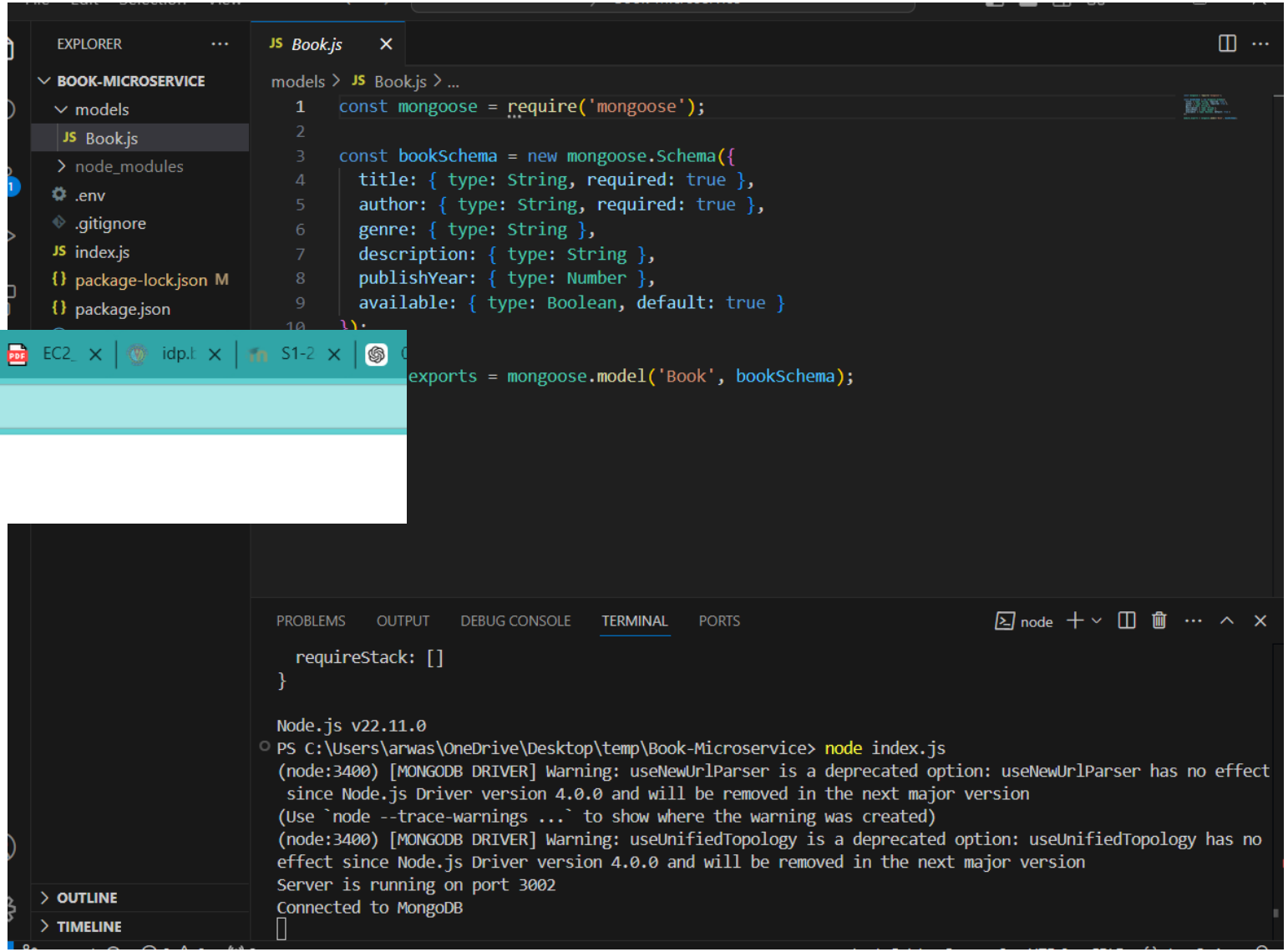
CREATED REPOSITORIES

Four Services – User, Book, Borrow and Review



Book Service Schema





CODE SNIPPETS

User Authentication and Authorization

```
46
47    // Authentication Route (example)
48    app.post('/login', async (req, res) => {
49      const { name, password } = req.body;
50      const user = await User.findOne({ name });
51
52      if (!user || !await bcrypt.compare(password, user.password)) {
53        return res.status(401).send({ message: 'Invalid credentials' });
54      }
55
56      const token = jwt.sign({ userId: user._id, role: user.role }, 'your_jwt_secret', { expiresIn: '1h' });
57      res.send({ token });
58    });
59
60    const PORT = 3001;
61    app.listen(PORT, () => console.log(`User-Service running on port ${PORT}`));
```

CODE SNIPPETS

Database Interactions – MongoDB Queries

```
56  // Get a single book by ID
57  app.get('/books/:id', async (req, res) => {
58    try {
59      const book = await Book.findById(req.params.id);
60      if (!book) {
61        return res.status(404).json({ error: 'Book not found' });
62      }
63      res.status(200).json(book);
64    } catch (error) {
65      res.status(500).json({ error: error.message });
66    }
67  });
68
69  // Update a book by ID
70  app.put('/books/:id', async (req, res) => {
71    try {
72      const book = await Book.findByIdAndUpdate(req.params.id, req.body, { new: true, runValidators: true });
73      if (!book) {
74        return res.status(404).json({ error: 'Book not found' });
75      }
76      res.status(200).json(book);
77    } catch (error) {
78      res.status(400).json({ error: error.message });
79    }
80  });
```

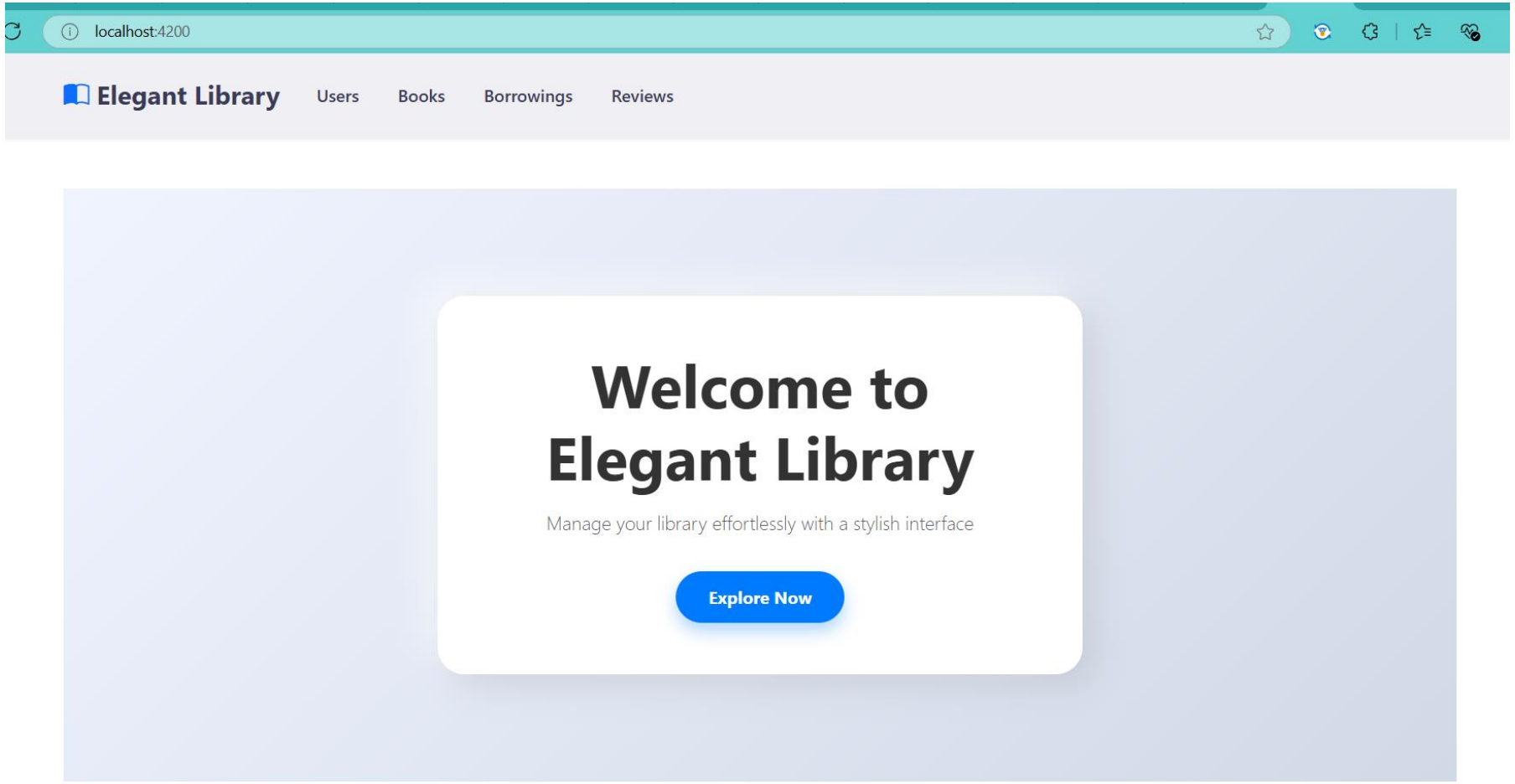
CODE SNIPPETS

API Endpoints

```
34
35    // Create a new book
36    app.post('/books', async (req, res) => {
37        try {
38          const book = new Book(req.body);
39          const savedBook = await book.save();
40          res.status(201).json(savedBook);
41        } catch (error) {
42          res.status(400).json({ error: error.message });
43        }
44      });
45
46      // Get all books
47      app.get('/books', async (req, res) => {
48        try {
49          const books = await Book.find();
50          res.status(200).json(books);
51        } catch (error) {
52          res.status(500).json({ error: error.message });
53        }
54      });
55
56      // Get a single book by ID
57      app.get('/books/:id', async (req, res) => {
58        try {
59          const book = await Book.findById(req.params.id);
```

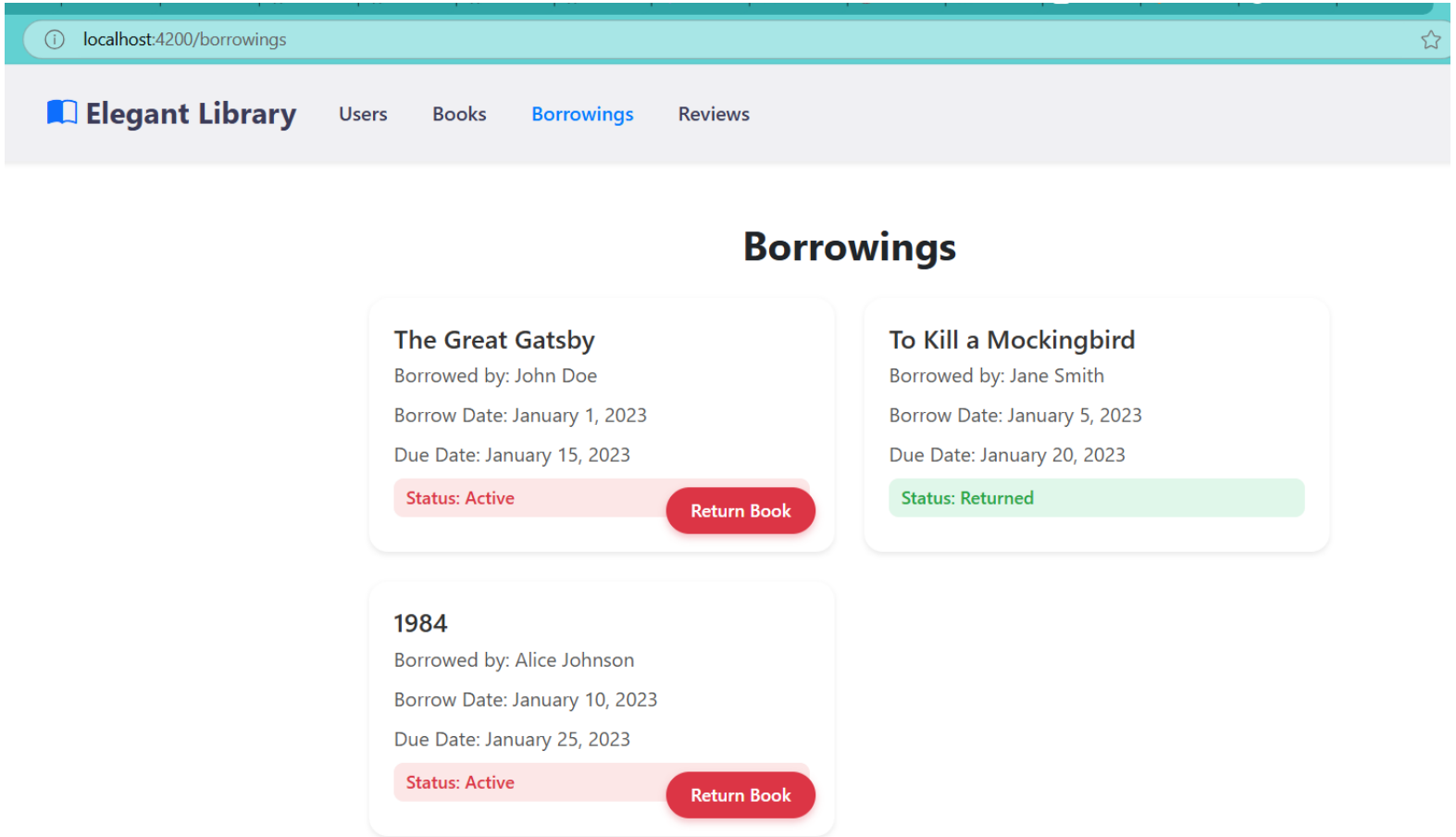
FRONTEND UI

Login Screen



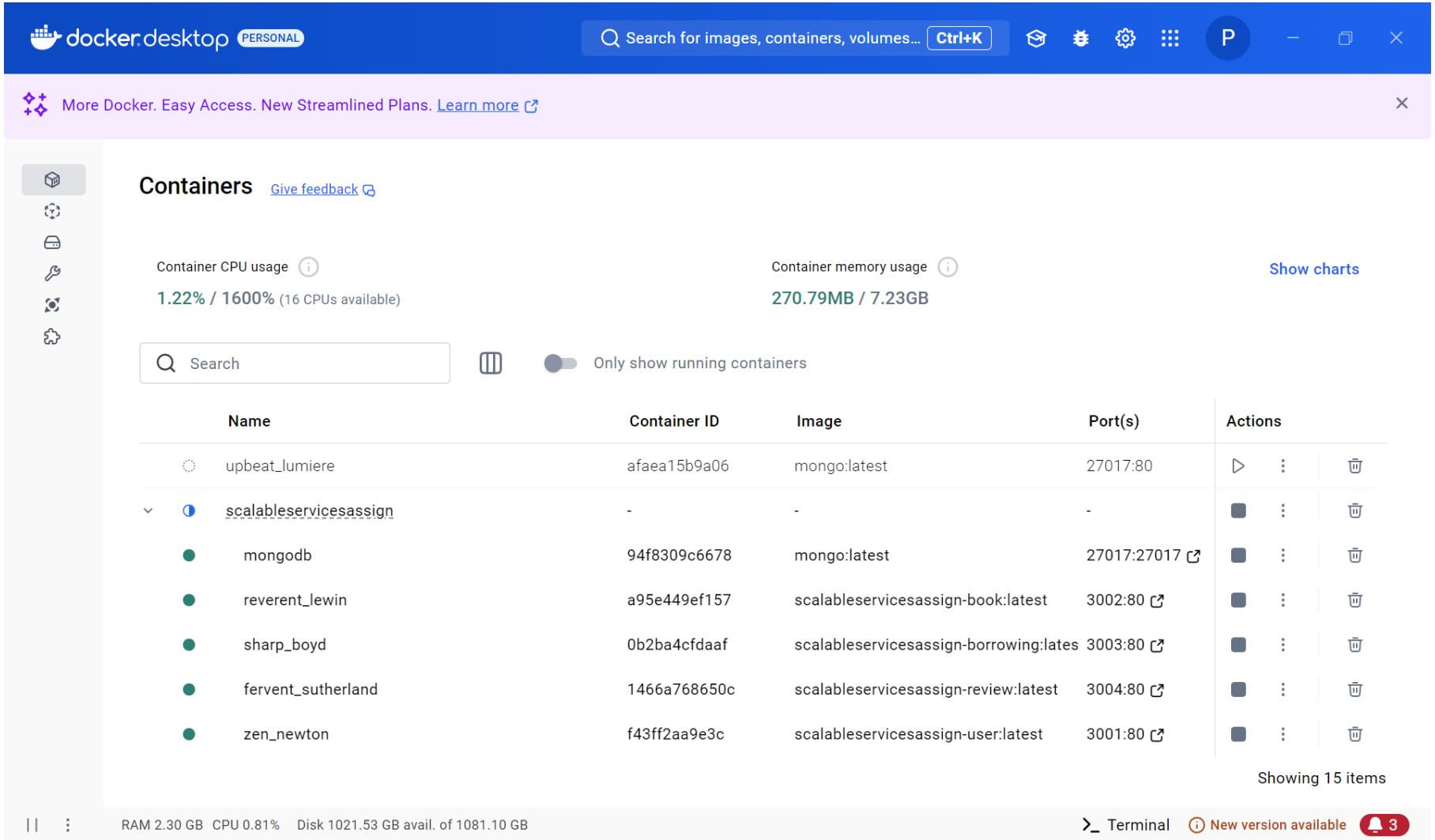
FRONTEND UI

Book Search and Browsing Page



DEPLOYMENT

Docker Deployment



DATA FLOW

1. Borrowing Flow:
   - A user sends a request to borrow a book via the API Gateway, which routes it to the Borrowing Service.
   - The Borrowing Service calls the Book Service to check availability. If the book is available, the Borrowing Service records the transaction and updates the book status.
2. Returning Flow:
   - The user initiates a return request, which the API Gateway routes to the Borrowing Service.
   - The Borrowing Service updates the transaction and notifies the Book Service to change the book's status to available.
3. Review Flow:
   - A user submits a review request for a specific book through the frontend application.
   - The frontend sends a POST /reviews request to the API Gateway, which routes the request to the Review Service.
   - If the book is valid, the Review Service saves the review to the Review Database.

DEPLOYMENT ARCHITECTURE

1. Containerization (Docker):
   - Each microservice is containerized using Docker, creating a consistent environment for deployment.
   - The containers include each microservice and its dependencies, allowing for isolated deployment and easy scaling.
2. Load Balancing and Fault Tolerance:
   - The API Gateway handles load balancing and ensures requests are distributed evenly across multiple instances of each service.
   - Docker automatically restarts failed containers and manages traffic routing, ensuring high availability and minimal downtime.

KEY LEARNINGS

1. Learned to break down a monolithic application into modular, independent services.
2. Developed proficiency in MongoDB schema design and learned about indexing, embedding, and referencing data in MongoDB collections.
3. Learned to use database patterns like Database per Service and how to manage data consistency using foreign keys and references where needed in MongoDB.
4. Learned how to implement JWT-based authentication and secure APIs. Developed an understanding of OAuth2 (even though optional) and its role in securing microservices.
5. Gained practical experience with Docker and Kubernetes, learning how to create Dockerfiles, manage containerized services, and configure Kubernetes deployments, pods, and services.
6. Understood the importance of clear documentation and regular updates within a team project.

THANK YOU !!