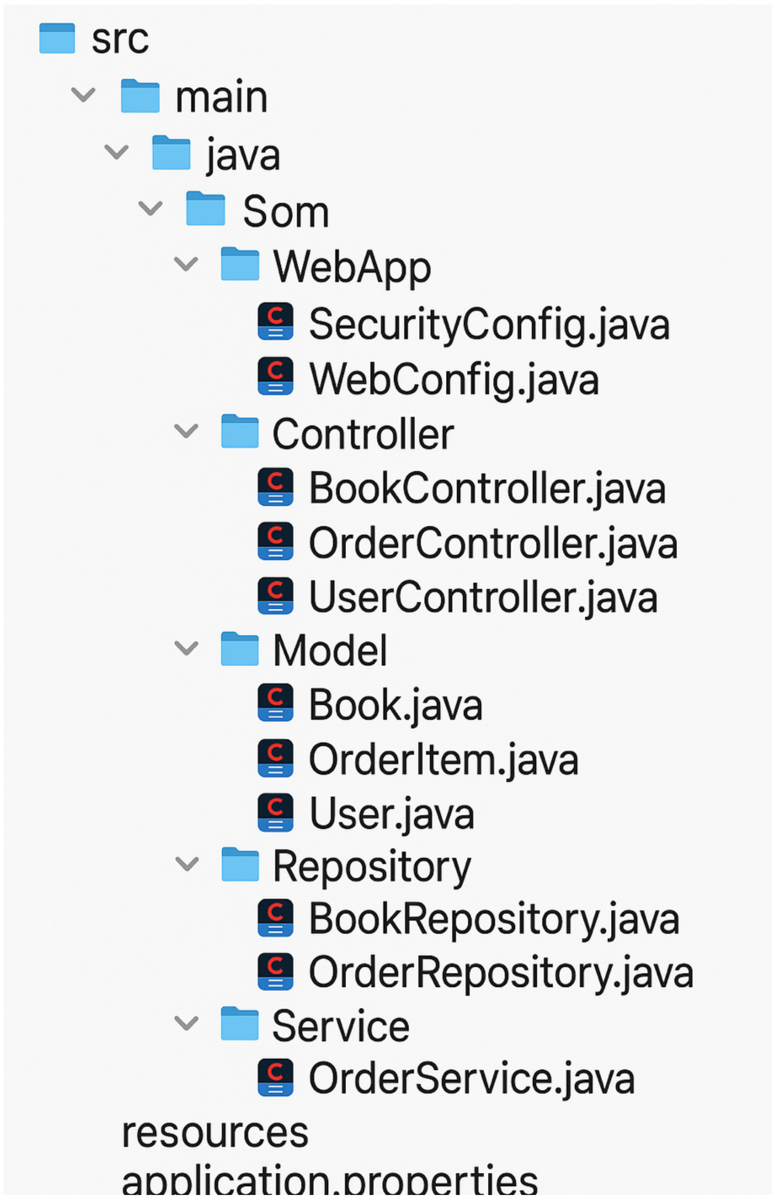
Virtual Book Store – Project Report

1. Project Structure Overview:

The Virtual Book Store is a full-Stack web application created with Spring Boot and MySQL, allowing role-based access for Users and Admin. Admins can handle the book inventory by adding or removing books, whereas users can log in, explore available books, and make purchases. The backend offers secure REST APIs and enables integration with a React frontend through CORS.

2. Project Structure:



```
src
  main
    java
      Som
        WebApp
          SecurityConfig.java
          WebConfig.java
        Controller
          BookController.java
          OrderController.java
          UserController.java
        Model
          Book.java
          OrderItem.java
          User.java
        Repository
          BookRepository.java
          OrderRepository.java
        Service
          OrderService.java
    resources
    application.properties
```

a) **SecurityConfig.java:** The SecurityConfig class in this Spring Boot application is responsible for configuring the security behavior using Spring Security. Annotated with @Configuration, it defines a bean securityFilterChain that customizes HTTP security settings. Inside this method, CSRF protection is explicitly disabled, which is a common approach for stateless REST APIs that don't rely on cookies for authentication. The authorizeHttpRequests block specifies that any requests to endpoints starting with /api/auth/, /api/admin/, /api/books/, and /api/orders/ are allowed without authentication (permitAll()), enabling open access for login, registration, or public resources like books and orders. All other endpoints require authentication (anyRequest().authenticated()). Additionally, form-based login is disabled using formLogin().disable(), implying that the application likely uses token-based authentication (such as JWT) rather than traditional login forms. Another bean, passwordEncoder, is defined to return an instance of BCryptPasswordEncoder, ensuring that any user passwords are securely hashed before being stored or compared during authentication.
4o

b) **WebConfig.java:** The WebConfig class configures Cross-Origin Resource Sharing (CORS) for a Spring Boot application to allow communication between the backend and a frontend, such as a React app running on http://localhost:3000. It defines a WebMvcConfigurer bean that overrides the addCorsMappings method to enable CORS for all API endpoints matching the /api/** pattern. This setup permits HTTP methods like GET, POST, PUT, DELETE, and OPTIONS, accepts any headers, and allows credentials such as cookies or tokens to be included in requests. This configuration is essential for enabling seamless interaction between a frontend and backend running on different domains or ports during development.

c) **BookController.java:** The BookController class functions as the REST controller for managing book-related tasks in the application, operating under the /api/admin/books endpoint. It is marked with @RestController and @CrossOrigin annotations to allow cross-origin requests from a React frontend hosted at http://localhost:3000. The controller utilizes a BookRepository for database interactions. It offers several endpoints: a GET endpoint to fetch books by genre (/genre/{genre}), another GET to retrieve all books, a POST endpoint to create a new book, a PUT endpoint to modify existing book details by ID, and a DELETE endpoint to delete a book by ID. Each endpoint communicates with the repository to execute the required CRUD operations and delivers suitable HTTP responses, making this controller essential for administrative book management.

d) **OrderController.java:** The Order class serves as a JPA entity that symbolizes an order made by a user within the application. Marked with @Entity and associated with the orders table, it employs Lombok annotations (@Data, @NoArgsConstructor, and @AllArgsConstructor) to automatically produce boilerplate code such as getters, setters, constructors, and additional features. The class includes multiple fields that record important order information like the user's email, shipping method, payment method, delivery address, and order status. It also comprises financial areas such as subtotal, tax, shipping fees, and the overall total. The items field consists of a list of OrderItem objects and is configured with @ElementCollection and @CollectionTable. This indicates that it maintains a collection of value-type objects in a distinct table (order_items) that connects to the orders table through a foreign key. Timestamps for creation and updates (createdAt and updatedAt) are included to monitor order history. This model is crucial for handling and retaining order information within the application

e) **UserController.java:** The Users class is a JPA entity that represents a user in the application. It is annotated with @Entity and leverages Lombok annotations such as @Data, @NoArgsConstructor, and @AllArgsConstructor to minimize boilerplate code by automatically generating constructors and accessor methods. The email field, designated as the primary key with the @Id annotation, uniquely identifies each user. Additional fields include the user's name, password, role (e.g., "USER" or "ADMIN"), and address, used for shipping purposes. Although Lombok simplifies code generation, the class explicitly includes getter and setter methods for all fields to ensure compatibility in environments where Lombok support may be limited or unavailable. This entity is essential for handling user data and authentication within the system.

f) **Book.java:** The Book class is a JPA entity in the com.Spring.WebApp.Model package, representing a book in the application. Annotated with @Entity and using Lombok (@Data, @NoArgsConstructor, @AllArgsConstructor), it minimizes boilerplate code. It includes fields for a unique ID (auto-generated), title, author, description, price, stock, genre, image URL, and availability, with a @Column annotation specifying a 1000-character limit for the description. This class is designed to manage book data in the database.

g) **Order.java:** The Order class models a user's order in the application and is mapped to the orders table using JPA annotations. It includes fields such as userEmail, shippingMethod, paymentMethod, address, and status, providing comprehensive details about the order's context. Financial aspects of the order—like subtotal, tax, shippingCost, and total—are stored as separate fields for clarity and computation. The items field holds a list of OrderItem objects, annotated with @ElementCollection and stored in a separate table (order_items) linked by a foreign key to the order. Additionally, the entity tracks timestamps for creation and updates via createdAt and updatedAt. Lombok annotations (@Data, @NoArgsConstructor, @AllArgsConstructor) simplify code maintenance by generating constructors and accessor methods automatically. This class forms a key part of the order processing system in your project.

h) **OrderItem.java and User.java:** The Users and OrderItem classes together support the user and order management system in the application. The Users class defines the structure of a user entity, with fields for name, email (used as the primary key), password, role, and address—ensuring that user identity and shipping details are stored. While Lombok annotations generate common methods like constructors and accessors, explicit getter and setter methods are also included to provide stability in environments where Lombok may not work. On the other hand, the OrderItem class is a simple, embeddable value object used within orders to represent individual purchased items. It contains fields for the book title, quantity, and price. As an @Embeddable class, it integrates directly into the Order entity's structure, allowing a list of items to be stored efficiently in a secondary table linked to each order. Together, these classes help manage both user data and the detailed composition of orders within the system.

i) **REPOSITORY (BookRepo,OrderRepo,UserRepo**): The UserRepository, OrderRepository, and BookRepository interfaces are Spring Data JPA repositories that abstract the database operations for their respective entities. UserRepository manages Users entities and includes a custom method findByEmailAndRole to fetch a user based on their email and role—useful for login or role-based access checks . OrderRepository handles Order entities and provides a custom method findByUserEmail to retrieve all orders associated with a specific user's email. Lastly, BookRepository manages Book entities and includes two custom query methods: findByTitleContainingIgnoreCase to support case-insensitive partial title searches and findByGenreIgnoreCase to retrieve books by genre regardless of case. These repositories play a vital role in separating persistence logic from business logic, simplifying data access across the application.

j) **OrderService.java:** The OrderService class functions as a service layer component responsible for managing the business logic of orders within the application. Marked with the @Service annotation, it employs @Autowired for dependency injection to interact with the OrderRepository. The class offers methods to create a new order (saveOrder), retrieve all orders (getAllOrders), fetch orders linked to a user's email (getOrdersByUserEmail), and obtain a specific order by its ID (getOrderById). It also provides an updateOrderStatus method, which finds an order by ID, modifies its status, updates the updatedAt timestamp, and saves the changes. This service layer ensures a clear separation of concerns by acting as a bridge between the controller and repository, encapsulating all order-related processing logic.

3. Flow of Execution:

The functionality of the project follows a structured flow starting from the frontend to the backend and database. When a user interacts with the React frontend running at localhost:3000, it sends HTTP requests to the backend Spring Boot application through defined API endpoints such as /api/admin/books or /api/orders. These requests are received by controller classes like BookController, which map the routes using annotations such as @RestController and @RequestMapping. The controller handles the incoming data and delegates logic to service classes where needed—for instance, OrderService handles operations like updating order status or fetching orders by email. These service or controller classes then communicate with the repository layer, which consists of interfaces like BookRepository, UserRepository, and OrderRepository. These interfaces use Spring Data JPA to interact with the database, automatically executing SQL queries based on method names such as findByGenreIgnoreCase. The data fetched from or written to the database is mapped to entity classes like Book, Order, Users, and OrderItem, which are annotated with JPA annotations such as @Entity, @Id, and @ElementCollection. Once the database operation completes, the response is returned up the chain to the controller, which sends it back to the frontend as a JSON response. Additionally, WebConfig enables cross-origin requests from the React frontend, while SecurityConfig manages route accessibility and password encoding using BCrypt. Overall, this setup ensures a clean and modular architecture where the flow of data is well-structured between user interface, controller logic, service processing, repository access, and database persistence.

4.API’s in the Project:

BookController:
GET /api/admin/books — Fetch all books.
GET /api/admin/books/genre/{genre} — Fetch books by genre.
POST /api/admin/books — Add a new book.
PUT /api/admin/books/{id} — Update an existing book by ID.
DELETE /api/admin/books/{id} — Delete a book by ID.

From OrderController
GET /api/orders — Get all orders.
GET /api/orders/user/{email} — Get orders by user email.
GET /api/orders/{id} — Get a specific order by ID.
POST /api/orders — Place/save a new order.
PUT /api/orders/{id}/status — Update order status by ID.

From SecurityConfig
POST /api/auth/login — User login.
POST /api/auth/register — User registration.

5.Properties File:

1. spring.application.name=WebApp (Setting the name of the Spring Boot application)
2.spring.datasource.url=jdbc:mysql://localhost:3306/app_database?useSSL=false& serverTimezone=UTC (Connects to a MySQL database running locally on port 3306, The name of the database schema, Sets the server time zone to UTC to avoid time zone-related issues.)
3. **spring.datasource.username=root** (Provides the username for authenticating with the MySQL database.)
4. Spring.datasource.password=Tump3r@1999 (password for the database)
5. spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver (Defines the JDBC driver class for the database)
5. spring.jpa.hibernate.ddl-auto=update (Hibernate automatically updates the database schema based on entity classes (creating or modifying tables) without dropping existing data)
6. spring.jpa.show-sql=true (SQL queries executed by the application are printed to the console)

6.POM File:

It sets up the necessary dependencies for web development, database management, security, and testing.
Dependencies added:

- **spring-boot-starter-web**: For building RESTful web applications.
- **spring-boot-starter-data-jpa**: For database interaction using JPA/Hibernate.
- **spring-boot-starter-security:** For adding security features (e.g., authentication, authorization).

- **spring-boot-devtools**: For development-time features like auto-restart (runtime, optional).

- **mysql-connector-j:** MySQL JDBC driver for database connectivity (runtime scope).

- **lombok:** Reduces boilerplate code with annotations (optional).

    - spring-boot-starter-test: For testing the application.
    - spring-security-test: For testing security configurations

7.Database Tables:

1. **book Table**: Stores book details.

2. **order Table**: Stores order information.

3**. users Table**: Stores user information.

8.Frontend Folder Structure:

```
virtual bookstore/
├── node_modules/           # Installed dependencies
├── public/                # Static assets (index.html, favicon, etc.)
├── src/                   # Source code for the React app
│   ├── components/          # Reusable UI components (e.g., navbar, cards)
│   ├── config/            # Configuration files/constants (e.g., API base URL)
│   ├── contexts/          # React Contexts for global state management
│   ├── pages/             # Full page components (e.g., Admin Dashboard, Book
List)
│   ├── services/          # API service modules (e.g., ordersService.js)
│   ├── App.js             # Root component
│   ├── App.css            # App-wide styling
│   ├── App.test.js        # Unit tests for App component
│   ├── index.js           # Entry point of the application
│   ├── index.css          # Global styles
│   ├── logo.svg           # App logo
│   ├── reportWebVitals.js      # Performance reporting (optional)
│   └── setupTests.js        # Testing setup for Jest
├── README.md               # Project documentation
├── package.json           # Project metadata and dependencies
├── package-lock.json        # Exact dependency tree
```

The Virtual Bookstore project is a React-based web application organized using a modular structure that follows modern best practices. At the root level, it includes essential configuration and metadata files such as package.json, package-lock.json, and a README.md for documentation. The node_modules directory contains all installed dependencies. Static assets like index.html and the favicon are stored in the public folder. The main application logic resides in the src directory, which includes core files like index.js (the entry point), App.js (the root component), and styling files such as App.css and index.css. The src folder is further divided into subdirectories: components holds reusable UI components like navbars and cards, pages contains full-page views like the Admin Dashboard and Book List, contexts manages global state using React Context, config stores configuration constants (e.g., API base URLs), and services handles API interactions through service modules. Additionally, the project includes testing files like App.test.js for unit testing and setupTests.js for setting up the Jest environment.

9. Technologies:

1. Spring Boot, Java 21, Spring Data JPA, Spring Security, React.js.

2. MySQL with mysql-connector-j

3. Lombok for reducing boilerplate code

4. Maven for dependency management

10.Conclusion:

Achievements: Successful implementation of a web-based bookstore with core functionalities.

Learned: Insights on Spring Boot, REST API design, and database management.

Future Enhancements: Adding payment gateways, book reviews, or advanced search filters.

11. References:

Official documentation for Spring Boot, Spring Data JPA, Spring Security, MySQL, and React.

Maven documentation for POM configuration & MVN Repository.

12. Team Members:

Student	Role	Responsibilities
Anudeep Pulluri	Backend Developer	Develop Spring Boot Rest API's & Implement business logic using java core principles. Manage session control, error handling, and logging.
Sujith Kumar Anumolu	Frontend Developer	Create user-friendly interface using React.js Implement UI elements with Bootstrap and custom CSS for responsiveness.

Soma Sekhar Bobbala	Database Engineer	Design and implement the MySQL database schema. Write SQL scripts to create tables, manage relationships, and ensure date integrity.
Ashwith Reddy Surkanti	API Developer & Tester	Develop and document RESTful endpoints. User Postman to test and validate all API functionalities. Assist in integrating frontend with backend APIs.