**1. Project Overview**

This documentation provides an in-depth view of our **Bricolage Store** web application. The application’s primary purpose is to allow **users** to browse and purchase products from different categories, manage orders, and provide delivery details. **Administrators** can manage products, oversee orders, and confirm/cancel orders as needed.

The project is built with:

* **Java 21 (Spring Boot)** for the back-end.
* **React** (with **Material UI**) for the front-end.
* **PostgreSQL** as the database, with **Liquibase** for database version control.

The application is containerized using **Docker**, with a docker-compose.yml file orchestrating services for:

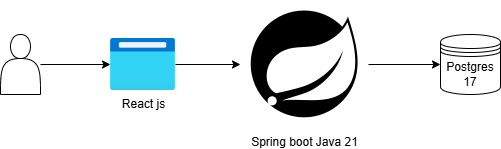
1. **PostgreSQL** database
2. **Liquibase** migration service
3. **Spring Boot** backend

This documentation covers:

* Architecture and Technology Stack
* Detailed Service Interfaces and Implementations
* Security (JWT, Spring Security)
* Searching and Filtering (via JPA Specifications)
* Deployment and Installation
* Project Structure

**2. Architecture and Technology Stack**

**2.1 High-Level Architecture**

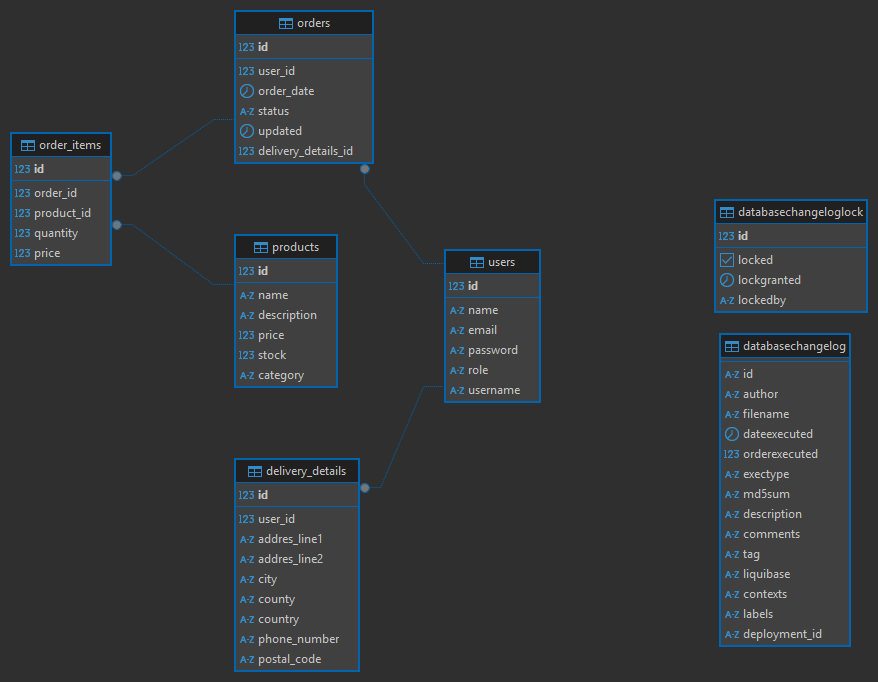
****

1. **Frontend (React + Material UI):** Provides a responsive user interface for customers and administrators.
2. **Backend (Spring Boot Java 21):** Exposes RESTful APIs, handles business logic, security, and interactions with the database.
3. **Database (PostgreSQL):** Stores user accounts, orders, products, and other data. Version control handled by **Liquibase**.

**2.2 Technology Stack Overview**

* **Java 21**: Main programming language.
* **Spring Boot**: For rapid application development, built-in Tomcat server, and production-ready features.
* **Spring Security + JWT**: Provides authentication, authorization, and token management.
* **React + Material UI**: Modern, responsive, component-based frontend application.
* **JPA (Hibernate)**: Object-relational mapping (ORM) for database interactions.
* **Liquibase**: Database version migration and schema management.
* **PostgreSQL**: Open-source relational database.
* **Docker & Docker Compose**: Containerization for easy deployment.
* **JUnit**: Unit testing framework for backend services.
* **Jakarta & Lombok**: For annotations, boilerplate reduction (getters, setters, etc.).

**2.3 Database Architecture**

****

**3. Backend Services & Interfaces**

The backend is organized into **service** and **service/interface** packages. Each interface defines the contract (methods available), while the corresponding implementation resides under the service package.

Below are the main service interfaces, their methods, and brief descriptions.

**3.1 User Service Interface**

Located under: service/interface/UserService.java

**Methods**:

1. createUser(User user): Creates a new user account.
2. updateUser(Long userId, User userDetails): Updates a user’s profile or credentials.
3. deleteUser(Long userId): Deletes a user by ID.
4. getUserById(Long userId): Retrieves user details by ID.
5. getUserByUsername(String username): Retrieves user details by username.
6. listUsers(): Lists all users (admin-only operation).

**Purpose**: Manages user profiles and credentials. Includes user creation (registration), profile updates, and retrieval.

**3.2 Delivery Details Service Interface**

Located under: service/interface/DeliveryDetailsService.java

**Methods**:

1. createDeliveryDetails(Long userId, DeliveryDetails details): Creates new delivery details for a user.
2. updateDeliveryDetails(Long detailId, DeliveryDetails details): Updates existing delivery details.
3. deleteDeliveryDetails(Long detailId): Deletes specified delivery details.
4. getDeliveryDetails(Long detailId): Retrieves a specific delivery detail by ID.
5. listDeliveryDetailsByUser(Long userId): Lists all delivery details for a given user.

**Purpose**: Handles CRUD operations for user delivery addresses and preferences.

**3.3 Order Service Interface**

Located under: service/interface/OrderService.java

**Methods**:

1. createOrder(Order order): Creates a new order (admin-initiated, or triggered by internal logic).
2. updateOrder(Long orderId, Order orderDetails): Updates an existing order (admin-only after submission).
3. deleteOrder(Long orderId): Deletes/cancels an order.
4. getOrderById(Long orderId): Retrieves a specific order by ID.
5. listAllOrders(): Lists all orders (admin-only).
6. searchOrders(SearchCriteria criteria): Searches for orders by product name, price range, date range.

**Purpose**: Provides administrative CRUD access to orders and searching functionality.

**3.4 Order Management Service Interface**

Located under: service/interface/OrderManagementService.java

**Methods**:

1. addItemToOrder(Long userId, Long productId, int quantity): Allows a user to add items to their cart (an “open” order).
2. removeItemFromOrder(Long userId, Long orderItemId): Removes an item from the user’s cart.
3. submitOrder(Long userId): Submits the current cart as a finalized order.
4. confirmOrder(Long orderId): Admin operation to confirm an order (ensures product stock is available).
5. cancelOrder(Long orderId): Admin operation to cancel an order at user request.
6. modifyOrderItem(Long orderId, Long orderItemId, int newQuantity): Admin operation to modify an order item after submission if needed.

**Purpose**: Handles user-driven and admin-driven order logic. Ensures that only admins can modify a submitted order.

**3.5 Product Service Interface**

Located under: service/interface/ProductService.java

**Methods**:

1. createProduct(Product product): Creates a new product in the catalog.
2. updateProduct(Long productId, Product productDetails): Updates product information.
3. deleteProduct(Long productId): Deletes (or marks unavailable) a product.
4. getProductById(Long productId): Retrieves product information by ID.
5. listAllProducts(): Lists all products.
6. searchProducts(String name, String description): Searches for products by name and/or description.

**Purpose**: Manages the product catalog, including search capabilities.

**3.6 Token Blacklist Service Interface**

Located under: service/interface/TokenBlacklistService.java

**Methods**:

1. addTokenToBlacklist(String token): Adds a JWT to the blacklist (e.g., on logout).
2. isTokenBlacklisted(String token): Checks if a given token is blacklisted.
3. removeExpiredTokens(): Cleans up tokens that are expired.

**Purpose**: Ensures that once a user logs out or invalidates a token, it cannot be reused, even if it has not yet reached its expiration time.

**4. Security & Authentication**

**4.1 Overview**

* **Spring Security** is used to secure the REST endpoints.
* **JWT** (JSON Web Tokens) are used to authenticate requests.
* Tokens are signed and have a 15-minute validity. A separate **refresh token** is valid for 2 hours.
* For each request, the token is inspected. If valid and not blacklisted, the request proceeds.
* The user’s role (e.g., ROLE\_ADMIN, ROLE\_USER), username, and expiration are encoded in the token.
* Unauthorized requests are redirected to the login page or receive an HTTP 401/403 error in case of API calls.

**4.2 Token Refresh Flow**

1. **User logs in** with valid credentials, receiving two tokens: an **access token (15 mins)** and a **refresh token (2 hours)**.
2. **Access token** is used for most requests. When it expires, the client can request a new token using the **refresh token**.
3. **Logout** triggers the addition of both tokens to the **Token Blacklist**, preventing reuse.

**4.3 Endpoint Security**

* /auth/\*\* endpoints are open to all users (login, register, token refresh).
* Other endpoints require valid **JWT** and a **role** with enough privileges:
  + **ADMIN** can manage products, orders, and users.
  + **USER** can manage their profile, cart, and submit orders.

**5. Frontend (React + Material UI)**

**5.1 Overview**

The frontend is a single-page application built with **React**. It uses **Material UI** components for the visual design and layout.

**Main Features:**

* **Product Browsing**: List of products, search by name or description.
* **User Cart**: Add, remove, and update items. Submitting the cart to create an order.
* **Delivery Details**: Add multiple addresses, update or remove existing ones.
* **Order Tracking**: Users can see their submitted orders and statuses.
* **Admin Panel**: Manage products, see all orders, confirm/cancel or modify orders.

**5.2 Running the Frontend**

1. Navigate to the frontend directory.
2. Run npm install.
3. Run npm start.
4. The app is typically available at http://localhost:3000.

**6. Search & Filtering (JPA Specifications)**

The project supports advanced search queries using **JPA Specifications**. These are defined under the specifications folder. The main idea is to build dynamic queries at runtime based on user input.

1. **Order Search**:
   * By **product name**: Finds orders whose items match the given product name.
   * By **price range**: Finds orders with total amounts within a specified range.
   * By **date range**: Finds orders placed between two dates.
2. **Product Search**:
   * By **name** and **description**: Uses partial matches (like %keyword%) to find relevant products.

These features provide flexible, user-friendly filtering for both **admin** and **user** views.

**7. Deployment & Installation**

**7.1 Docker & Docker Compose**

**7.1.1 Prerequisites**

* **Docker** installed (version >= 20.x recommended).
* **Docker Compose** installed.

**7.1.2 Steps to Run**

1. **Clone the repository** containing docker-compose.yml, Dockerfile, and Docker.postgres.
2. Open a terminal in the project’s root directory (where docker-compose.yml is located).
3. Run:

docker-compose build

docker-compose up -d

1. This will spin up the following containers:
   * **Postgres** database
   * **Liquibase** (which automatically applies database migrations)
   * **Spring Boot** application (exposing REST APIs typically on port **8080**)

Once containers are up, you can verify by checking:

docker ps

Or by opening http://localhost:8080 in your browser (or an API testing tool).

**7.1.3 Verifying Services**

* **Liquibase logs**: Check the migrations are applied successfully.
* **Spring Boot logs**: Confirm the application started with no errors.

**7.2 Frontend**

* Ensure **Node.js** (v14 or newer) and **npm** are installed.
* Move to frontend/ directory and execute:

npm install

npm start

* The frontend runs on http://localhost:3000 by default.

**8. Additional Considerations**

**8.1 Order Lifecycle**

1. **Cart Creation**: When a user without an open order accesses their cart, an empty order is created in the backend.
2. **Adding Items**: User adds products to this cart (order).
3. **Submitting Order**:
   * User finalizes the cart.
   * The system sets the order status to “SUBMITTED.”
   * Stock checks are performed at the admin confirmation step.
4. **Admin Confirmation**:
   * Admin sees the order in a pending queue.
   * Confirms if all products are in stock.
   * Finalizes the order and updates inventory.
5. **Cancellation/Modification**:
   * Users cannot change the order once submitted.
   * Admin can modify or cancel the order upon user request.

**8.2 Token Blacklist Workflow**

* **Logout**: Immediately adds the token to the blacklist.
* **Scheduled Cleanup**: A scheduled job (e.g., once an hour) or on each request, the system removes expired tokens from the blacklist.

**8.3 Testing & Quality Assurance**

* **Unit Tests**: JUnit tests for service layers.
* **Integration Tests**: Spring Boot integration tests for REST controllers (optional coverage).
* **Frontend Testing**: React Testing Library or Jest (where applicable).

**9. Conclusion**

The **Bricolage Store** project demonstrates a robust full-stack application with microservice-like separation of concerns:

* **React** handles the user experience.
* **Spring Boot** manages business logic, security, and data persistence.
* **Docker & Docker Compose** streamline deployment and ensure reproducible builds.
* **JWT** secures the application with short-lived tokens and refresh logic.
* **JPA Specifications** allow flexible searching for both products and orders.

With this documentation, developers and administrators should be able to **run**, **maintain**, and **extend** the application. For detailed code references, please see the provided source files and configurations within the repository.

**10. Screenshots**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated