**1. Introduction**

This document outlines the system architecture for the National Asset Price Tracker (NAP Tracker), a full-stack web application that allows administrators to manage asset price data and provides users with detailed, interactive dashboards to explore asset trends.

**Purpose**

To provide a clear and comprehensive description of the system's structure, components, data flow, and interaction between subsystems.

**Scope**

* **Admin Side:** Dashboard for adding, editing, deleting assets, and viewing analytics.
* **User Side:** Dashboard displaying assets with images, price history, and detailed information.

**Target Audience**

Software developers, system architects, product managers, and technical stakeholders.

**2. System Overview**

**Functional Modules**

* **Admin Asset Management**

Enables administrators to create, update, and delete asset records within the system. This includes entering asset metadata such as name, category, price, description, and uploading associated images. Admins can also view logs of changes and monitor asset updates in real time.

* **Asset Detail Visualization**

Both admins and users can access a detailed view of each asset. This includes full metadata, the current price, associated category, and a visual layout with images. It supports navigation from summarized views (like tables or cards) to a detailed page.

* **Price History and Trend Analysis**

Displays historical price changes for each asset using interactive charts. Users can observe trends such as rising, falling, or stable assets over specific date ranges. Admins can input and manage historical price data, enabling insight-driven decisions.

* **Image Integration for Assets**

Assets are enhanced with high-quality images to offer a more intuitive and visual experience. Images are uploaded by admins and rendered on both dashboards and detail views, allowing users to recognize and distinguish between asset types quickly.

**System Actors**

* **Admin**The administrator has full control over the system’s content and structure. Admin privileges include full CRUD (Create, Read, Update, Delete) operations on all asset-related data, category management, log access, and control over historical price inputs. Admins are also responsible for maintaining data integrity and visibility.
* **User**End users have read-only access to explore assets within the system. They can view detailed asset profiles, including images and historical price trends. Users can navigate through categories, search or filter assets, and interact with visual representations such as charts or tables. They do not have permission to modify or manage data.

**3. Architectural Style**

* **Mindmap**

This mind map presents a high-level visual breakdown of the architecture and capabilities of the **National Asset Price Tracker** system. It is structured around five core branches that reflect the major design pillars of the application:

**1. User Types**

Divided into two roles:

* **Admin**
  + *Manage Assets*: Full control over assets (Add, Edit, Delete)
  + *View Price Trends*: Monitor asset price fluctuations over time
  + *Review Logs*: View system activity logs for auditing and transparency
* **User**
  + *View Assets with Images*: Access a visual catalog of assets
  + *See Price History*: Examine historical trends of asset prices
  + *Explore Categories*: Filter and browse assets by their assigned category

**2. Frontend Components (React)**

This branch emphasizes the **client-side architecture**, built using React. It includes modular UI components and pages responsible for:

* Displaying assets and charts
* Managing user interaction
* Routing between pages like dashboard, asset details, and price views

**3. Backend Services (Spring Boot)**

Highlights the **server-side logic**:

* Handles REST API requests from the frontend
* Processes data via service classes (e.g., AssetService, LogService)
* Connects to the database through repositories

**4. Database Tables**

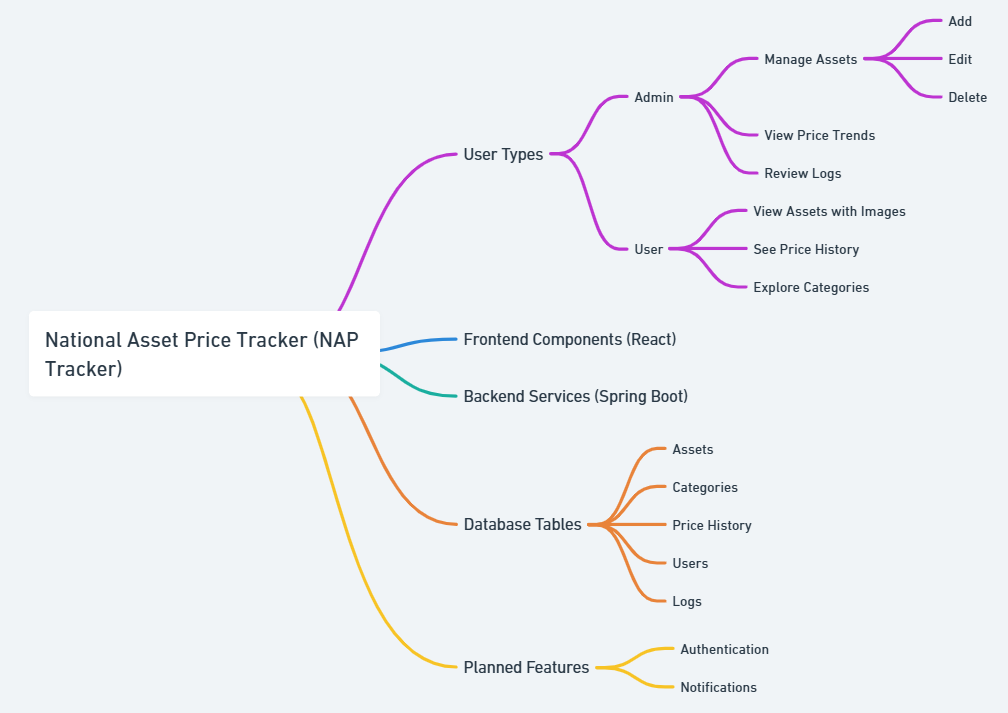
Illustrates the core data models:

* **Assets** – Core entity containing asset information
* **Categories** – Used to group assets logically
* **Price History** – Stores time-series data for each asset
* **Users** – Manages login credentials and roles
* **Logs** – Tracks admin actions for auditing purposes

**5. Planned Features**

Outlines **future system enhancements**:

* **Authentication** – Role-based login (Admin, User)
* **Notifications** – Optional alerts for price changes or system updates



* **Primary Architecture**: Client-Server

This diagram illustrates the Primary Architecture of the National Asset Price Tracker (NAP Tracker) system using the classic Client-Server Model.

**1. Client**

* Built using React.
* Handles the user interface and user interactions.
* Sends HTTP requests to the backend when a user performs actions like viewing asset details, adding new data (admin), or filtering categories.

**2. HTTP Requests**

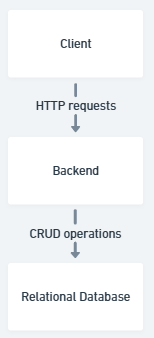
* Standard RESTful API calls (GET, POST, PUT, DELETE).
* Sent from the frontend to the backend to retrieve or manipulate data.

**3. Backend**

* Implemented with Spring Boot (Java).
* Contains:
  + Controllers to handle incoming HTTP requests
  + Services to manage business logic
  + Repositories to access the database
* Converts frontend requests into logical actions.

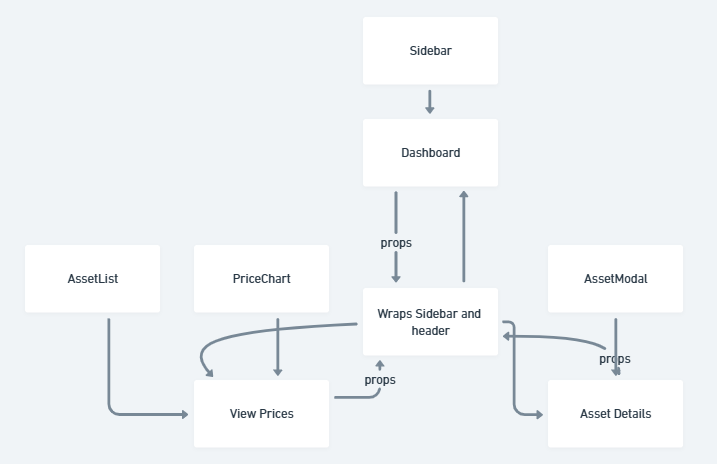
**4. Relational Database**

* A **PostgreSQL (or MySQL)** database stores the application's persistent data.
* Tables include: assets, categories, price\_history, users, and logs.
* Accessed via JPA repositories in the backend.

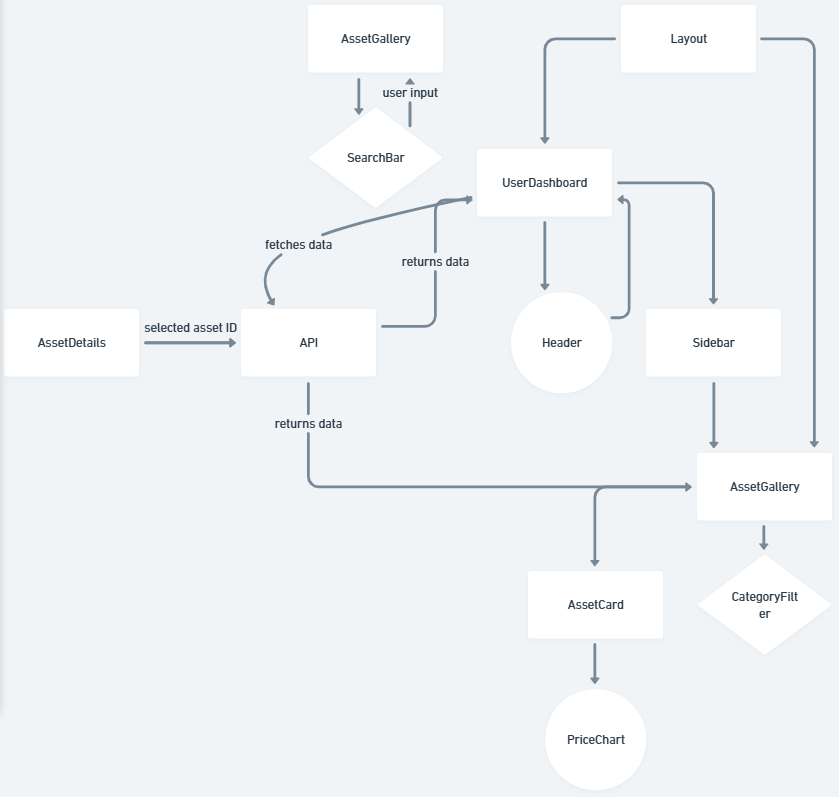


* **Frontend Pattern**: Modular, Component-Based (React)

**Admin**

****

**Normal User:**

****

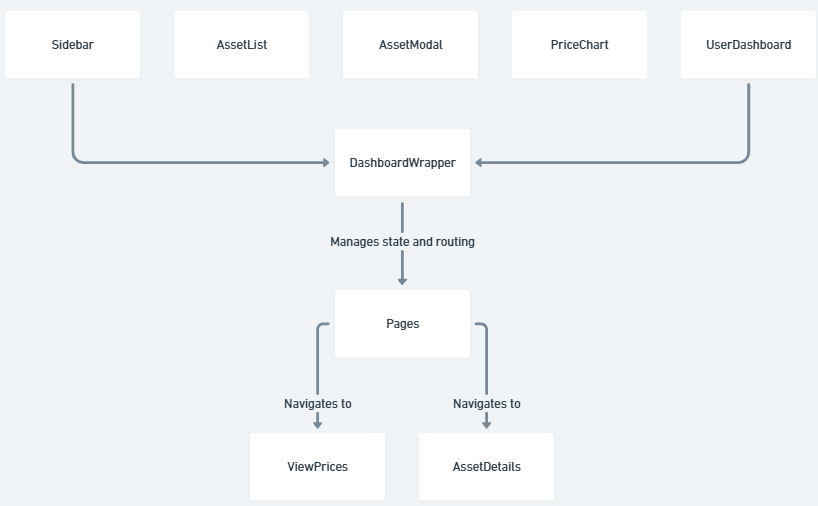
* **Backend Pattern**: Layered Architecture (Controller-Service-Repository)

Technologies:

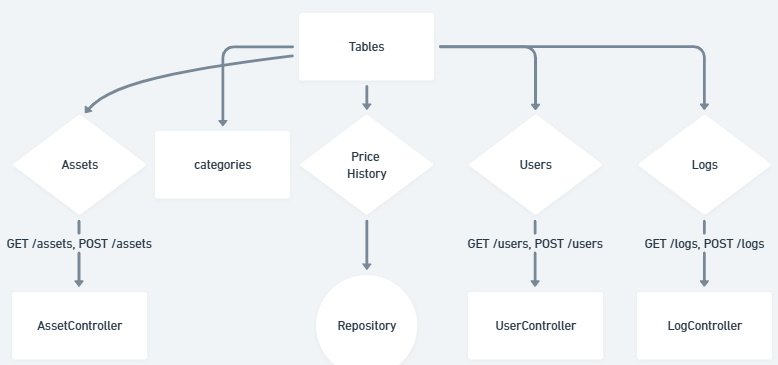
* **Frontend**: React, Bootstrap, Recharts
* **Backend**: Java, Spring Boot, Maven
* **Database**: PostgreSQL
* **Communication**: RESTful APIs

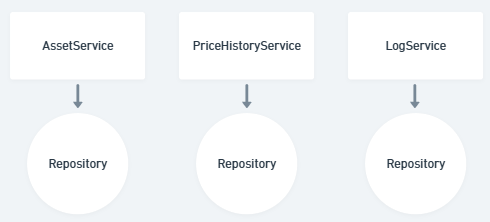
**4. Component Diagram**

* Show system broken into components:
  + Frontend: React (pages, components, routing)

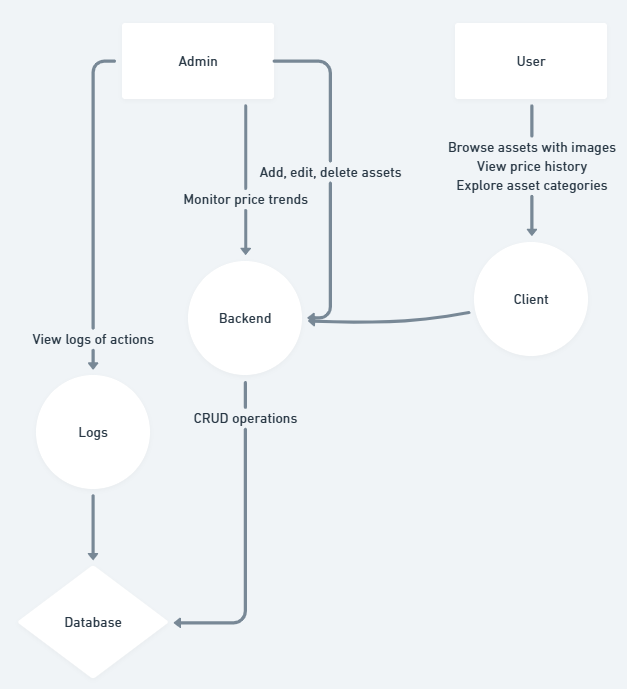


* + Backend: Spring Boot (controllers, services, models)



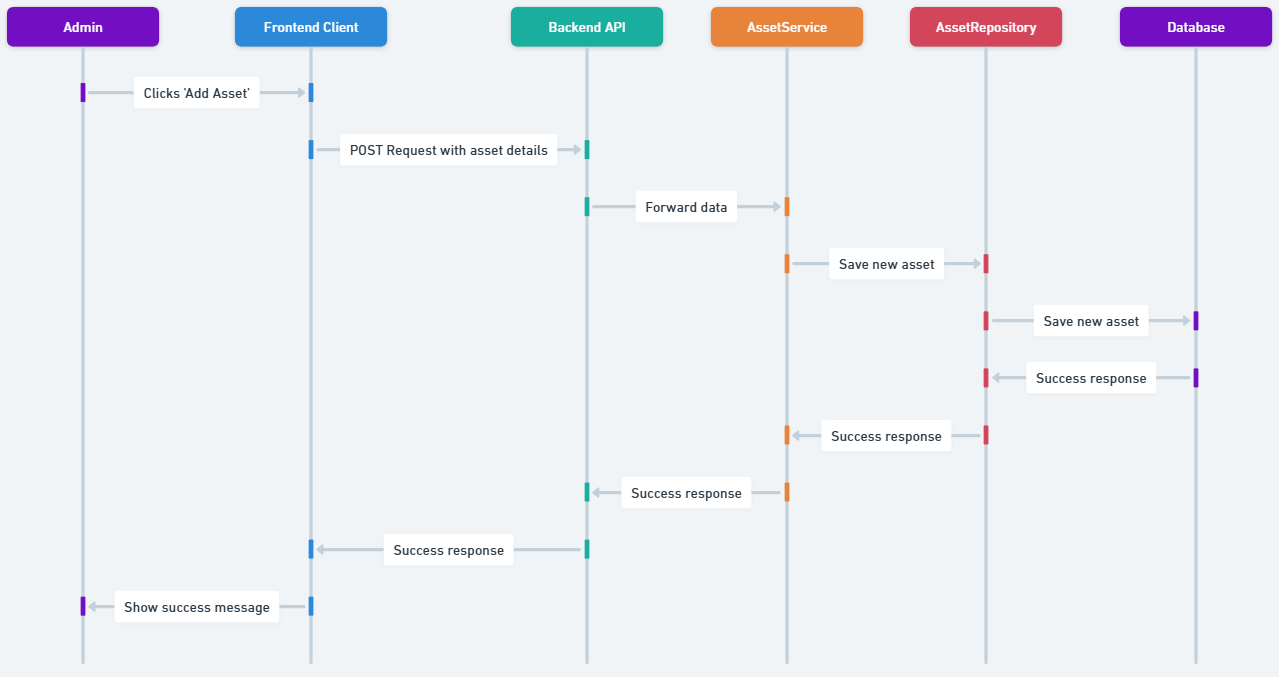


**5. Data Flow Diagram (DFD)**

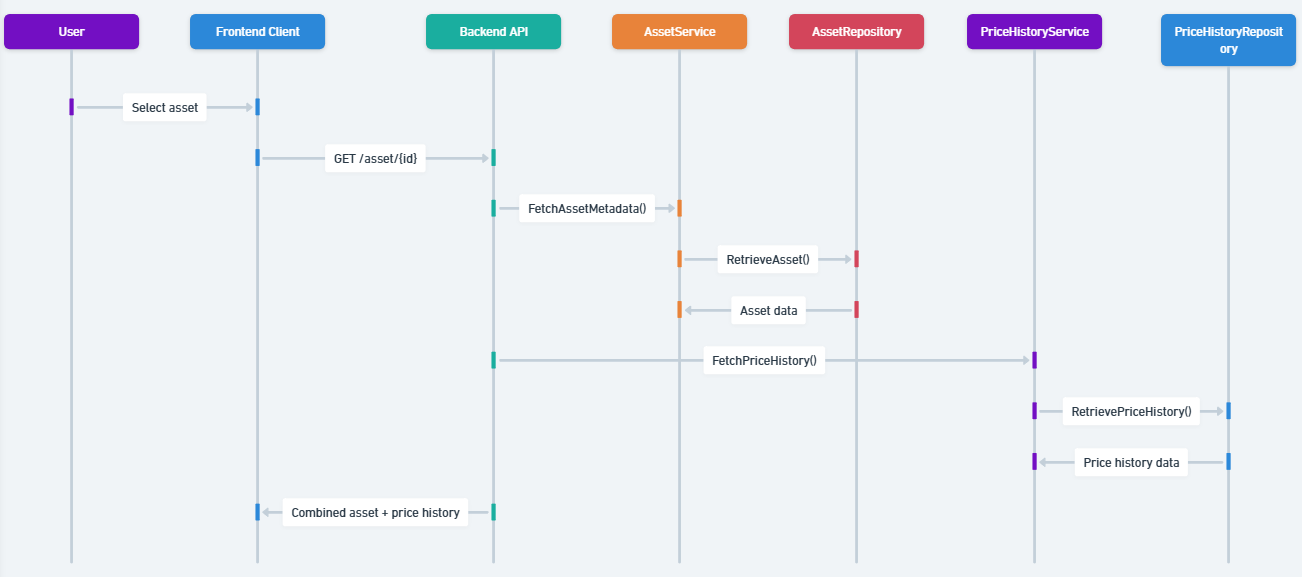


**6. Sequence diagram**

**Admin**

****

**Normal User**

****

**6. Frontend Architecture**

**Frontend folder structure**

src/

components/

AssetList.js

AssetModal.js

ConfirmDeleteModal.js

DashboardWrapper.js

Layout.js

PriceChart.js

Sidebar.js

pages/

AssetDetails.js

Dashboard.js

ViewPrices.js

App.js

App.css

**Backend folder architecture**

com.naptracker

├── controller

│ ├── AssetController.java

│ ├── CategoryController.java

│ ├── PriceHistoryController.java

│ ├── UserController.java

│ └── LogController.java

│

├── service

│ ├── AssetService.java

│ ├── CategoryService.java

│ ├── PriceHistoryService.java

│ ├── UserService.java

│ └── LogService.java

│

├── repository

│ ├── AssetRepository.java

│ ├── CategoryRepository.java

│ ├── PriceHistoryRepository.java

│ ├── UserRepository.java

│ └── LogRepository.java

│

├── model

│ ├── Asset.java

│ ├── Category.java

│ ├── PriceHistory.java

│ ├── User.java

│ └── Log.java

│

├── dto

│ ├── AssetDTO.java

│ ├── UserDTO.java

│ └── ResponseDTO.java

│

├── exception

│ ├── ResourceNotFoundException.java

│ └── GlobalExceptionHandler.java

│

├── config

│ └── WebSecurityConfig.java // For JWT/security if needed

│

└── NapTrackerApplication.java

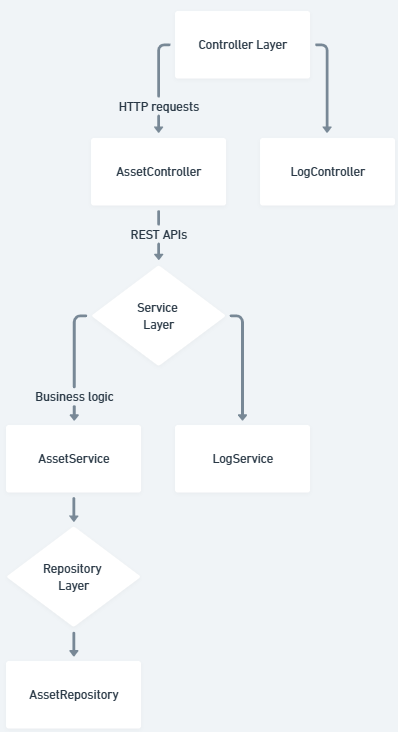
**Key Components**

* **Sidebar**: Navigation and quick actions (Add Asset)
* **Dashboard**: Admin analytics and asset table
* **AssetDetails**: View images, metadata, and history
* **DashboardWrapper**: Lifts modal state for layout-wide access

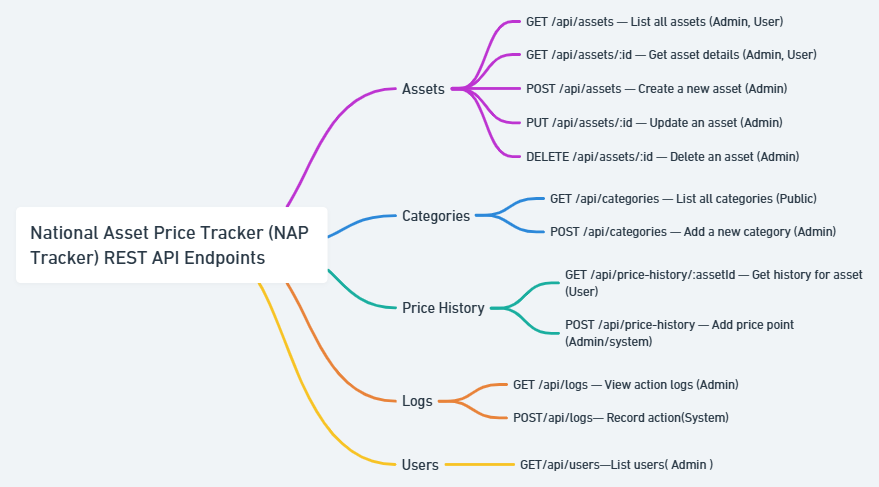
**7. Backend Architecture**

**Layered Architecture**

* **Controller Layer**: Handles HTTP requests
* **Service Layer**: Business logic
* **Repository Layer**: Data persistence



**Endpoints**

****

**8. Database Design**

**Tables and Descriptions**

|  |  |
| --- | --- |
| **Table Name** | **Description** |
| assets | Stores asset details, price, category, and status. |
| categories | Lookup table for asset categories. |
| price\_history | Historical prices for each asset. |
| logs | Tracks admin actions on assets. |
| users | Stores admin and user login data. |

**Detailed Schema**

**assets**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| id | int (PK) | Unique identifier |
| name | varchar | Asset name |
| category\_id | int (FK) | References categories.id |
| price | decimal | Current price of asset |
| description | text | Detailed asset description |
| image\_url | varchar | Path to asset image |
| trend | varchar | Asset trend (e.g., Rising, Falling) |
| updated\_at | datetime | Last update timestamp |

**categories**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| id | int (PK) | Unique category ID |
| name | varchar | Name of the category |

**price\_history**

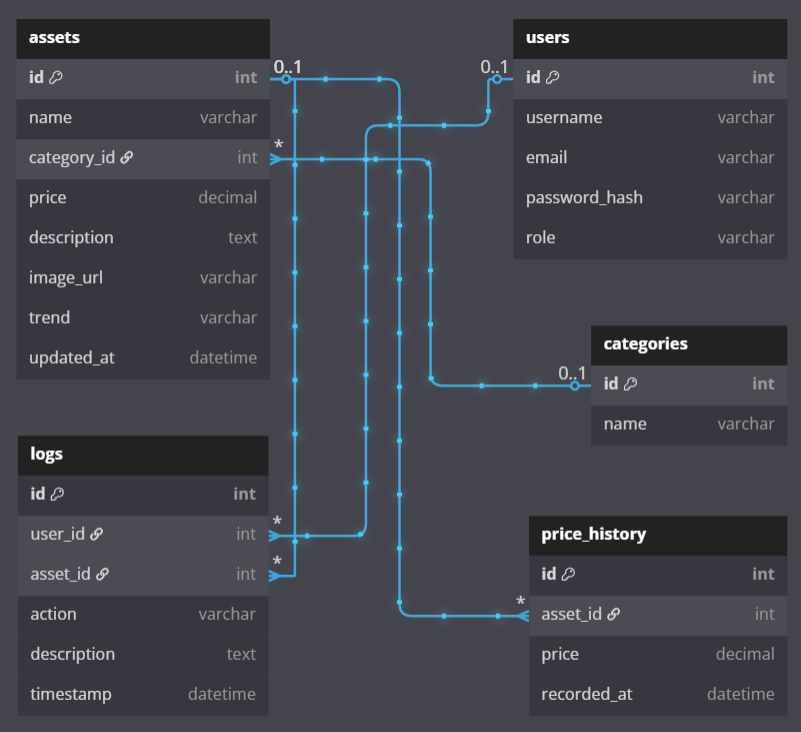
|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| id | int (PK) | Unique history entry |
| asset\_id | int (FK) | References assets.id |
| price | decimal | Price value recorded |
| recorded\_at | datetime | Timestamp of the price snapshot |

**logs**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| id | int (PK) | Unique log ID |
| user\_id | int (FK) | References users.id |
| asset\_id | int (FK) | References assets.id |
| action | varchar | Type of action (create, update, delete) |
| description | text | Details of the change |
| timestamp | datetime | When the action occurred |

**users**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| id | int (PK) | User ID |
| username | varchar | User's login name |
| email | varchar | User email |
| password\_hash | varchar | Encrypted password |
| role | varchar | 'admin' or 'user' |



**9. Class diagram**

<https://www.mermaidchart.com/raw/fb239ea8-ff8e-43cf-a4b1-c5cef8d40106?theme=light&version=v0.1&format=svg>

**10. Security Architecture (if applicable)**

* Role-based access (admin vs user)
* Authentication (JWT, sessions, etc.)
* Asset protection and validation

**11. Non-Functional Requirements**

* **Performance**: Sub-second load time for dashboard
* **Responsiveness**: Bootstrap UI for mobile/tablet/desktop
* **Scalability**: Microservices-compatible if needed
* **Maintainability**: Modular codebase, clean separation of concerns

**12. Future Extensions**

* User Authentication & Registration
* Asset Image Upload Support
* Notification System
* Integration with Real-Time Market APIs
* Progressive Web App (PWA) or Mobile App