ATM Monitoring System

Technical Design Document

Table of Contents

1. System Overview

The ATM Monitoring System is a Spring Boot-based web application that provides an API to monitor the status and behaviour of ATMs within a bank's network in real-time. The API exposes functionalities such as:

- Authorization and authentication of API requests.
- Monitoring customer transactions over the past 24 hours.
- Providing a breakdown of transactions by type: deposits, withdrawals, and balance inquiries.
- Logging and retrieving system and device failure incidents.
- Downloading camera images and videos from ATM devices based on a time range.
- Additional features to extend the API functionality.

The application uses an in-memory H2 database for data storage during development and testing. For real-world deployments, a more robust relational database (e.g., PostgreSQL or MySQL) would be suitable.

2. Prerequisites

Before setting up and running the ATM Monitoring System application, ensure you have the following prerequisites:

1. Development Environment

- **Java Development Kit (JDK)**: Version 17 or higher.
  - Verify installation by running:
    java -version

- Spring Tool Suite (STS) or IntelliJ IDEA: Any Java IDE that supports Spring Boot development.
- **Maven**: Version 2.7.4 for dependency management (if not included in your IDE).
- **Postman or Curl**: For testing API endpoints.

2. Dependencies

- **Spring Boot**: Version 2.7+ (specified in `pom.xml`).
- **H2 Database**: In-memory database used for development and testing. (Configured in `application.properties`).
- **Spring Security**: For securing API endpoints with token-based authentication.
- **Spring Data JPA**: To handle data persistence and interaction with the H2 database.

4. Running the Application

- **Build the project using Maven:**
  mvn clean install
- **Start the Spring Boot application:**
  mvn spring-boot:run
- **Access the H2 Console (for testing and debugging database data):**
  - **URL**: http://localhost:8080/h2-console
  - **JDBC URL**: jdbc:h2:mem:testdb
  - **Username**: sa
  - **Password**: password

3. System Design

2.1 Architecture Overview

The system follows a **layered architecture**, consisting of the following layers:

1. **Controller Layer**: Handles incoming HTTP requests and returns responses.
2. **Service Layer**: Contains the business logic.
3. **Repository Layer**: Manages data persistence and retrieval from the database.
4. **Data Access Layer (Entities)**: Represents the data models mapped to database tables.

The system leverages Spring Boot features like Spring Data JPA for ORM (Object-Relational Mapping) and Spring Security for authentication and authorization.

2.2 Authentication and Authorization

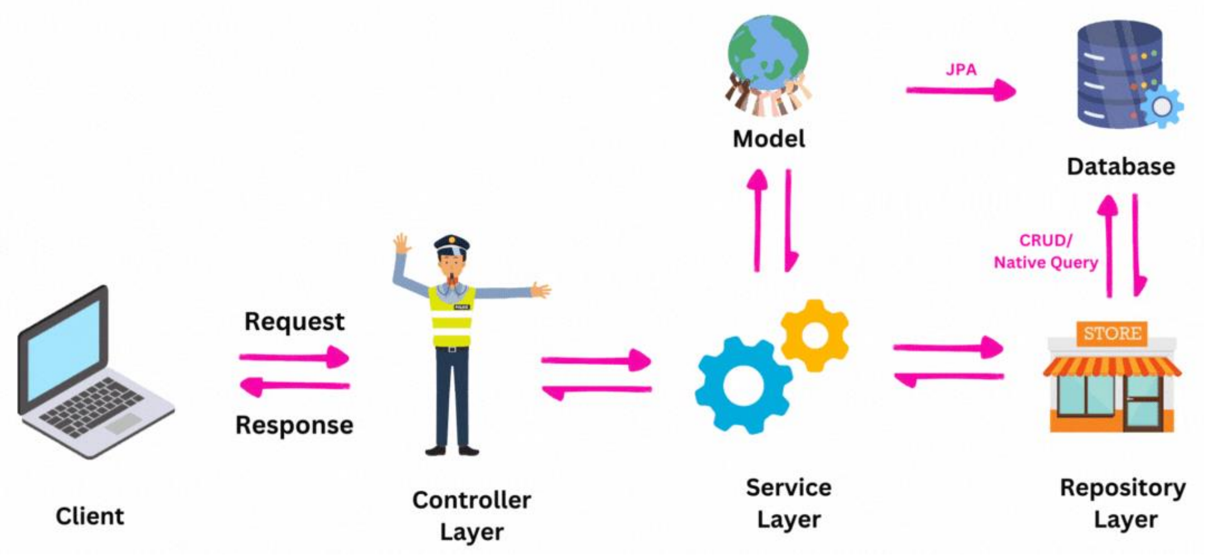
The system uses token-based authentication (e.g., JWT or OAuth 2.0) for securing API endpoints. Tokens are validated for each request to ensure that only authorized applications can access the resources.

2.3 Error Handling

The system uses global exception handling to provide meaningful error messages for client applications, including HTTP status codes and error details.

4. Application Flow Diagram

Below diagram application flow:



5. Component Design

3.1 Controller Layer

- `TransactionController`: Handles requests related to transactions.
- `FailureLogController`: Manages failure logs.
- `MediaController`: Provides access to video/image download functionalities.
- `ATMStatusController`: Checks the ATM operational status.

3.2 Service Layer

- `TransactionService`: Contains business logic for transaction operations.
- `FailureLogService`: Manages the logic for system/device failures.
- `MediaService`: Handles the retrieval of media files.
- `ATMStatusService`: Provides the operational status of ATMs.

3.3 Repository Layer

- `TransactionRepository`: Interface for CRUD operations on transactions.
- `FailureLogRepository`: Interface for managing failure logs.
- `MediaFileRepository`: Interface for storing and retrieving media files.

3.4 Security Layer

- Configured to validate authentication tokens and authorize users based on roles.

6. API Design

The API follows the **OpenAPI 3.1 specification**. Below are the main endpoints:

4.1 Authorization

- **POST** `/auth/login`: Authenticates a user and returns a token.
- **GET** `/auth/validate`: Validates a token.

4.2 Transaction Endpoints

- **GET** `/transactions/last24hours`: Returns the total number of customers and a breakdown of transactions in the last 24 hours.
- **GET** `/transactions/history?start={startTime}&end={endTime}`: Retrieves transaction history within the specified time range.

4.3 Failure Log Endpoints

- **GET** `/failures`: Returns all failure logs.
- **POST** `/failures`: Adds a new failure log.

4.4 Media Endpoints

- **GET** `/media/download?start={startTime}&end={endTime}`: Downloads media files (images or videos) within the specified time range.

4.5 Additional Endpoints

- **GET** `/atm/status`: Returns the operational status of the ATM.

7. Activity Flow Diagrams

5.1 Authentication Flow

1. Client sends login request with credentials.
2. System validates credentials and issues a token.
3. Token is used to access secured endpoints.

5.2 Transaction Monitoring Flow

1. Request for the last 24-hour transactions is made.
2. The system queries the database for transactions within the time range.
3. Response is sent with the transaction breakdown.

5.3 Failure Logging Flow

1. Failure logs can be retrieved or added via the respective endpoints.
2. When a new failure log is added, it is saved to the database.

8. Data Model Design

The data model consists of three primary entities:

6.1 `Transaction` Entity

```
@Entity
public class Transaction {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String type; // e.g., DEPOSIT, WITHDRAWAL, BALANCE_INQUIRY
   private Double amount;
   private LocalDateTime timestamp;
}
```

6.2 `FailureLog` Entity

```
@Entity
public class FailureLog {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String errorType; // e.g., SYSTEM, DEVICE
   private String description;
   private LocalDateTime timestamp;
}
```

6.3 `MediaFile` Entity

```
@Entity
public class MediaFile {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String filePath;
   private LocalDateTime startTime;
   private LocalDateTime endTime;
}
```

9. Data Model Examples

7.1 Example `Transaction` JSON

```
{
  "id": 1,
  "type": "DEPOSIT",
  "amount": 100.0,
  "timestamp": "2024-10-28T14:30:00"
}
```

7.2 Example `FailureLog` JSON

```
{
  "id": 1,
  "errorType": "SYSTEM",
  "description": "System failure occurred during cash withdrawal",
  "timestamp": "2024-10-28T15:00:00"
}
```

7.3 Example `MediaFile` JSON

```
{
  "id": 1,
  "filePath": "/path/to/media1.mp4",
  "startTime": "2024-10-27T10:00:00",
  "endTime": "2024-10-27T11:00:00"
}
```

10. Development Tasks Breakdown

1. **Setup Spring Boot Project**: Configure dependencies (Spring Data JPA, Spring Security, H2).
2. **Define Data Models and Repositories**: Create entities and JPA repository interfaces.
3. **Implement Service Layer**: Add business logic for transactions, failure logs, and media files.
4. **Create Controller Layer**: Expose REST endpoints.
5. **Configure Security**: Implement token-based authentication.
6. **Write Unit Tests**: Cover services and controllers with JUnit tests.
7. **Documentation and OpenAPI Specification**: Provide API documentation.
8. **Testing and Deployment:** Test with H2 database and prepare for production deployment.