TUS - Technological University of the Shannon

Athlone Campus

Student: Weverton de Souza Castanho
Email: wevertonsc@gmail.com
Course: Object-Oriented Programming I (AL_KCNCM_9_1) 29468
Date: November 2025

Card Authorizer Application

Credit card transaction authorization system built with Spring Boot, implementing Luhn algorithm validation, brand identification, and balance management.

Table of Contents

Overview

The Card Authorizer is an enterprise-grade Spring Boot application designed to process and authorize credit card transactions. The system validates card numbers using the Luhn algorithm, identifies card brands (Visa, Mastercard, etc.), and manages transaction authorization based on available balance and card validation rules.

This application was developed as part of the Object-Oriented Programming I course at Technology University Shannon - Athlone (TUS).

Features

Core Functionality

- **Card Validation**: Luhn algorithm implementation for credit card number validation
- **Brand Identification**: Automatic detection of card brands (Visa, Mastercard, American Express, Discover)
- **Transaction Authorization**: Real-time authorization based on card validation and balance availability
- **Balance Management**: Automatic balance updates after successful transactions
- **Transaction History**: Complete audit trail of all operations
- **Cardholder Verification**: Validates cardholder name, email, expiration date, and CVV

Technical Features

- RESTful API architecture
- H2 in-memory database for development
- JPA/Hibernate for data persistence
- Comprehensive unit and integration testing (127 tests)
- Spring Boot auto-configuration
- Lombok for reduced boilerplate code

Technology Stack

Backend

- **Java**: 17+
- **Spring Boot**: 3.x
- **Spring Data JPA**: Database abstraction layer
- **Spring Web**: REST API implementation

Database

- **H2 Database**: In-memory database for development
- **Hibernate**: ORM framework

Build & Dependencies

- **Maven**: Dependency management and build automation
- **Lombok**: Code generation for DTOs and entities

Testing

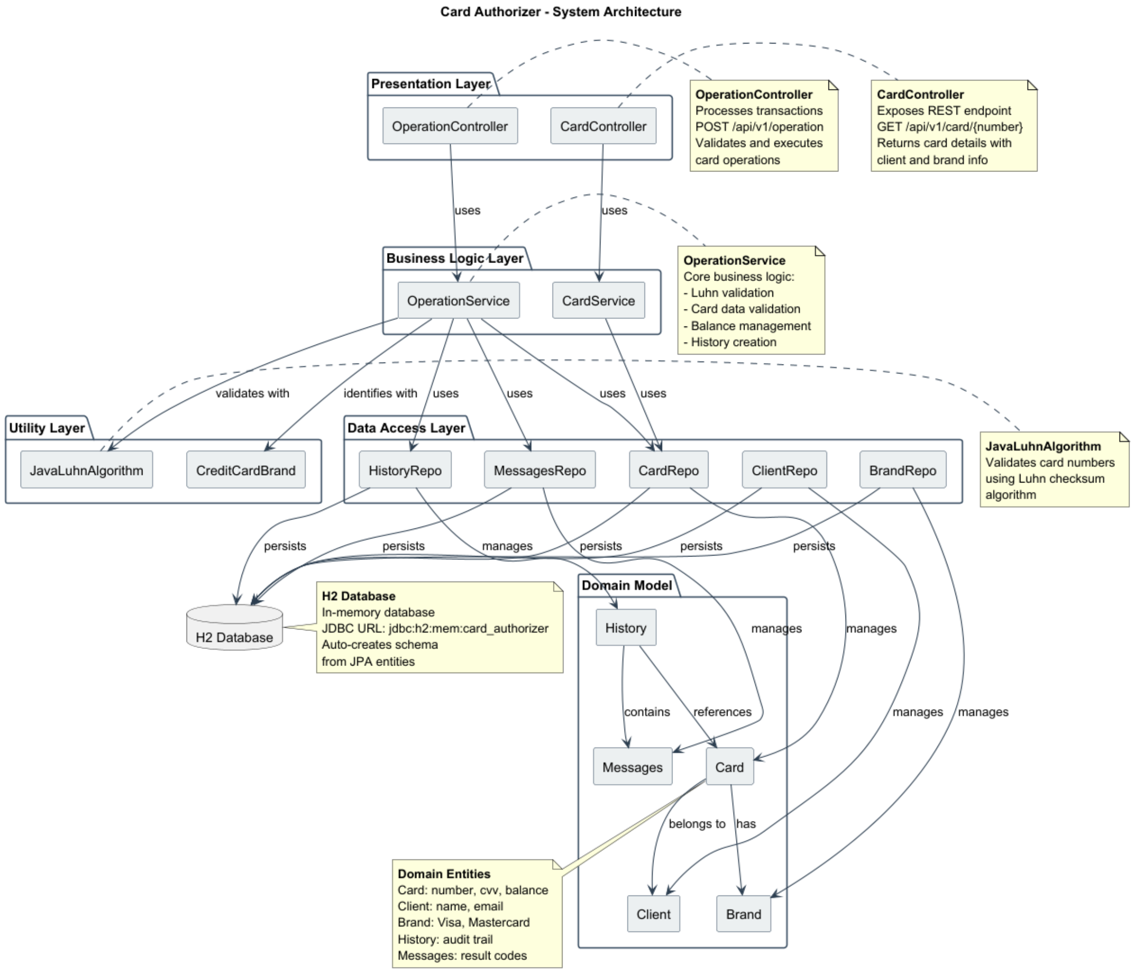
- **JUnit 5**: Unit testing framework

- **Mockito**: Mocking framework
- **Spring Test**: Integration testing support
- **MockMvc**: REST API testing

Architecture

Design Pattern

The application follows a layered architecture pattern:



Card Authorizer - System Architecture

OperationController
Processes transactions
POST /api/v1/operation
Validates and executes
card operations

CardController
Exposes REST endpoint
GET /api/v1/card/{number}
Returns card details with
client and brand info

OperationService
Core business logic:
- Luhn validation
- Card data validation
- Balance management
- History creation

JavaLuhnAlgorithm
Validates card numbers
using Luhn checksum
algorithm

H2 Database
In-memory database
JDBC URL: jdbc:h2:mem:card_authorizer
Auto-creates schema
from JPA entities

Domain Entities
Card: number, cvv, balance
Client: name, email
Brand: Visa, Mastercard
History: audit trail
Messages: result codes

Core Components

Models

- **Card**: Credit card entity with number, expiration, CVV, limits, balance
- **Client**: Cardholder information
- **Brand**: Card brand details (Visa, Mastercard, etc.)
- **History**: Transaction audit trail
- **Messages**: Operation result messages

Services

- **CardService**: Card retrieval and management
- **OperationService**: Transaction processing and validation

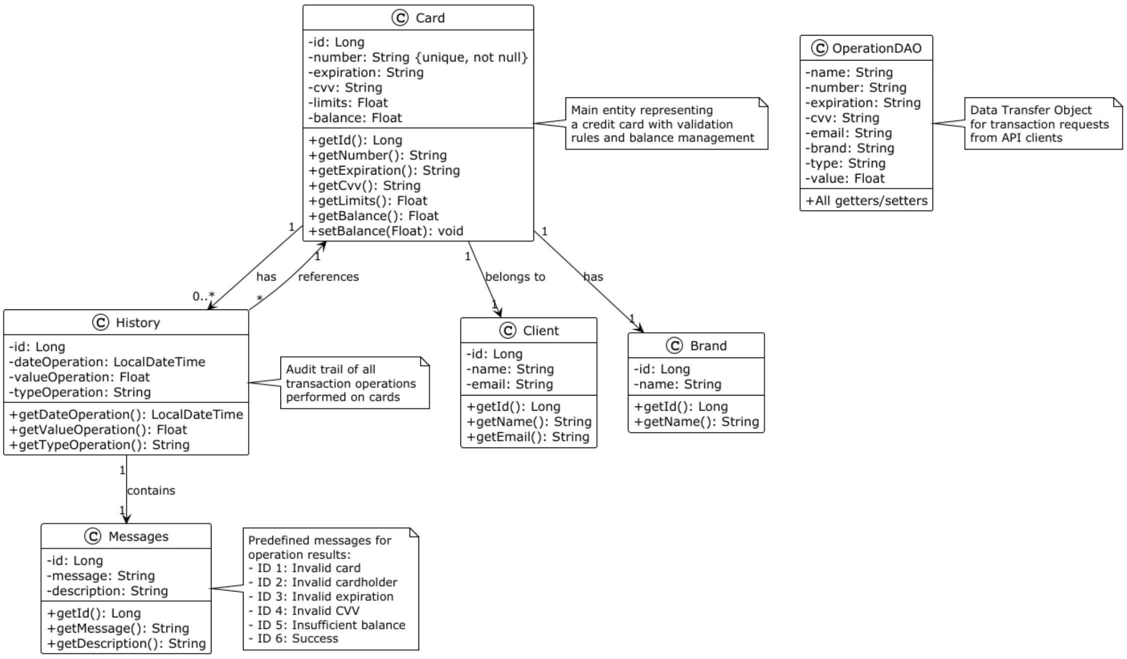
Controllers

- **CardController**: Card information endpoints
- **OperationController**: Transaction processing endpoints

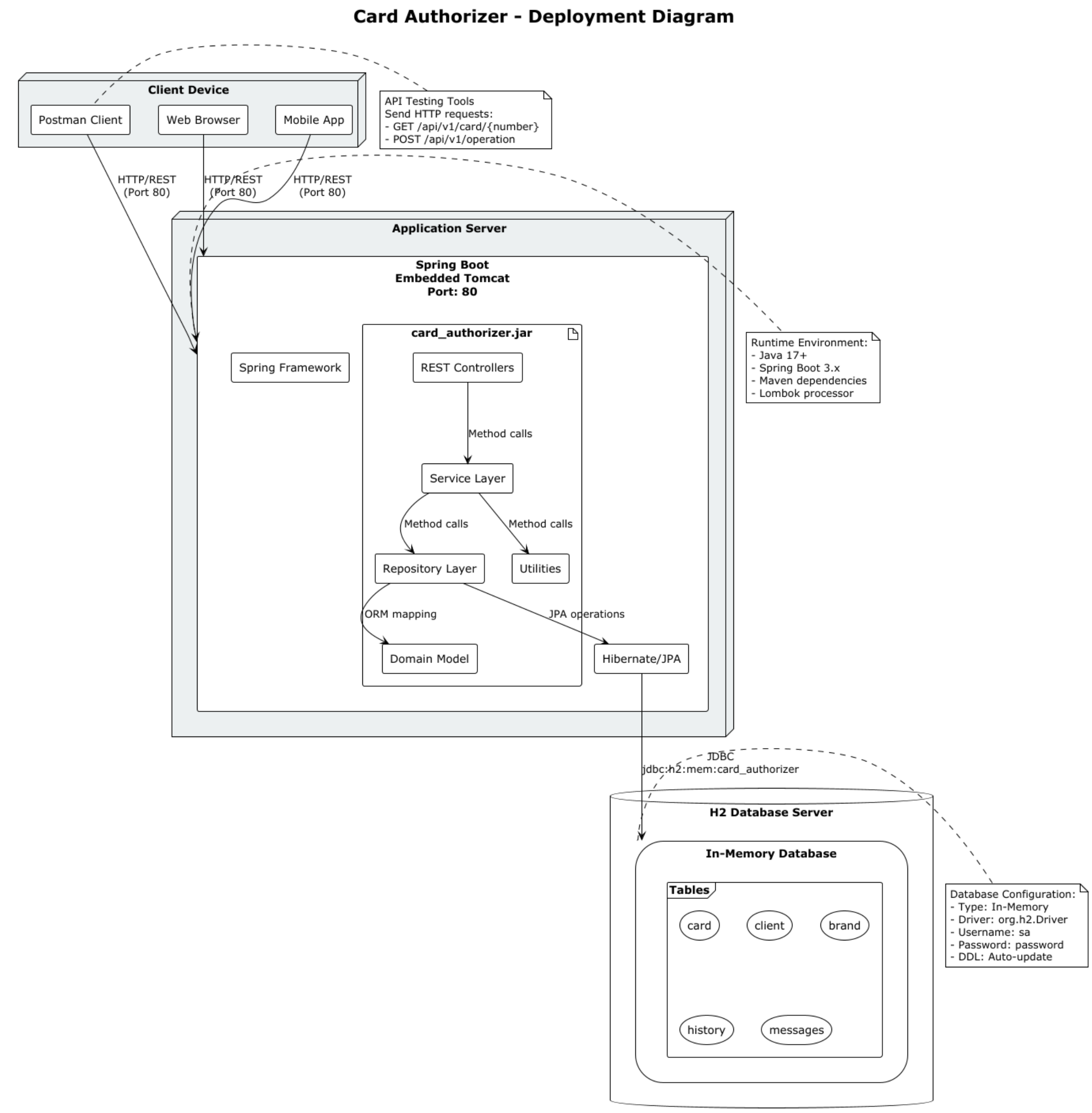
Utilities

- **JavaLuhnAlgorithm**: Credit card number validation
- **CreditCardBrand**: Brand identification logic

Card Authorizer - Domain Model Class Diagram



Getting Started



Prerequisites

- Java Development Kit (JDK) 17 or higher
- Maven 3.6 or higher
- IDE (IntelliJ IDEA, Eclipse, or VS Code recommended)
- Postman (for API testing)

Installation

1. Clone the repository:

```
git clone <repository-url>
cd card_authorizer
```

2. Build the project:

```
mvn clean install
```

3. Run the application:

```
mvn spring-boot:run
```

The application will start on port 80 by default.

Configuration

The application uses the following default configuration in application.properties:

```
spring.application.name=card_authorizer

# H2 Database Configuration
spring.datasource.url=jdbc:h2:mem:card_authorizer
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.defer-datasource-initialization=true
logging.level.org.hibernate.SQL= DEBUG

# Server Port Configuration
server.port=8088
```

To change the server port, modify server.port in application.properties.

API Documentation

Base URL

http://localhost:8088/api/v1

Endpoints

1. Get Card by Number

Retrieves card information by card number.

Endpoint: GET /card/{number}

Path Parameters:

- number (string, required): Credit card number

Response: Card object with client and brand relationships

Example Request:

```
GET http://localhost:80/api/v1/card/4532015112830366
```

Example Response (200 OK):

```
{
  "id": 1,
  "number": "4532015112830366",
  "expiration": "12/25",
  "cvv": "123",
  "limits": 5000.0,
  "balance": 3000.0,
  "client": {
    "id": 1,
    "name": "John Doe",
    "email": "john.doe@example.com"
  },
  "brand": {
    "id": 1,
    "name": "Visa"
  }
}
```

Error Response (404 Not Found):

```
Card not found with number: 9999999999999999
```

---

2. Process Transaction

Processes a credit card transaction (purchase, withdrawal, etc.).

Endpoint: POST /operation

Request Body: OperationDAO object

Fields:

- name (string, required): Cardholder name
- number (string, required): Credit card number
- expiration (string, required): Card expiration date (MM/YY)
- cvv (string, required): Card CVV code
- email (string, required): Cardholder email
- brand (string, required): Card brand
- type (string, required): Operation type (e.g., "PURCHASE")
- value (float, required): Transaction amount

Response: Messages object with operation result

Example Request:

```
⏩ POST http://localhost:80/api/v1/operation
Content-Type: application/json

{
  "name": "Plato of Athens",
  "number": "4532015112830366",
  "expiration": "12/25",
  "cvv": "123",
  "email": "plato.of.athens@macunaima.com",
  "brand": "Visa",
  "type": "PURCHASE",
  "value": 100.00
}
```

Success Response (200 OK):

```
{
  "id": 6,
  "message": "SUCCESS",
  "description": "Operation completed successfully"
}
```

Error Response (200 OK with error message):

```
{
  "id": 1,
  "message": "ERROR",
  "description": "Invalid card number"
}
```

Error Response (500 Internal Server Error):

```
Error processing operation: <error details>
```

Validation Rules

The system applies the following validation rules for transactions:

1. **Luhn Algorithm**: Card number must pass Luhn algorithm validation
2. **Card Existence**: Card must exist in database
3. **Cardholder Name**: Must match the name on record
4. **Email**: Must match the email on record
5. **Expiration Date**: Must match the expiration date on record
6. **CVV**: Must match the CVV on record
7. **Brand**: Must match the card brand

8. **Balance**: Transaction amount must not exceed available balance

Error Message IDs

The system uses the following message IDs for operation results:

- **ID 1**: Invalid card number (Luhn algorithm failed or card not found)
- **ID 2**: Invalid cardholder information (name or email mismatch)
- **ID 3**: Invalid expiration date
- **ID 4**: Invalid CVV
- **ID 5**: Insufficient balance
- **ID 6**: Success

Testing

Running Tests

Execute all tests:

```
mvn test
```

Execute specific test class:

```
mvn test -Dtest=CardServiceTest
```

Execute integration tests only:

```
mvn test -Dtest=*IntegrationTest
```

Test Coverage

The application includes comprehensive test coverage with 127 tests:

Unit Tests

- **CardServiceTest**: Card service business logic (13 tests)
- **OperationServiceTest**: Operation service validation (22 tests)

- **ModelEntityTest**: Entity model validation (54 tests)
- **CreditCardBrandTest**: Brand identification logic (9 tests)
- **JavaLuhnAlgorithmTest**: Luhn algorithm implementation (17 tests)
- **OperationControllerTest**: Controller layer testing (2 tests)

Integration Tests

- **CardControllerIntegrationTest**: End-to-end API testing (10 tests)

Postman Testing

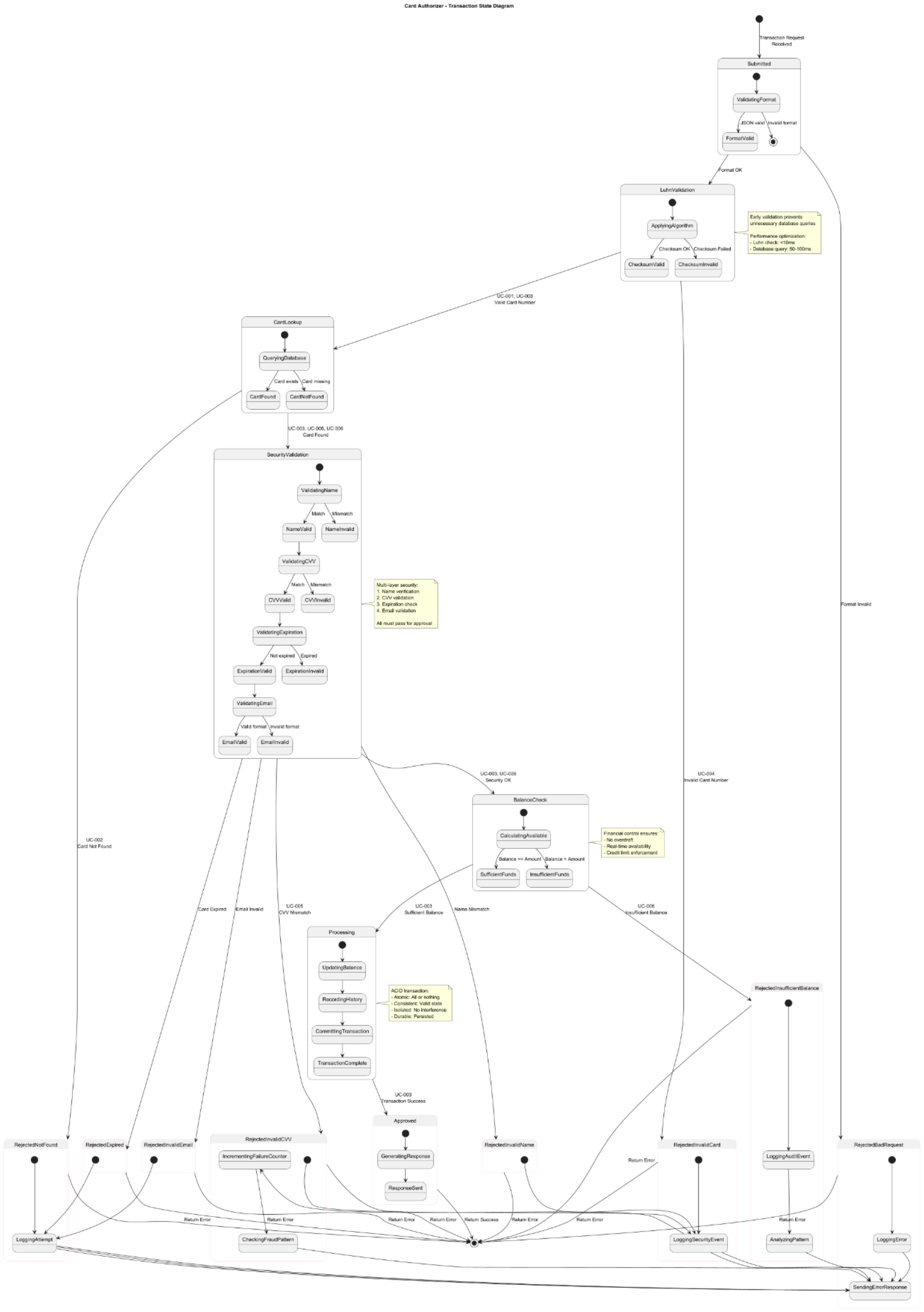
Import the provided Postman collection: Card_Authorizer_Postman_Collection.json

The collection includes:

1. **Card Operations** (3 tests)
   - Get valid Visa card
   - Get valid Mastercard
   - Card not found scenario
2. **Transaction Operations** (4 tests)
   - Valid purchase transactions
   - Small and large amount transactions
   - Multi-brand support
3. **Validation Tests** (8 tests)
   - Invalid Luhn algorithm
   - Invalid cardholder information
   - Invalid card details (CVV, expiration)
   - Insufficient balance
   - Card number format validation
4. **Edge Cases** (6 tests)
   - Exact balance transactions
   - Zero and negative amounts
   - Missing required fields
   - Very long card numbers
5. **Performance Tests** (2 tests)
   - Sequential transaction processing
   - Card retrieval performance

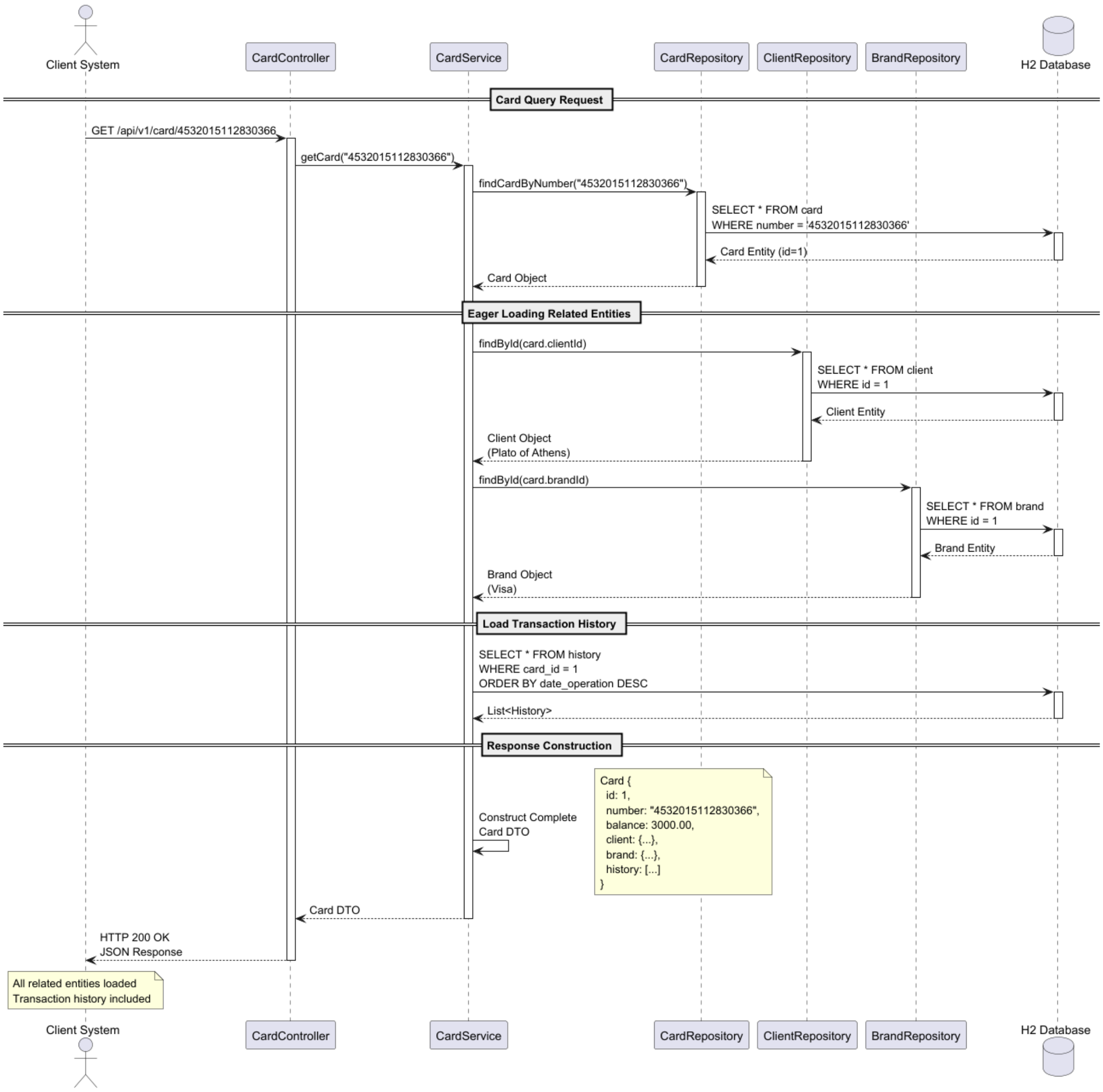
Using Postman Collection

1. Import the collection file into Postman
2. Set the environment variable:
   - base_url: http://localhost:80
3. Run individual tests or the entire collection
4. Review test results in the Postman test runner

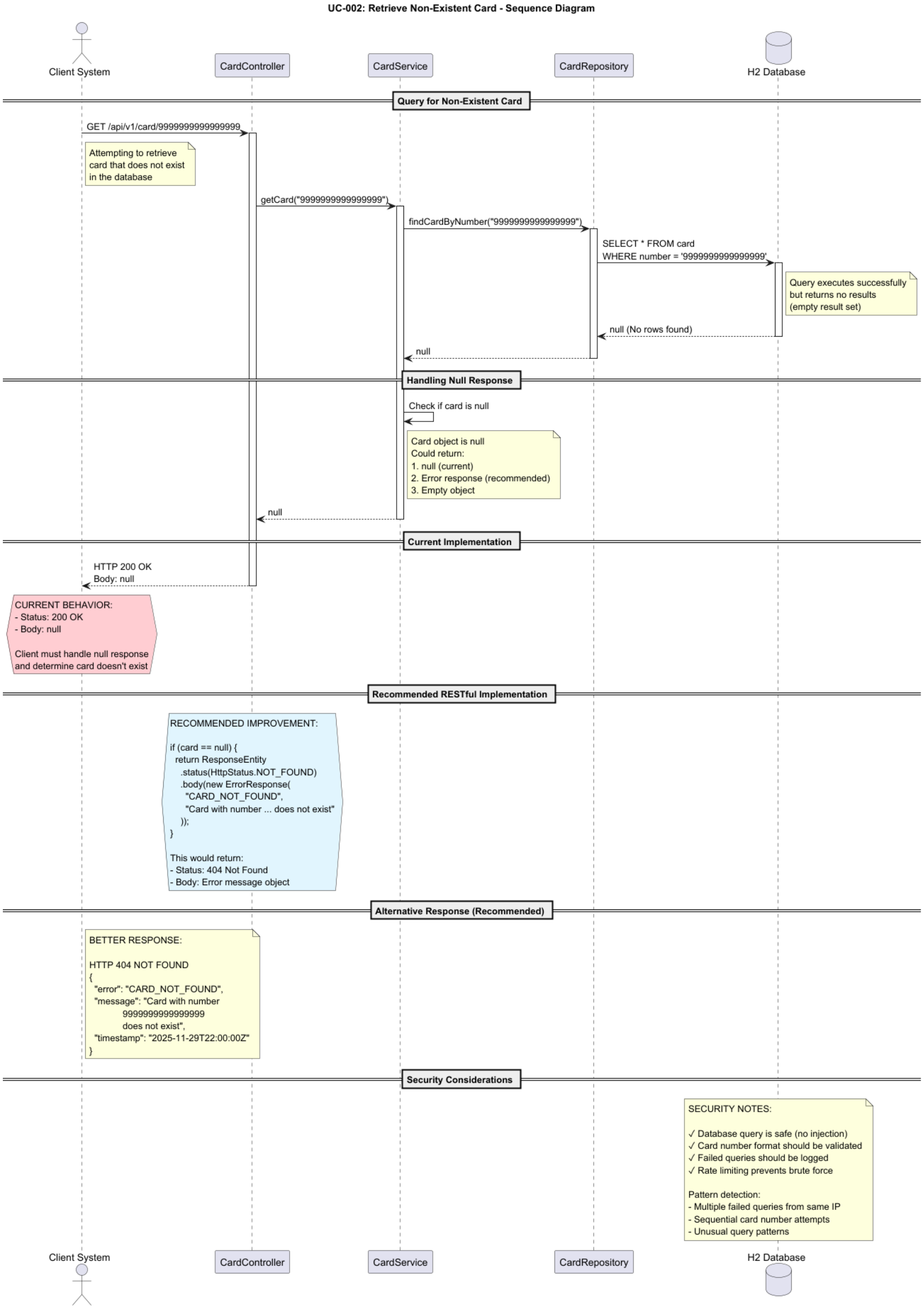

Card Authorizer - Transaction State Diagram
Transaction Request Received
Submitted
ValidatingFormat
JSON valid
Invalid format
FormatValid
Format OK
Format Invalid
LuhnValidation
ApplyingAlgorithm
Checksum OK
Checksum Failed
ChecksumValid
ChecksumInvalid
Early validation prevents unnecessary database queries
Performance optimization:
- Luhn check: <10ms
- Database query: 50-100ms
UC-001, UC-003
Valid Card Number
CardLookup
QueryingDatabase
Card exists
Card missing
CardFound
CardNotFound
UC-003, UC-005, UC-006
Card Found
UC-004
Invalid Card Number
SecurityValidation
ValidatingName
Match
Mismatch
NameValid
NameInvalid
ValidatingCVV
Match
Mismatch
CVVValid
CVVInvalid
ValidatingExpiration
Not expired
Expired
ExpirationValid
ExpirationInvalid
ValidatingEmail
Valid format
Invalid format
EmailValid
EmailInvalid
Multi-layer security:
1. Name verification
2. CVV validation
3. Expiration check
4. Email validation
All must pass for approval
UC-002
Card Not Found
UC-003, UC-006
Security OK
BalanceCheck
CalculatingAvailable
Balance >= Amount
Balance < Amount
SufficientFunds
InsufficientFunds
Financial control ensures:
- No overdraft
- Real-time availability
- Credit limit enforcement
Card Expired
Email Invalid
UC-005
CVV Mismatch
UC-003
Sufficient Balance
Name Mismatch
UC-006
Insufficient Balance
Processing
UpdatingBalance
RecordingHistory
CommittingTransaction
TransactionComplete
ACID transaction:
- Atomic: All or nothing
- Consistent: Valid state
- Isolated: No interference
- Durable: Persisted
RejectedInsufficientBalance
UC-003
Transaction Success
Approved
GeneratingResponse
ResponseSent
RejectedNotFound
RejectedExpired
RejectedInvalidEmail
RejectedInvalidCVV
IncrementingFailureCounter
RejectedInvalidName
RejectedInvalidCard
LoggingAuditEvent
RejectedBadRequest
Return Error
Return Error
Return Error
Return Error
Return Error
Return Success
Return Error
Return Error
Return Error
Return Error
LoggingAttempt
CheckingFraudPattern
LoggingSecurityEvent
AnalyzingPattern
LoggingError
SendingErrorResponse

USE CASE 01

UC-001: Retrieve Existing Card - Sequence Diagram



USE CASE 02



UC-002: Retrieve Non-Existent Card - Sequence Diagram

Client System
CardController
CardService
CardRepository
H2 Database
Query for Non-Existent Card
GET /api/v1/card/9999999999999999
getCard("9999999999999999")
findCardByNumber("9999999999999999")
SELECT * FROM card
WHERE number = '9999999999999999'
null (No rows found)
null
Handling Null Response
Check if card is null
null
Current Implementation
HTTP 200 OK
Body: null
Recommended RESTful Implementation
Alternative Response (Recommended)
Security Considerations

Attempting to retrieve card that does not exist in the database

Query executes successfully but returns no results (empty result set)

Card object is null
Could return:
1. null (current)
2. Error response (recommended)
3. Empty object

CURRENT BEHAVIOR:
- Status: 200 OK
- Body: null

Client must handle null response and determine card doesn't exist

RECOMMENDED IMPROVEMENT:

```
if (card == null) {
  return ResponseEntity
    .status(HttpStatus.NOT_FOUND)
    .body(new ErrorResponse(
      "CARD_NOT_FOUND",
      "Card with number ... does not exist"
    ));
}
```

This would return:
- Status: 404 Not Found
- Body: Error message object

BETTER RESPONSE:

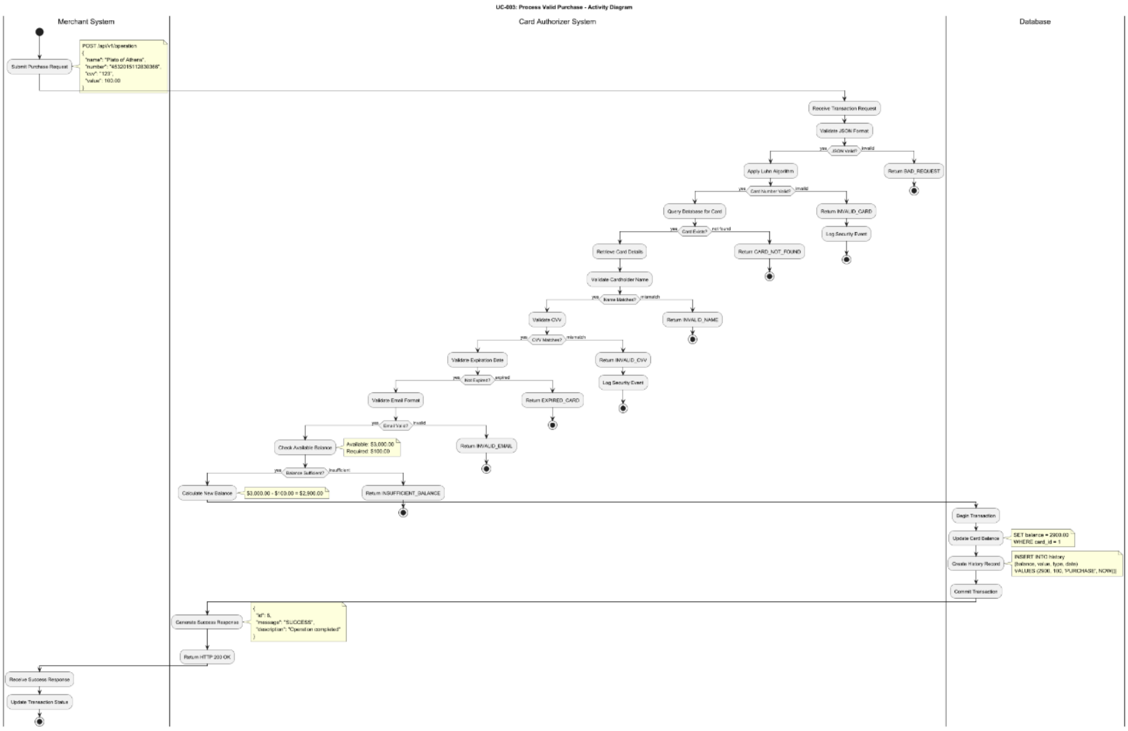
```
HTTP 404 NOT FOUND
{
  "error": "CARD_NOT_FOUND",
  "message": "Card with number
            9999999999999999
            does not exist",
  "timestamp": "2025-11-29T22:00:00Z"
}
```

SECURITY NOTES:

✓ Database query is safe (no injection)
✓ Card number format should be validated
✓ Failed queries should be logged
✓ Rate limiting prevents brute force

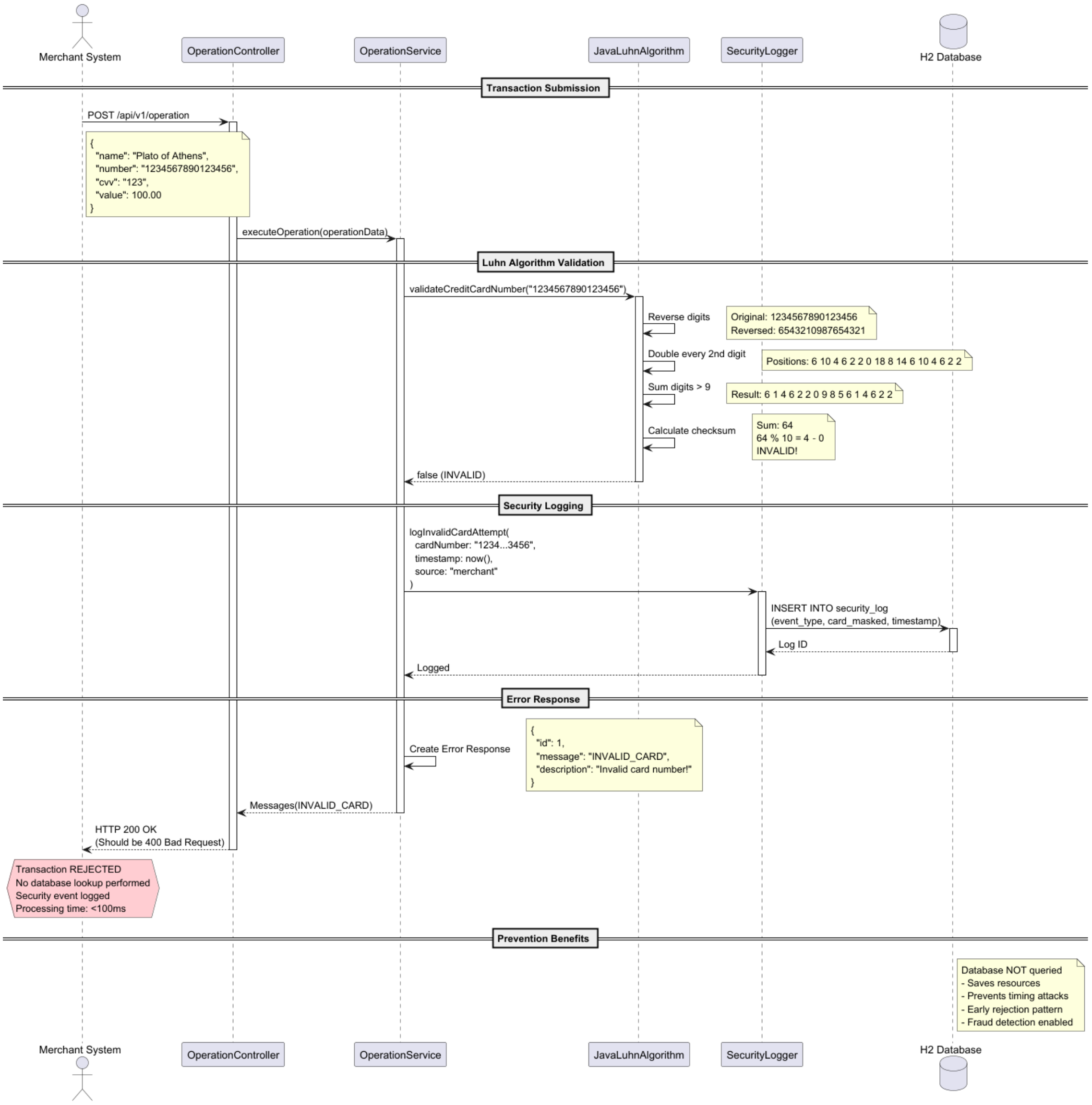
Pattern detection:
- Multiple failed queries from same IP
- Sequential card number attempts
- Unusual query patterns

USE CASE 03

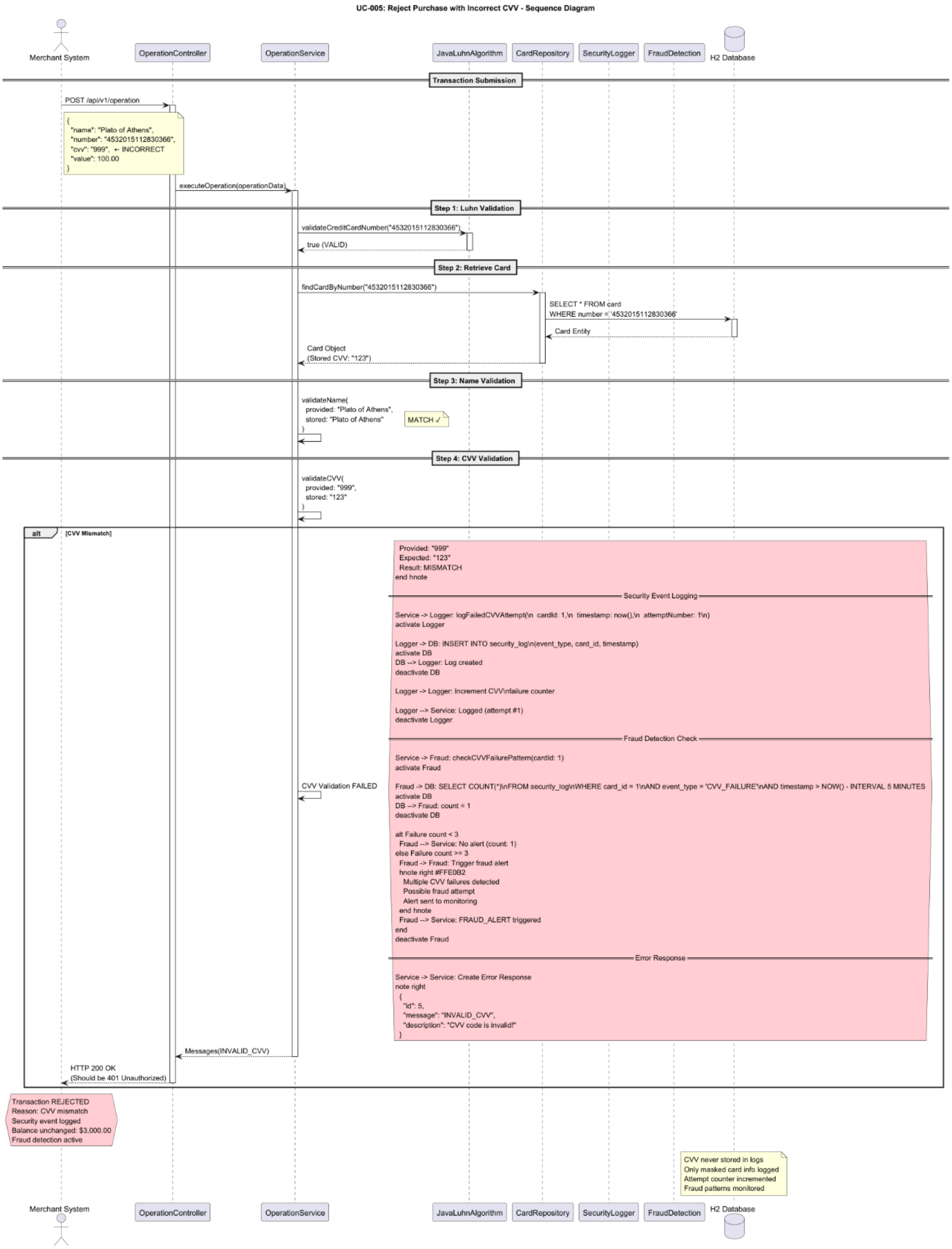
UC-003: Process Valid Purchase - Activity Diagram
Merchant System
Card Authorizer System
Database
Submit Purchase Request
POST /api/v1/operation
{
"name": "Plato of Athens",
"number": "4532015112830366",
"cvv": "123",
"value": 100.00
}
Receive Transaction Request
Validate JSON Format
JSON Valid?
yes
invalid
Apply Luhn Algorithm
Return BAD_REQUEST
Card Number Valid?
yes
invalid
Query Database for Card
Return INVALID_CARD
Log Security Event
Card Exists?
yes
not found
Retrieve Card Details
Return CARD_NOT_FOUND
Validate Cardholder Name
Name Matches?
yes
mismatch
Validate CVV
Return INVALID_NAME
CVV Matches?
yes
mismatch
Validate Expiration Date
Return INVALID_CVV
Log Security Event
Not Expired?
yes
expired
Validate Email Format
Return EXPIRED_CARD
Email Valid?
yes
invalid
Check Available Balance
Available: $3,000.00
Required: $100.00
Return INVALID_EMAIL
Balance Sufficient?
yes
insufficient
Calculate New Balance
$3,000.00 - $100.00 = $2,900.00
Return INSUFFICIENT_BALANCE
Begin Transaction
Update Card Balance
SET balance = 2900.00
WHERE card_id = 1
Create History Record
INSERT INTO history
(balance, value, type, date)
VALUES (2900, 100, 'PURCHASE', NOW())
Commit Transaction
Generate Success Response
{
"id": 5,
"message": "SUCCESS",
"description": "Operation completed"
}
Return HTTP 200 OK
Receive Success Response
Update Transaction Status

USE CASE 04

UC-004: Reject Purchase with Invalid Card - Sequence Diagram



USE CASE 05

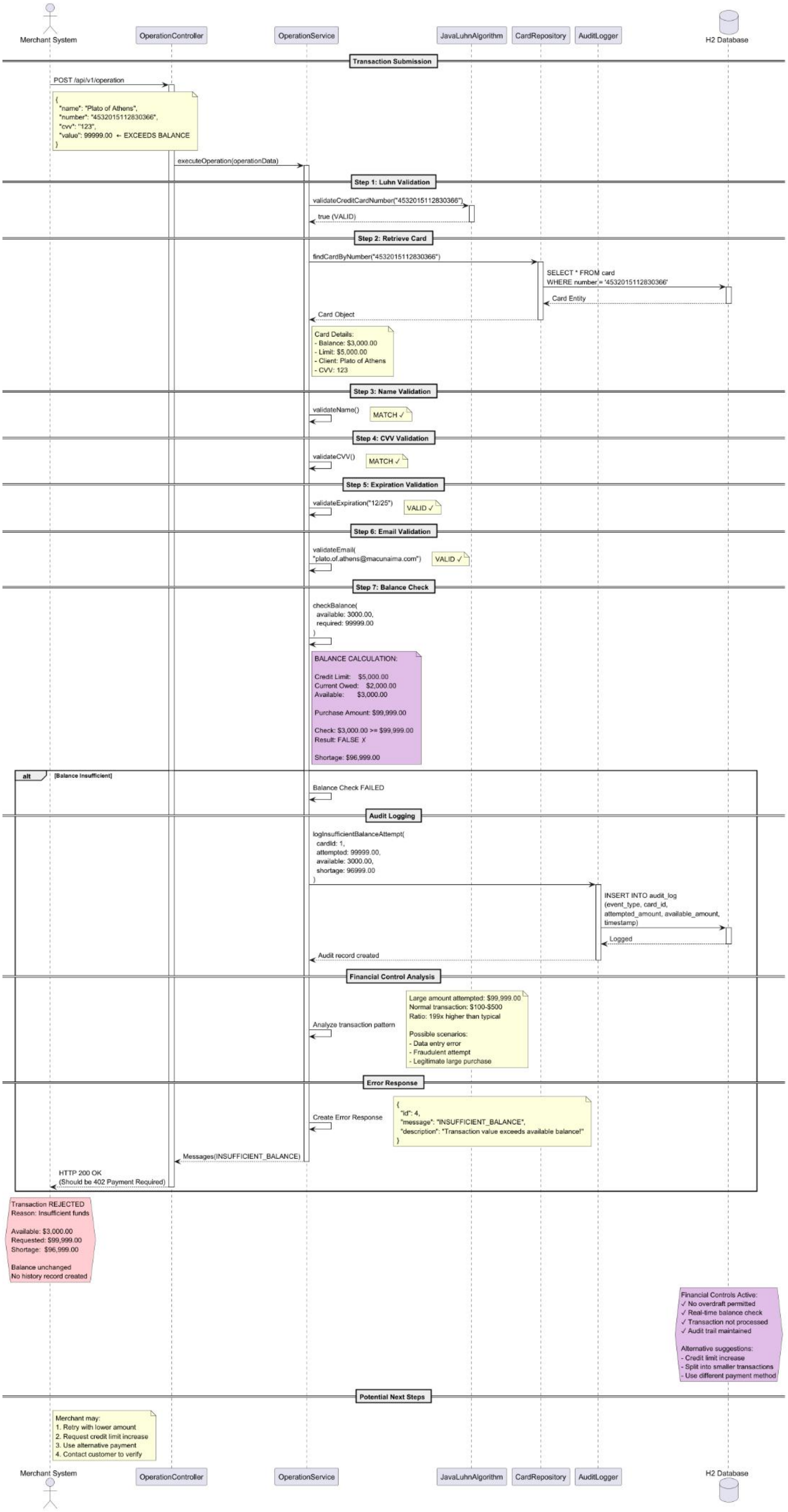


UC-005: Reject Purchase with Incorrect CVV - Sequence Diagram

```
 Provided: "999"
 Expected: "123"
 Result: MISMATCH
end hnote

Security Event Logging

Service -> Logger: logFailedCVVAttempt(\n  cardId: 1,\n  timestamp: now(),\n  attemptNumber: 1\n)
activate Logger

Logger -> DB: INSERT INTO security_log\n(event_type, card_id, timestamp)
activate DB
DB --> Logger: Log created
deactivate DB

Logger -> Logger: Increment CVV\nfailure counter

Logger --> Service: Logged (attempt #1)
deactivate Logger

Fraud Detection Check

Service -> Fraud: checkCVVFailurePattern(cardId: 1)
activate Fraud

Fraud -> DB: SELECT COUNT(*)\nFROM security_log\nWHERE card_id = 1\nAND event_type = 'CVV_FAILURE'\nAND timestamp > NOW() - INTERVAL 5 MINUTES
activate DB
DB --> Fraud: count = 1
deactivate DB

alt Failure count < 3
 Fraud --> Service: No alert (count: 1)
else Failure count >= 3
 Fraud -> Fraud: Trigger fraud alert
 hnote right #FFE0B2
  Multiple CVV failures detected
  Possible fraud attempt
  Alert sent to monitoring
 end hnote
 Fraud --> Service: FRAUD_ALERT triggered
end
deactivate Fraud

Error Response

Service -> Service: Create Error Response
note right
 {
  "id": 5,
  "message": "INVALID_CVV",
  "description": "CVV code is invalid!"
 }
```

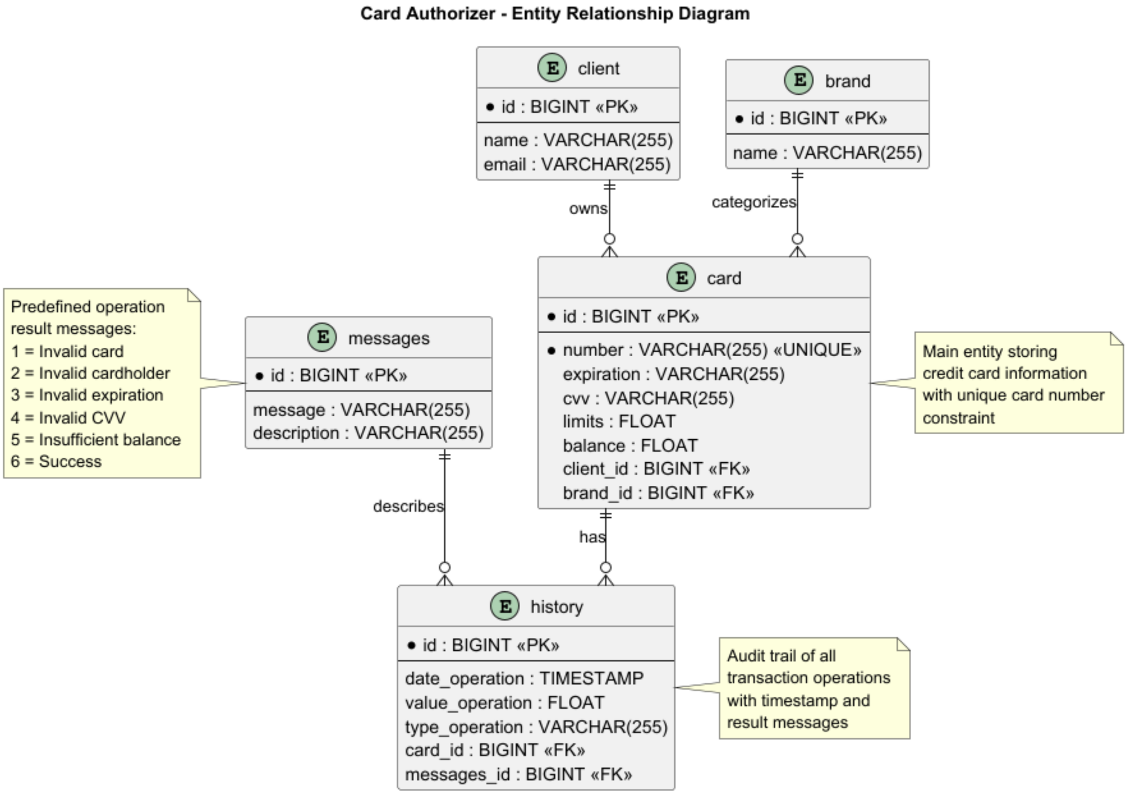
USE CASE 06

UC-006: Reject Purchase with Insufficient Balance - Sequence Diagram



Database Schema

Entity Relationship Diagram



Card Authorizer - Entity Relationship Diagram

Predefined operation result messages:
1 = Invalid card
2 = Invalid cardholder
3 = Invalid expiration
4 = Invalid CVV
5 = Insufficient balance
6 = Success

Main entity storing credit card information with unique card number constraint

Audit trail of all transaction operations with timestamp and result messages

Tables

card

- id (BIGINT, PK): Unique identifier
- number (VARCHAR, UNIQUE, NOT NULL): Card number
- expiration (VARCHAR): Expiration date
- cvv (VARCHAR): CVV code
- limits (FLOAT): Credit limit
- balance (FLOAT): Available balance
- client_id (BIGINT, FK): Client reference
- brand_id (BIGINT, FK): Brand reference

client

- id (BIGINT, PK): Unique identifier
- name (VARCHAR): Cardholder name
- email (VARCHAR): Cardholder email

brand

- id (BIGINT, PK): Unique identifier
- name (VARCHAR): Brand name (Visa, Mastercard, etc.)

history

- id (BIGINT, PK): Unique identifier
- card_id (BIGINT, FK): Card reference
- messages_id (BIGINT, FK): Message reference
- date_operation (TIMESTAMP): Transaction timestamp
- value_operation (FLOAT): Transaction amount
- type_operation (VARCHAR): Operation type

messages

- id (BIGINT, PK): Unique identifier
- message (VARCHAR): Message code (SUCCESS, ERROR)
- description (VARCHAR): Message description

Project Structure

```
card_authorizer/
├── src/
│   ├── main/
│   │   ├── java/com/tus/oop1/card_authorizer/
│   │   │   ├── CardAuthorizerApplication.java
│   │   │   ├── controller/
│   │   │   │   ├── CardController.java
│   │   │   │   └── OperationController.java
│   │   │   ├── dao/
│   │   │   │   └── OperationDAO.java
│   │   │   ├── model/
│   │   │   │   ├── Brand.java
│   │   │   │   ├── Card.java
│   │   │   │   ├── Client.java
│ │ │ │   ├── History.java
│ │ │ │   └── Messages.java
│ │ │   ├── repo/
│ │ │ │   ├── BrandRepo.java
│ │ │ │   ├── CardRepo.java
│ │ │ │   ├── ClientRepo.java
│ │ │ │   ├── HistoryRepo.java
│ │ │ │   └── MessagesRepo.java
│ │ │   ├── service/
│ │ │ │   ├── CardService.java
│ │ │ │   └── OperationService.java
│ │ │   └── util/
│ │ │       ├── CreditCardBrand.java
│ │ │       └── JavaLuhnAlgorithm.java
│ │   └── resources/
│ │       └── application.properties
│ └── test/
│     └── java/com/tus/oop1/card_authorizer/
│         ├── CardAuthorizerApplicationTests.java
│         ├── controller/
│         │   └── OperationControllerTest.java
│         ├── integration/
│         │   └── CardControllerIntegrationTest.java
│         ├── model/
│         │   └── ModelEntityTest.java
│         ├── service/
│         │   ├── CardServiceTest.java
│         │   └── OperationServiceTest.java
│         └── util/
│             ├── CreditCardBrandTest.java
│       └── JavaLuhnAlgorithmTest.java
├── uml
│   ├── images
│   ├── tests
│   │   ├── images
│   │   ├── uc001
│   │   ├── uc002
│   │   ├── uc003
│   │   ├── uc004
│   │   ├── uc005
│   │   └── uc005
│   ├── architecture.puml
│   ├── class.puml
│   ├── deployment.puml
│   ├── er_diagram.puml
│   ├── sequence.puml
│   ├── sequence_transaction.puml
│   ├── state.puml
│   └── use_case.puml
├── Card_Authorizer_Postman_Collection.json
├── Card_Authorizer_Postman_Collection.json
├── pom.xml
└── README.md
```

Development

Code Style

The project follows standard Java coding conventions:

- Package naming: lowercase (e.g., com.tus.oop1.card_authorizer)
- Class naming: PascalCase (e.g., CardController)
- Method naming: camelCase (e.g., executeOperation)

- Constant naming: UPPER_SNAKE_CASE

Lombok Annotations

The project uses Lombok to reduce boilerplate code:

- @Data: Generates getters, setters, toString, equals, and hashCode
- @Getter / @Setter: Generates getter/setter methods
- @NoArgsConstructor: Generates no-argument constructor
- @AllArgsConstructor: Generates all-arguments constructor
- @ToString: Generates toString method
- @EqualsAndHashCode: Generates equals and hashCode methods

Adding New Features

1. Create model entities in model/ package
2. Create repository interfaces in repo/ package
3. Implement business logic in service/ package
4. Create REST endpoints in controller/ package
5. Add comprehensive tests in test/ directory

Sample Test Data

Valid Cards

Visa Card

```
{
  "number": "4532015112830366",
  "expiration": "12/25",
  "cvv": "123",
  "name": "John Doe",
  "email": "john.doe@example.com",
  "brand": "Visa",
  "balance": 3000.00,
  "limits": 5000.00
}
```

Mastercard

```
{
  "number": "5425233430109903",
  "expiration": "11/24",
  "cvv": "456",
  "name": "John Doe",
  "email": "john.doe@example.com",
  "brand": "Mastercard",
  "balance": 2000.00,
  "limits": 3000.00
}
```

Other Valid Card Numbers (for testing Luhn algorithm)

- **Visa**: 4111111111111111
- **Mastercard**: 5555555555554444
- **American Express**: 378282246310005
- **Discover**: 6011111111111117

Common Issues and Solutions

Port Already in Use

If port 80 is already in use, change the port in application.properties:

```
server.port=8080
```

Database Connection Issues

If you encounter database connection issues, ensure H2 configuration is correct:

```
spring.datasource.url=jdbc:h2:mem:card_authorizer

spring.datasource.driverClassName=org.h2.Driver
```

Test Failures

If tests fail, ensure the application is not running on the same port:

```
# Stop the application before running tests
mvn test
```

Build Failures

Clean and rebuild the project:

```
mvn clean install -U
```

Performance Considerations

Database Optimization

- Use an in-memory H2 database for fast development
- Indexes on card number for quick lookups
- Lazy loading for relationships to reduce memory footprint

API Response Times

Expected response times:

- Card retrieval: < 500ms
- Transaction processing: < 2000ms

Scalability

For production deployment, consider:

- Migrating to PostgresSQL or MySQL
- Implementing caching (Redis)
- Adding connection pooling
- Horizontal scaling with load balancer

Security Considerations

Note: This is a demonstration application. For production use, implement:

1. **Authentication & Authorization**: Add Spring Security
2. **HTTPS**: Enable SSL/TLS encryption
3. **Input Validation**: Enhance validation rules

4. **Rate Limiting**: Prevent abuse
5. **Logging & Monitoring**: Add comprehensive logging
6. **Database Security**: Use encrypted connections
7. **CVV Handling**: Never log or store CVV permanently
8. **PCI DSS Compliance**: Follow payment card industry standards

---

License

This project was developed as part of academic coursework at Technology University Shannon - Athlone (TUS).

Student: Weverton de Souza Castanho
Email: wevertonsc@gmail.com
Course: Object-Oriented Programming I (AL_KCNCM_9_1) 29468
Date: November 2025

Contact

For questions or support, please contact:

Weverton de Souza Castanho

Email: wevertonsc@gmail.com / A00324822@student.tus.ie

Institution: Technology University Shannon - Athlone (TUS)

References

- Spring Boot framework and documentation
- H2 Database project
- Lombok project for code generation
- JUnit and Mockito testing frameworks
- TUS Object-Oriented Programming I course materials

Version: 1.0.0

Last Updated: November 2025