Credit Card Management Portal

Zeta- Software Development Foundation Program

Abhay Dhek (Team Lead) - Card Details and User Dashboard

Anchal Jakhar – JWT Authentication and user profile management

Nikil Saini – My Cards Section and Admin Dashboard

Sumit Negi- Card Application Form and Application History, Admin Application Updates.

Saurav Kumar

Table of Contents

1. Project Overview

1.1 Project Title

SwipeSmart Credit Management Platform

1.2 Project Objective

The SwipeSmart Credit Management Platform is a state-of-the-art web application developed to empower customers of a financial institution to manage their credit cards and Buy Now Pay Later (BNPL) plans with unparalleled ease and security. The platform offers a robust suite of features, enabling users to view card details, apply for new credit cards, activate or deactivate cards, adjust credit limits, simulate transactions, and access detailed transaction histories, including BNPL-specific insights. Additionally, it provides administrative tools for reviewing applications, managing user accounts, and monitoring transactions. The primary objective is to deliver a seamless, user-centric experience that enhances customer satisfaction, reduces operational overhead for the bank, and supports modern financial flexibility through BNPL integration. By adhering to stringent security standards and leveraging a scalable architecture, SwipeSmart ensures reliability, compliance, and adaptability to future financial innovations.

1.3 Technology Stack

The platform is built on a modern, industry-standard technology stack designed for performance, scalability, and developer productivity:

Backend

- **Java 21 with Spring Boot 3.5.5**: Provides a high-performance framework for building RESTful APIs, enabling modular service development and seamless integration with external systems.
- **Spring Security with JWT**: Ensures secure user authentication and role-based authorization, safeguarding sensitive operations like card management and transactions.
- **Spring Data JPA with Hibernate**: Facilitates efficient data persistence and object-relational mapping for PostgreSQL (production).
- **Maven and Lombok**: Streamline dependency management and reduce boilerplate code, enhancing development efficiency.

Frontend

- **Vue.js 3.4.38 with TypeScript 5.8.3**: Delivers a reactive, type-safe, and component-based user interface for a dynamic and responsive user experience.
- **Vite 7.1.7 and Tailwind CSS 4.1.13**: Enable rapid development with fast builds and utility-first styling, ensuring responsive and accessible designs.

- **Vue Router 4.4.5 and Vuex 4.1.0**: Support client-side navigation and centralized state management for a cohesive user journey.
- **Chart.js 4.5.0 and jsPDF 3.0.3**: Provide interactive data visualizations for analytics and PDF export capabilities for transaction reports.
- **Axios 1.12.2**: Handles HTTP requests for efficient communication with backend APIs.

Database

- **PostgreSQL 15.x**: Serves as the primary relational database, optimized for production with HikariCP for efficient connection pooling.

Testing and Development

- **JUnit 5 and Mockito**: Enable comprehensive unit and integration testing for backend services, targeting 80%+ code coverage.
- **Vitest 3.2.4 with Vue Test Utils 2.4.6**: Support unit and component testing for the frontend, ensuring robust UI functionality.
- **JSDOM 27.0.0**: Simulates the DOM for frontend testing, enabling accurate validation of Vue.js components.
- **Cypress**: Facilitates end-to-end testing of user workflows, such as login, transaction simulation, and BNPL payment cycles.

1.4 Key Functionalities

The SwipeSmart platform offers a comprehensive set of features tailored for both end-users and administrators, leveraging RESTful APIs for seamless interaction:

User-Facing Features

- **Secure Authentication**: JWT-based login, registration, and token refresh for secure access.
- **Credit Card Management**: View card details, toggle card status (active/blocked), and manage multiple cards.
- **Card Applications**: Submit and track applications for new credit cards with real-time status updates.
- **Transaction Management**: Simulate regular and BNPL transactions, view paginated transaction histories, and export reports as PDFs.
- **BNPL Capabilities**: Split purchases into installments with flexible tenures (3, 6, 9, or 12 months), track payment progress, and manage EMI schedules.
- **Analytics Dashboard**: Visualize spending trends and category breakdowns using interactive charts powered by Chart.js.
- **Profile Management**: Update personal details, including address and income, to maintain BNPL eligibility.
- **Real-Time Notifications**: Display toast notifications for user actions like transaction creation or status changes.

Administrative Features

- **Admin Dashboard**: Provides an overview of system metrics, user activities, and transaction volumes.
- **Application Review Workflow**: Enables administrators to review, approve, or reject card applications with audit trails.
- **User and Transaction Oversight**: Supports management of user accounts and monitoring of transactions for compliance and fraud detection.

1.5 Buy Now Pay Later (BNPL) Feature

The BNPL feature is a cornerstone of the platform, offering users flexible payment options:

- **Installment Plans**: Users can split large purchases into manageable monthly installments with tenure options of 3, 6, 9, or 12 months.
- **Payment Tracking**: Visual indicators display payment progress, remaining balances, and upcoming dues.
- **EMI Management**: Users can make payments via card, with automatic credit limit adjustments based on BNPL usage.
- **History and Insights**: Detailed payment histories and BNPL-specific analytics help users plan finances effectively.
  This feature is supported by dedicated APIs (e.g., `/api/bnpl/overview`, `/api/bnpl/payment`) to ensure seamless integration and real-time updates.

1.6 Target Audience

- **Primary Users**: Credit cardholders seeking intuitive, digital tools to manage their cards and BNPL plans efficiently.
- **Secondary Users**: Bank administrators and customer service representatives responsible for application reviews, user management, and transaction monitoring.
- **Tertiary Users**: Financial institutions looking for scalable, secure solutions to enhance their credit management offerings.

1.7 Repository

- **Source Control**: Hosted in a private Git repository to ensure secure collaboration among development teams.
- **Branching Strategy**: Employs a feature-branch workflow with `main` and `develop` branches, facilitating iterative development and stable releases.
- **Versioning**: Maintains a comprehensive commit history for traceability, auditability, and rollback capabilities.

2. System Architecture

The SwipeSmart Credit Management Platform is built on a modular, client-server architecture designed for scalability, maintainability, and security. The system integrates a robust backend, a dynamic frontend, a relational database, and containerized deployment to deliver a seamless user experience for credit card and Buy Now Pay Later (BNPL) management. This section outlines the architectural components, their interactions, and the technologies enabling the platform's functionality.

2.1 Backend (Spring Boot)

The backend is implemented using **Spring Boot 3.5.5** with **Java 21**, providing a high-performance, RESTful API-driven foundation for the platform. Key components include:

- **RESTful APIs**: The backend exposes a comprehensive set of REST endpoints to support user authentication, card management, transaction processing, BNPL operations, and analytics. Examples include `/api/auth/login` for user authentication, `/api/cards` for retrieving card details, and `/api/bnpl/payment` for processing BNPL installments. These APIs follow REST principles, ensuring stateless communication and scalability.
- **Spring Security with JWT**: Authentication and authorization are managed using JSON Web Tokens (JWT), securing endpoints with bearer token authentication. Role-based access control (RBAC) distinguishes between user and admin privileges, protecting sensitive operations like card limit updates and application reviews.
- **Spring Data JPA with Hibernate**: Facilitates object-relational mapping (ORM) for efficient data persistence. Repositories handle CRUD operations for entities like User, Card, Transaction, and BNPL Installments, with custom queries (e.g., for transaction analytics) optimized for performance.
- **Service Layer**: Encapsulates business logic, such as transaction validation, credit limit adjustments, and BNPL installment calculations. Services like `CardService`, `TransactionService`, and `BnplService` ensure modularity and reusability.
- **Maven and Lombok**: Maven manages dependencies and build processes, while Lombok reduces boilerplate code for entities and services, improving developer productivity.

The backend is designed for high throughput, with asynchronous processing for non-blocking operations (e.g., transaction creation) and caching (planned for future enhancements) to optimize frequent API calls.

2.2 Frontend (Vue.js)

The frontend is built using **Vue.js 3.4.38** with **TypeScript 5.8.3**, delivering a reactive, component-based user interface optimized for usability and responsiveness. Key aspects include:

- **Component Architecture**: The frontend is organized into reusable Vue components, such as `MyCards.vue` for card management, `CreateTransaction.vue` for

transaction simulation, and `BnplSection.vue` for BNPL tracking. Components leverage the Composition API for better state management and TypeScript for type safety.

- **Vue Router 4.4.5**: Handles client-side navigation, enabling seamless transitions between views like the dashboard, transaction history, and profile pages. Routes are protected with navigation guards to enforce authentication.
- **Vuex 4.1.0**: Manages centralized state for user data, card details, and transaction histories, ensuring consistent data flow across components.
- **Tailwind CSS 4.1.13**: Provides utility-first styling for responsive, accessible designs. The frontend adheres to a mobile-first approach, with layouts optimized for various screen sizes.
- **Vite 7.1.7**: Serves as the build tool and development server, offering fast compilation and hot module replacement for efficient development.
- **Chart.js 4.5.0 and jsPDF 3.0.3**: Enable interactive visualizations for spending analytics and PDF export for transaction reports, respectively.
- **Axios 1.12.2**: Facilitates HTTP requests to backend APIs, with interceptors for handling authentication tokens and error responses.

The frontend prioritizes performance with code splitting, lazy loading, and optimized asset delivery, ensuring a smooth user experience.

2.3 Database

The platform uses a relational database to store and manage data, with a primary focus on **PostgreSQL 15.x** for production.

PostgreSQL (Production): A robust, open-source relational database chosen for its scalability, advanced querying capabilities, and compatibility with Spring Data JPA. It hosts tables for Users, Cards, Card Applications, Transactions, User Profiles, and BNPL Installments, with optimized indexes for frequent queries (e.g., transaction history by card ID).

- **Connection Pooling with HikariCP**: Spring Boot's default connection pooling mechanism optimizes database connections, ensuring efficient resource utilization and low latency for API requests.
- **ORM with JPA/Hibernate**: Maps database tables to Java entities, enabling seamless data operations. Custom JPA queries (e.g., for analytics or BNPL payment tracking) are implemented in repositories to support complex business logic.

The database schema is normalized to maintain data integrity, with foreign key relationships (e.g., User to Cards, Card to Transactions) ensuring referential consistency.

3. Schema Design

The SwipeSmart Credit Management Platform relies on a normalized relational database schema to ensure data integrity, efficient querying, and support for core functionalities such as credit card management, transaction processing, and Buy Now Pay Later (BNPL) operations. The schema is implemented in **PostgreSQL 15.x** for production. This section details the database tables, their fields, data types, descriptions, and relationships, designed to support the platform's RESTful APIs and business logic.

3.1 User Table

The User table stores core information for authentication and identification of platform users.

Field	Type	Description
id	BIGINT	Primary key, auto-incremented
name	VARCHAR(100)	Full name of the user
email	VARCHAR(255)	Unique email for login and identification
phoneNumber	VARCHAR(20)	Optional contact phone number
password	VARCHAR(255)	Hashed password (BCrypt) for secure login
role	VARCHAR(40)	To assign role(user/admin)
createdAt	TIMESTAMP	Record creation timestamp
updatedAt	TIMESTAMP	Last update timestamp

Constraints:

- Primary Key: `id`
- Unique Constraint: `email`
- Not Null: `name`, `email`, `password`, `role`

3.2 Card Table

The Card table manages credit card details, linked to users via a foreign key.

Field	Type	Description
id	BIGINT	Primary key, auto-incremented
cardNumber	VARCHAR(16)	Unique card number (masked in UI)
cardTypeId	BIGINT	Foreign key referencing Card Type
status	VARCHAR(20)	Card status (ACTIVE, BLOCKED, INACTIVE)
creditLimit	DECIMAL(15,2)	Maximum allowed credit
availableLimit	DECIMAL(15,2)	Remaining spendable limit
expiryDate	DATE	Card expiration date
userId	BIGINT	Foreign key referencing User
createdAt	TIMESTAMP	Record creation timestamp
updatedAt	TIMESTAMP	Last update timestamp

Constraints:

- Primary Key: `id`
- Unique Constraint: `cardNumber`
- Foreign Key: `userId` → User(id), `cardTypeId` → CardType(id)
- Not Null: `cardNumber`, `cardTypeId`, `status`, `creditLimit`, `availableLimit`, `expiryDate`, `userId`

3.3 Card Applications Table

The Card Applications table tracks requests for new credit cards and their processing status.

Field	Type	Description
id	BIGINT	Primary key, auto-incremented
userId	BIGINT	Foreign key referencing User

Field	Type	Description
cardTypeId	BIGINT	Foreign key referencing Card Type
requestedLimit	DECIMAL(15,2)	Requested credit limit
annualIncome	DECIMAL(15,2)	User's annual income for eligibility
employmentStatus	VARCHAR(50)	Employment status (e.g., EMPLOYED, SELF_EMPLOYED)
companyName	VARCHAR(100)	Optional employer name
applicationDate	DATE	Date of application submission
status	VARCHAR(20)	Application status (PENDING, APPROVED, REJECTED)
reviewedBy	VARCHAR(100)	Optional admin/reviewer identifier
reviewDate	DATE	Optional date of review completion
createdAt	TIMESTAMP	Record creation timestamp
updatedAt	TIMESTAMP	Last update timestamp

Constraints:

- Primary Key: `id`
- Foreign Key: `userId` → User(id), `cardTypeId` → CardType(id)
- Not Null: `userId`, `cardTypeId`, `requestedLimit`, `annualIncome`, `employmentStatus`, `applicationDate`, `status`

3.4 Transactions Table

The Transactions table records all credit card transactions, including BNPL transactions.

Field	Type	Description
id	BIGINT	Primary key, auto-incremented
cardId	BIGINT	Foreign key referencing Card

Field	Type	Description
merchantId	BIGINT	Foreign key referencing Merchant
amount	DECIMAL(15,2)	Transaction amount
transactionDate	DATE	Date of transaction
category	VARCHAR(50)	Transaction category (e.g., SHOPPING, TRAVEL)
isBnpl	BOOLEAN	Indicates if transaction uses BNPL
createdAt	TIMESTAMP	Record creation timestamp
updatedAt	TIMESTAMP	Last update timestamp

Constraints:

- Primary Key: `id`
- Foreign Key: `cardId` → Card(id), `merchantId` → Merchant(id)
- Not Null: `cardId`, `merchantId`, `amount`, `transactionDate`, `category`, `isBnpl`

3.5 User Profiles Table

The User Profiles table extends user information for personalization and eligibility assessment.

Field	Type	Description
id	BIGINT	Primary key, auto-incremented
userId	BIGINT	Foreign key referencing User
fullName	VARCHAR(100)	User's full name
email	VARCHAR(255)	Email address (synced with User)
phone	VARCHAR(20)	Contact phone number
address	VARCHAR(255)	Residential address
annualIncome	DECIMAL(15,2)	Annual income for credit eligibility

Field	Type	Description
isEligibleBnpl	boolean	Whether the user is eligible for bnpl or not
createdAt	TIMESTAMP	Record creation timestamp
updatedAt	TIMESTAMP	Last update timestamp

Constraints:

- Primary Key: `id`
- Foreign Key: `userId` → User(id)
- Unique Constraint: `email`
- Not Null: `userId`, `fullName`, `email`, `annualIncome`

3.6 BNPL Installments Table

The BNPL Installments table tracks payment schedules for BNPL transactions.

Field	Type	Description
id	BIGINT	Primary key, auto-incremented
transactionId	BIGINT	Foreign key referencing Transaction
installmentNumber	INTEGER	Sequential number of the installment
amount	DECIMAL(15,2)	Installment amount
dueDate	DATE	Payment due date
status	VARCHAR(20)	Status (PENDING, PAID, OVERDUE)
paymentDate	DATE	Optional date of payment completion
paymentMethod	VARCHAR(50)	Payment method (e.g., CARD, BANK_TRANSFER)
createdAt	TIMESTAMP	Record creation timestamp
updatedAt	TIMESTAMP	Last update timestamp

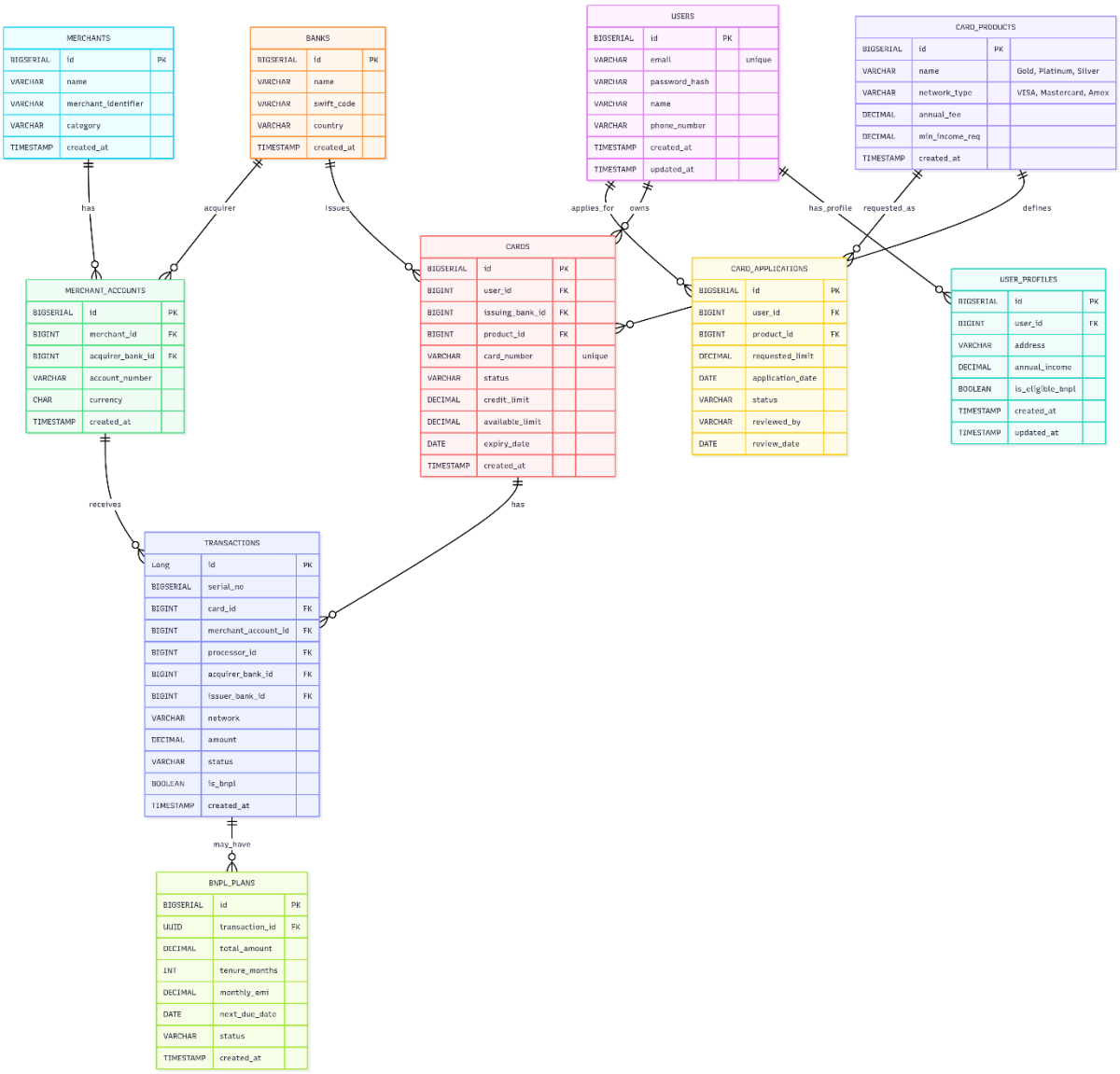
Constraints:

- Primary Key: `id`
- Foreign Key: `transactionId` → Transaction(id)
- Not Null: `transactionId`, `installmentNumber`, `amount`, `dueDate`, `status`

3.7 Relationships

The schema is designed with the following relationships to ensure data consistency and support business logic:

- **One-to-Many**: User → Cards (one user can own multiple credit cards).
- **One-to-Many**: User → Card Applications (one user can submit multiple applications).
- **One-to-Many**: Card → Transactions (one card can have multiple transactions).
- **One-to-Many**: Transaction → BNPL Installments (one BNPL transaction can have multiple installments).
- **Many-to-One**: Card → Card Type (multiple cards can belong to a single card type, e.g., VISA, Mastercard).
- **One-to-One**: User → User Profile (each user has a single profile for extended details).
- **Many-to-One**: Transaction → Merchant (multiple transactions can reference a single merchant).



4. Setup and Configuration

The SwipeSmart Credit Management Platform requires a structured setup process to ensure seamless operation across development, testing, and production environments. This section provides comprehensive instructions for configuring the backend (Spring Boot), frontend (Vue.js), database (PostgreSQL). The setup ensures compatibility with the platform's RESTful APIs, secure authentication, and Buy Now Pay Later (BNPL) functionality, enabling developers to initialize the system efficiently.

4.1 Backend Setup

The backend is built with **Spring Boot 3.5.5** and **Java 21**, utilizing **Maven** for dependency management and **Spring Data JPA** for database interactions. The following steps outline the setup process.

Prerequisites

- **Java 21**: Install the latest JDK (e.g., OpenJDK or Oracle JDK).
- **Maven 3.9.6+**: Ensure Maven is configured for dependency resolution.
- **PostgreSQL 15.x**: Required for production database setup.
  IDE: Recommended tools include IntelliJ IDEA or Eclipse for development.

Steps

1. **Clone Repository**:
   - Clone the private Git repository:
   - `git clone https://github.com/nikilsaini89/Credit-Card-Management-Portal`
     `cd credit-card-management-portal -backend`
   - Ensure access credentials are configured for the private repository.
2. **Configure Environment**:
   - Create a `application.properties` file in `src/main/resources` to define environment-specific settings:
   - `spring:`
   - `datasource:`
   - `url: jdbc:postgresql://<db-host>:<db-port>/smartswipe`
   - `username: <db-username>`
   - `password: <db-password>`
   - `driver-class-name: org.postgresql.Driver`
   - `jpa:`
   - `hibernate:`
   - `ddl-auto: update`
   - `show-sql: true`
   - `jwt:`
   - `secret: <secure-jwt-secret>`
   - `expiration: 86400000`
3. **Install Dependencies**:
   - Run Maven to resolve dependencies:
   - `mvn clean install`
4. **Build and Run**:
   - Start the Spring Boot application:
   - `mvn spring-boot:run`
   - The backend will initialize, creating the database schema via JPA and exposing RESTful APIs (e.g., `/api/auth/login`, `/api/cards`).
5. **Verify Setup**:
   - Confirm the application is running by accessing the health endpoint or testing APIs via tools like Postman.

4.2 Frontend Setup

The frontend is built with **Vue.js 3.4.38**, **TypeScript 5.8.3**, and **Vite 7.1.7**, styled with **Tailwind CSS 4.1.13**. The following steps ensure the frontend is properly configured.

Prerequisites

- **Node.js 18.x+**: Install the latest LTS version.

- **npm 9.x+ or Yarn 1.x+**: Package manager for dependency installation.
- **IDE**: Recommended tools include Visual Studio Code with Vue and TypeScript extensions.

Steps

1. **Navigate to Frontend Directory**:
   - Access the frontend codebase:
   - `cd credit-card-management-portal-frontend`
2. **Install Dependencies**:
   - Install Node.js packages:
   - `npm install`

     or

     `yarn install`

3. **Configure Environment**:
   - Create a `.env` file in the root of the frontend directory:
   - `VITE_API_BASE_URL=<backend-api-url>`
4. Replace `<backend-api-url>` with the production or staging backend URL .
5. **Run Development Server**:
   - Start the Vite development server:
   - `npm run dev`

     or

     `yarn dev`

   - The frontend will be accessible, typically at `http://localhost:5173` during development.
6. **Build for Production**:
   - Generate optimized production assets:
   - `npm run build`
   - The output will be in the `dist` directory, ready for deployment.
7. **Verify Setup**:
   - Access the frontend in a browser to confirm components (e.g., `MyCards.vue`, `BnplSection.vue`) render correctly and communicate with backend APIs.

4.3 Database Setup

The platform uses **PostgreSQL 15.x** for production with **HikariCP** for connection pooling and **JPA/Hibernate** for schema management.

PostgreSQL Setup

1. **Install PostgreSQL**:

- Install PostgreSQL 15.x on the target environment
- Ensure the service is running:
- `sudo systemctl start postgresql`

2. **Create Database**:

   - Access the PostgreSQL CLI:
   - `psql -U postgres`
   - Create the database:
   - `CREATE DATABASE smartswipe;`

3. **Configure Access**:

   - Update `pg_hba.conf` to allow connections (if needed) and set credentials in the backend's `application.yml` (see Backend Setup).

4. **Initialize Schema**:

   - JPA/Hibernate automatically generates the schema based on entity definitions (e.g., User, Card, Transaction) when `spring.jpa.hibernate.ddl-auto` is set to `update` or `create`.

Notes

- **Environment Variables**: Secure sensitive values (e.g., `JWT_SECRET`, database credentials) using environment files or a secrets manager in production.
- **Schema Initialization**: JPA automatically manages schema creation, but manual SQL scripts can be used for custom seeding or migrations.
- This setup ensures the SwipeSmart platform is fully operational, with all components configured to support secure authentication, card management, transaction processing, and BNPL functionality.

5. API Documentation

The SwipeSmart Credit Management Platform exposes a comprehensive set of RESTful APIs to support user authentication, credit card management, transaction processing, Buy Now Pay Later (BNPL) operations, and analytics. Built with **Spring Boot 3.5.5**, the APIs utilize **JWT-based authentication** for security and follow REST conventions for stateless, scalable interactions. This section details the key endpoints, their parameters, and example requests/responses, providing a clear reference for developers integrating with or extending the platform.

5.1 Endpoints

The APIs are organized by functional modules, with each endpoint requiring appropriate authentication headers where specified. The base URL is represented as `<api-base-url>` (e.g., `https://api.smartswipe.com` in production).

Authentication Endpoints

These endpoints manage user authentication and session lifecycle.

- **POST /api/auth/register**
  Description: Registers a new user account.
  Headers:
    - Content-Type: application/json

  Body:

```
{
  "name": String,
  "email": String,
  "phoneNumber": String,
  "password": String
}
```

  Response Codes:
    - 200: Successful registration
    - 400: Invalid input (e.g., duplicate email)
      Authentication: None

- **POST /api/auth/login**
  Description: Authenticates a user and returns a JWT token.
  Headers:
    - Content-Type: application/json
      Body:

```
{
  "email": String,
  "password": String
}
```

  Response Codes:
    - 200: Successful login
    - 401: Invalid credentials
      Authentication: None

- **POST /api/auth/refresh**
  Description: Refreshes an expired JWT token using a refresh token.
  Headers:
    - Content-Type: application/json
    - Authorization: Bearer
      Response Codes:
    - 200: Token refreshed

- 401: Invalid or expired refresh token
  Authentication: JWT (refresh token)

- **POST /api/auth/logout**
  Description: Invalidates the user's session.
  Headers:
  - Authorization: Bearer
    Response Codes:
  - 200: Successful logout
  - 401: Invalid token
    Authentication: JWT

Card Management Endpoints

These endpoints handle credit card operations.

- **GET /api/cards**
  Description: Retrieves all credit cards for the authenticated user.
  Headers:
  - Authorization: Bearer
    Query Parameters: None
    Response Codes:
  - 200: Cards retrieved
  - 401: Unauthorized
    Authentication: JWT

- **GET /api/card-types**
  Description: Lists available card types (e.g., VISA, Mastercard).
  Headers:
  - Authorization: Bearer
    Response Codes:
  - 200: Card types retrieved
  - 401: Unauthorized
    Authentication: JWT

- **PUT /api/cards/{cardId}/status**
  Description: Updates the status of a specific card (e.g., ACTIVE, BLOCKED).
  Headers:
  - Content-Type: application/json
  - Authorization: Bearer
    Path Parameters:
  - cardId: BIGINT (Card ID)

  Body:

  {

```
  "status": String (e.g., "ACTIVE", "BLOCKED")
}
```

Response Codes:

- 200: Status updated
- 400: Invalid status
- 401: Unauthorized
- 404: Card not found

Authentication: JWT

- **PUT /api/cards/limit**
  Description: Updates the credit limit for a specific card.
  Headers:
  - Content-Type: application/json
  - Authorization: Bearer

  Body:

```
{
  "cardId": BIGINT,
  "newLimit": DECIMAL
}
```

  Response Codes:

  - 200: Limit updated
  - 400: Invalid limit
  - 401: Unauthorized
  - 404: Card not found

  Authentication: JWT

Card Application Endpoints

These endpoints manage credit card applications.

- **POST /api/card-applications**
  Description: Submits a new credit card application.
  Headers:
  - Content-Type: application/json
  - Authorization: Bearer

  Body:

```
{
  "cardTypeId": BIGINT,
  "annualIncome": DECIMAL,
  "employmentStatus": String,
  "companyName": String
}
```

Response Codes:

1. 200: Application submitted
2. 400: Invalid input
3. 401: Unauthorized
   [Authentication: JWT**]**

- **GET /api/card-applications/my-applications**
  Description: Retrieves the authenticated user's card applications.
  Headers:

  - Authorization: Bearer
    Response Codes:
  - 200: Applications retrieved
  - 401: Unauthorized
    Authentication: JWT

Transaction Endpoints

These endpoints handle transaction creation and retrieval.

- **POST /api/transactions**
  Description: Creates a regular or BNPL transaction.
  Headers:

  - Content-Type: application/json
  - Authorization: Bearer

  Body:

```
{
  "cardId": BIGINT,
  "merchantName": String,
  "amount": DECIMAL,
  "category": String,
  "isBnpl": BOOLEAN,
  "transactionDate": String (ISO 8601 format)
}
```

  Response Codes:

  - 200: Transaction created
  - 400: Invalid input (e.g., negative amount)
  - 401: Unauthorized
  - 403: BNPL not eligible
  - 404: Card not found
    Authentication: JWT

- **GET /api/transactions**
  Description: Retrieves paginated transaction history for a card.
  Headers:

- Authorization: Bearer
  Query Parameters:

- cardId: BIGINT (required)
- page: INTEGER (default: 0)
- size: INTEGER (default: 10)
  Response Codes:

- 200: Transactions retrieved
- 401: Unauthorized
- 404: Card not found
  Authentication: JWT

- **GET /api/transactions/{transactionId}**
  Description: Retrieves details of a specific transaction.
  Headers:

  - Authorization: Bearer
    Path Parameters:
  - transactionId: BIGINT
    Response Codes:

  - 200: Transaction retrieved
  - 401: Unauthorized
  - 404: Transaction not found
    Authentication: JWT

- **GET /api/transactions/{cardId}/analytics**
  Description: Retrieves spending analytics for a card.
  Headers:

  - Authorization: Bearer
    Path Parameters:
  - cardId: BIGINT

    Response Codes:

    - 200: Analytics retrieved
    - 401: Unauthorized
    - 404: Card not found
      Authentication: JWT

BNPL Endpoints

These endpoints manage BNPL transactions and payments.

- **GET /api/bnpl/overview/{cardId}**
  Description: Retrieves BNPL overview for a card, including active plans and balances.
  Headers:

  - Authorization: Bearer
    Path Parameters:

  - cardId: BIGINT
    Response Codes:

  - 200: Overview retrieved
  - 401: Unauthorized
  - 404: Card not found
    Authentication: JWT

- **POST /api/bnpl/payment**
  Description: Processes a BNPL installment payment.
  Headers:

  - Content-Type: application/json
  - Authorization: Bearer

  Body:

```
{
  "transactionId": BIGINT,
  "paymentAmount": DECIMAL,
  "paymentMethod": String,
  "notes": String
}
```

  Response Codes:

  - 200: Payment processed
  - 400: Invalid payment amount
  - 401: Unauthorized
  - 404: Transaction not found
    Authentication: JWT

- **GET /api/bnpl/payments/{transactionId}**
  Description: Retrieves payment history for a BNPL transaction.
  Headers:

  - Authorization: Bearer
    Path Parameters:
  - transactionId: BIGINT
    Response Codes:
  - 200: Payment history retrieved
  - 401: Unauthorized

- 404: Transaction not found
  Authentication: JWT

Statement Endpoints

These endpoints manage credit card statements.

- **GET /api/statements/card/{cardId}/current**
  Description: Retrieves the current statement for a card.
  Headers:
  - Authorization: Bearer
    Path Parameters:
  - cardId: BIGINT
    Response Codes:
  - 200: Statement retrieved
  - 401: Unauthorized
  - 404: Card not found
    Authentication: JWT

- **GET /api/statements/card/{cardId}/history**
  Description: Retrieves historical statements for a card.
  Headers:
  - Authorization: Bearer
    Path Parameters:
  - cardId: BIGINT
    Response Codes:
  - 200: Statements retrieved
  - 401: Unauthorized
  - 404: Card not found
    Authentication: JWT

- **POST /api/statements/payment**
  Description: Processes a payment for a card statement.
  Headers:
  - Content-Type: application/json
  - Authorization: Bearer

  Body:

```
{
  "cardId": BIGINT,
  "paymentAmount": DECIMAL,
  "paymentMethod": String
}
```

  Response Codes:

- 200: Payment processed
- 400: Invalid payment amount
- 401: Unauthorized
- 404: Card not found

Authentication: JWT

Analytics Endpoints

These endpoints provide spending insights.

- **GET /api/analytics/transactions/{cardId}**
  Description: Retrieves transaction analytics for a card.
  Headers:
  - Authorization: Bearer

  Path Parameters:
  - cardId: BIGINT

  Response Codes:
  - 200: Analytics retrieved
  - 401: Unauthorized
  - 404: Card not found

  Authentication: JWT

- **GET /api/analytics/spending/category/{cardId}**
  Description: Retrieves spending breakdown by category.
  Headers:
  - Authorization: Bearer

  Path Parameters:
  - cardId: BIGINT

  Response Codes:
  - 200: Category analytics retrieved
  - 401: Unauthorized
  - 404: Card not found

  Authentication: JWT

- **GET /api/analytics/trends/monthly/{cardId}**
  Description: Retrieves monthly spending trends.
  Headers:
  - Authorization: Bearer

  Path Parameters:
  - cardId: BIGINT

  Response Codes:
  - 200: Trends retrieved
  - 401: Unauthorized
  - 404: Card not found

  Authentication: JWT

Merchant Endpoints

These endpoints manage merchant data for transactions.

- **GET /api/merchants**
  Description: Retrieves a list of available merchants.
  Headers:
    - Authorization: Bearer
      Query Parameters: None
      Response Codes:
    - 200: Merchants retrieved
    - 401: Unauthorized
      Authentication: JWT

- **GET /api/merchants?category={category}**
  Description: Retrieves merchants filtered by category (e.g., E-commerce).
  Headers:
    - Authorization: Bearer
      Query Parameters:
    - category: String (optional)
      Response Codes:
    - 200: Merchants retrieved
    - 401: Unauthorized
      Authentication: JWT

5.2 Example Request/Response

Below are example requests and responses for key endpoints, demonstrating typical usage.

Create Transaction (POST /api/transactions)

Request:

```
POST <api-base-url>/api/transactions
Content-Type: application/json
Authorization: Bearer <jwt-token>

{
  "cardId": 1,
  "merchantName": "Amazon",
  "amount": 2500.00,
  "category": "SHOPPING",
  "isBnpl": false,
  "transactionDate": "2025-09-28"
}
```

Response (200 OK):

```
{
  "id": 1001,
  "cardId": 1,
  "merchantName": "Amazon",
  "amount": 2500.00,
  "category": "SHOPPING",
  "isBnpl": false,
  "transactionDate": "2025-09-28",
  "createdAt": "2025-09-30T18:44:00Z",
  "updatedAt": "2025-09-30T18:44:00Z"
}
```

Response (400 Bad Request):

```
{
  "error": "Invalid transaction amount: Amount must be positive"
}
```

Get BNPL Overview (GET /api/bnpl-overview/{cardId})

Request:

```
GET <api-base-url>/api/bnpl/overview/1
Authorization: Bearer <jwt-token>
```

Response (200 OK):

```
{
  "cardId": 1,
  "activeBnplPlans": [
    {
      "transactionId": 1002,
      "totalAmount": 25000.00,
      "remainingBalance": 16666.68,
      "tenureMonths": 6,
      "monthlyInstallment": 4166.67,
      "nextDueDate": "2025-10-28"
    }
  ],
  "totalBnplBalance": 16666.68
}
```

Response (404 Not Found):

```
{
  "error": "Card not found"
}
```

Update User Profile (PUT /api/profile)

Request:

```
PUT <api-base-url>/api/profile
Content-Type: application/json
Authorization: Bearer <jwt-token>

{
```

```
  "address": "123 Test Street, Test City, Test State 12345",
  "annualIncome": 750000.00,
  "isEligibleBnpl": true
}
```

Response (200 OK):

```
{
  "id": 1,
  "userId": 1,
  "fullName": "Test User",
  "email": "testuser@example.com",
  "phone": "9876543210",
  "address": "123 Test Street, Test City, Test State 12345",
  "annualIncome": 750000.00,
  "isEligibleBnpl": true,
  "createdAt": "2025-09-30T18:44:00Z",
  "updatedAt": "2025-09-30T18:44:00Z"
}
```

Response (400 Bad Request):

```
{
  "error": "Invalid email format"
}
```

Notes

- **Authentication**: Most endpoints require a valid JWT token in the `Authorization` header, obtained via `/api/auth/login`.
- **Error Handling**: Responses include descriptive error messages for debugging (e.g., "BNPL Not Available" for ineligible BNPL transactions).
- **Pagination**: Endpoints like `/api/transactions` support pagination via `page` and `size` query parameters.
- **Security**: Input validation and sanitization prevent common vulnerabilities (e.g., SQL injection, XSS).
- **Extensibility**: The API design allows for future additions, such as advanced analytics or third-party integrations.

6. Test Plan

The SwipeSmart Credit Management Platform requires a robust testing strategy to ensure reliability, security, and performance across its features, including user authentication, credit card management, transaction processing, and Buy Now Pay Later (BNPL) operations. The test plan encompasses unit tests, integration tests, and end-to-end tests, targeting a minimum code coverage of 80%. Testing leverages **JUnit 5**, **Vitest**, **Vue Test Utils**, **Cypress**, and **JSDOM** to validate backend, frontend, and database interactions. This section outlines the testing approach, tools, and specific test cases to ensure the platform meets functional and non-functional requirements.

6.1 Unit Tests

Unit tests validate individual components in isolation, focusing on backend services, controllers, and frontend components.

Backend Unit Tests

- **Tools**: JUnit 5, Mockito
- **Scope**: Tests Spring Boot services (e.g., `CardService`, `TransactionService`, `BnplService`), controllers, and repository methods.
- **Objectives**:
    - Verify business logic, such as credit limit calculations, BNPL installment generation, and authentication workflows.
    - Mock dependencies (e.g., repositories, external services) to isolate unit behavior.
- **Coverage**: Target 80%+ coverage for service and controller layers.
- **Example Tests**:
    - Validate `TransactionService.createTransaction` for correct amount validation and credit limit checks.
    - Test `BnplService.calculateInstallments` for accurate tenure-based EMI calculations.

Frontend Unit Tests

- **Tools**: Vitest, Vue Test Utils, JSDOM
- **Scope**: Tests Vue.js components (e.g., `MyCards.vue`, `CreateTransaction.vue`, `BnplSection.vue`) for rendering, event handling, and state management.
- **Objectives**:
    - Ensure components render correctly with mock data.
    - Validate user interactions (e.g., button clicks, form submissions) and Vuex state updates.
- **Coverage**: Target 80%+ coverage for critical components.
- **Example Tests**:
    - Verify `MyCards.vue` displays card details and status toggles correctly.
    - Test `BnplSection.vue` for rendering BNPL payment progress indicators.

7. Implementation Details

The SwipeSmart Credit Management Platform is structured into modular components that deliver a seamless experience for managing credit cards and Buy Now Pay Later (BNPL)

operations. This section details the implementation of key modules, including their backend services (Spring Boot), frontend components (Vue.js), and associated APIs. Each module is designed to handle specific functionalities, such as viewing and managing cards, applying for new cards, adjusting credit limits, processing transactions, managing user profiles, and handling BNPL workflows. The implementation leverages a robust technology stack, ensuring scalability, security, and maintainability.

7.1 Module 1: View Credit Cards and Activate/Block

Purpose: Enables users to view their credit cards and toggle their status (e.g., ACTIVE, BLOCKED).

- **Frontend Implementation**:
  - **Component**: `MyCards.vue`
  - **Description**: A Vue.js component that displays a list of user-owned credit cards with details like card number (masked), status, credit limit, and available limit. The component uses **Vuex** for state management and **Axios** to fetch card data from the backend. It includes interactive buttons for activating or blocking cards, with real-time status updates and toast notifications for user feedback.
  - **Technologies**: Vue.js 3.4.38, TypeScript 5.8.3, Tailwind CSS 4.1.13 for responsive styling.
- **Backend Implementation**:
  - **Service**: `CardService.java`
  - **Description**: Handles business logic for retrieving card details and updating card status. The service interacts with the `CardRepository` (Spring Data JPA) to query the Card table and enforces rules (e.g., only card owners can modify status).
  - **APIs**:
    - `GET /api/cards`: Retrieves all cards for the authenticated user.
      - **Headers**: Authorization: Bearer
      - **Response**: JSON array of card objects (e.g., `{ id, cardNumber, status, creditLimit, availableLimit }`).
    - `PUT /api/cards/{cardId}/status`: Updates the status of a specific card.
      - **Headers**: Content-Type: application/json, Authorization: Bearer
      - **Body**: `{ "status": "ACTIVE" | "BLOCKED" }`
      - **Response Codes**: 200 (success), 400 (invalid status), 401 (unauthorized), 404 (card not found).
- **Features**:
  - Displays cards with visual status indicators (e.g., green for ACTIVE, red for BLOCKED).
  - Validates user permissions before status updates.

- Updates the Card table's `status` and `updatedAt` fields in real time.

7.2 Module 2: Apply for Credit Card and View Requests

Purpose: Allows users to submit new credit card applications and track their status, with admin review capabilities.

- **Frontend Implementation**:
  - **Component**: CardApplicationPage.vue
  - **Description**: A Vue.js component with a form for submitting card applications, including fields for card type, annual income, employment status, and company name. The component validates inputs client-side and uses **Axios** to send requests to the backend. A separate view displays application history with statuses (PENDING, APPROVED, REJECTED).
  - **Technologies**: Vue.js, Vue Router 4.4.5 for navigation, Tailwind CSS for form styling.
- **Backend Implementation**:
  - **Service**: CardApplicationService.java
  - **Description**: Manages application creation, validation, and admin review workflows. Interacts with CardApplicationRepository to store and retrieve application data. Admin endpoints allow reviewing and updating application statuses.
  - **APIs**:
    - POST /api/card-applications: Submits a new card application.
      - **Headers**: Content-Type: application/json, Authorization: Bearer
      - **Body**: { "cardTypeId": BIGINT, "annualIncome": DECIMAL, "employmentStatus": String, "companyName": String }
      - **Response Codes**: 200 (success), 400 (invalid input), 401 (unauthorized).
    - GET /api/card-applications/my-applications: Retrieves the user's application history.
      - **Headers**: Authorization: Bearer
      - **Response**: JSON array of application objects (e.g., { id, cardTypeId, status, applicationDate }).
- **Features**:
  - Form validation ensures required fields (e.g., annual income, employment status) meet criteria.
  - Admin dashboard supports application approval/rejection with audit trails.
  - Updates the Card Applications table with status changes and timestamps.

7.3 Module 3: Manage Card Limit

Purpose: Enables users to request credit limit adjustments, subject to validation and approval.

- **Frontend Implementation**:
  - **Component**: UpdateCreditLimitModal.vue

- **Description**: A Vue.js modal component for requesting credit limit changes, displaying the current limit and allowing input for a new limit. Uses **Axios** to send requests and **Vuex** to update card state. Notifications confirm success or failure.
- **Technologies**: Vue.js, TypeScript, Tailwind CSS for modal styling.

- **Backend Implementation**:
  - **Service**: CreditLimitService.java
  - **Description**: Validates limit change requests against business rules (e.g., user creditworthiness, BNPL usage). Updates the Card table's creditLimit and availableLimit fields.
  - **API**:
    - PUT /api/cards/limit: Updates a card's credit limit.
      - **Headers**: Content-Type: application/json, Authorization: Bearer
      - **Body**: { "cardId": BIGINT, "newLimit": DECIMAL }
      - **Response Codes**: 200 (success), 400 (invalid limit), 401 (unauthorized), 404 (card not found).
- **Features**:
  - Validates new limits against user profile data (e.g., annual income).
  - Supports admin approval workflows for large limit changes.
  - Updates Card table and triggers notifications for status changes.

7.4 Module 4: Simulate and View Transactions

Purpose: Allows users to create transactions (regular or BNPL) and view transaction histories with analytics.

- **Frontend Implementation**:
  - **Component**: CreateTransaction.vue, TransactionHistory.vue
  - **Description**: CreateTransaction.vue provides a form for creating transactions, with fields for card selection, merchant, amount, category, and BNPL option. TransactionHistory.vue displays paginated transaction lists with filters (e.g., date, category) and export options (PDF via **jsPDF**). **Chart.js** powers spending analytics visualizations.
  - **Technologies**: Vue.js, Vuex, Chart.js 4.5.0, jsPDF 3.0.3, Axios.

- **Backend Implementation**:
  - **Service**: TransactionService.java
  - **Description**: Handles transaction creation, validation (e.g., sufficient credit limit), and analytics generation. For BNPL transactions, it coordinates with BnplService to create installment plans. Interacts with TransactionRepository and CardRepository.
  - **APIs**:
    - POST /api/transactions: Creates a new transaction.

        - **Headers**: Content-Type: application/json, Authorization: Bearer
        - **Body**: { "cardId": BIGINT, "merchantName": String, "amount": DECIMAL, "category": String, "isBnpl": BOOLEAN, "transactionDate": String }
        - **Response Codes**: 200 (success), 400 (invalid input), 401 (unauthorized), 403 (BNPL ineligible), 404 (card not found).
    - GET /api/transactions?cardId={cardId}&page={page}&size={size}: Retrieves paginated transaction history.
        - **Headers**: Authorization: Bearer
        - **Response**: JSON array of transactions.
    - GET /api/transactions/{cardId}/analytics: Retrieves spending analytics.
        - **Headers**: Authorization: Bearer
        - **Response**: JSON with spending trends and category breakdowns.
- **Features**:
    - Supports regular and BNPL transactions with real-time credit limit updates.
    - Provides filtering and pagination for transaction history.
    - Generates PDF reports and visual analytics for spending patterns.

7.5 Module 5: Customer Profile Management

Purpose: Enables users to view and update their profile details, including BNPL eligibility.

- **Frontend Implementation**:
    - **Component**: UserProfile.vue
    - **Description**: A Vue.js component for editing user details (e.g., address, annual income) and viewing BNPL eligibility status. Uses **Axios** for API calls and **Vuex** for state management, with form validation to ensure data integrity.
    - **Technologies**: Vue.js, TypeScript, Tailwind CSS.
- **Backend Implementation**:
    - **Service**: ProfileService.java
    - **Description**: Manages profile updates and BNPL eligibility checks based on user data (e.g., annual income). Interacts with UserProfileRepository to persist changes.
    - **APIs**:
        - GET /api/profile: Retrieves the user's profile.
            - **Headers**: Authorization: Bearer
            - **Response**: JSON with profile details (e.g., { fullName, email, address, annualIncome, isEligibleBnpl }).
        - PUT /api/profile: Updates the user's profile.
            - **Headers**: Content-Type: application/json, Authorization: Bearer
            - **Body**: { "address": String, "annualIncome": DECIMAL, "isEligibleBnpl": BOOLEAN }

        - **Response Codes**: 200 (success), 400 (invalid input), 401 (unauthorized).
- **Features**:
    - Validates profile updates (e.g., email format, income range).
    - Recalculates BNPL eligibility based on updated income or credit history.
    - Updates User Profiles table with timestamps.

7.6 BNPL Feature Implementation

Purpose: Provides flexible BNPL payment options, including installment creation and payment tracking.

- **Frontend Implementation**:
    - **Component**: BnplSection.vue
    - **Description**: Displays active BNPL plans, payment schedules, and progress indicators (e.g., progress bars for remaining balance). Allows users to make installment payments and view payment history. Uses **Chart.js** for visualizations and **Axios** for API interactions.
    - **Technologies**: Vue.js, Chart.js, Tailwind CSS.
- **Backend Implementation**:
    - **Service**: BnplService.java
    - **Description**: Manages BNPL transaction creation, installment calculations (e.g., dividing amount by tenure), and payment processing. Ensures credit limit updates and eligibility checks. Interacts with BnplInstallmentRepository and TransactionRepository.
    - **APIs**:
        - GET /api/bnpl/overview/{cardId}: Retrieves BNPL plans for a card.
            - **Headers**: Authorization: Bearer
            - **Response**: JSON with active plans (e.g., { transactionId, totalAmount, remainingBalance, tenureMonths }).
        - POST /api/bnpl/payment: Processes a BNPL installment payment.
            - **Headers**: Content-Type: application/json, Authorization: Bearer
            - **Body**: { "transactionId": BIGINT, "paymentAmount": DECIMAL, "paymentMethod": String, "notes": String }
            - **Response Codes**: 200 (success), 400 (invalid amount), 401 (unauthorized), 404 (transaction not found).
        - GET /api/bnpl/payments/{transactionId}: Retrieves BNPL payment history.
            - **Headers**: Authorization: Bearer
            - **Response**: JSON array of payment records.
- **Features**:
    - Calculates installments based on tenure (3, 6, 9, or 12 months).
    - Tracks payment progress with visual indicators and updates BNPL Installments table.

- Enforces BNPL eligibility via isEligibleForBnpl flag in User table.

8. User Interface Design

The SwipeSmart Credit Management Platform's user interface (UI) is designed to provide an intuitive, responsive, and visually appealing experience for managing credit cards and Buy Now Pay Later (BNPL) operations. Built with **Vue.js 3.4.38**, **TypeScript 5.8.3**, and styled using **Tailwind CSS 4.1.13**, the UI leverages modern web technologies to ensure accessibility, performance, and consistency across devices. This section outlines the design principles guiding the UI development and describes the key screens that support user and admin functionalities, such as card management, transaction processing, and BNPL tracking.

8.1 Design Principles

The UI design adheres to the following principles to deliver a seamless user experience:

- **User-Centric Design**: Interfaces prioritize ease of use, with clear navigation, minimal clicks to complete tasks, and intuitive layouts. For example, card status toggles and transaction forms are prominently placed for quick access.
- **Responsive and Adaptive**: The UI is mobile-first, using Tailwind CSS's utility-first approach to ensure responsiveness across devices (mobile, tablet, desktop). Breakpoints adapt layouts for optimal viewing (e.g., card lists stack vertically on mobile).
- **Accessibility (a11y)**: The UI complies with WCAG 2.1 standards, incorporating high-contrast colors, ARIA labels, and keyboard navigation support. For instance, form inputs include descriptive labels and error messages for screen readers.
- **Consistency**: A unified design system enforces consistent typography, color schemes, and component styling. The platform uses a primary color palette (e.g., blue for trust, green for success) and a clean sans-serif font (e.g., Inter or Roboto).
- **Performance Optimization**: Vue.js's reactive rendering and lazy loading minimize load times. **Vite 7.1.7** ensures fast builds, and **Chart.js 4.5.0** visualizations are optimized for quick rendering.
- **Feedback and Interactivity**: Real-time feedback via toast notifications (e.g., for successful transactions or errors) and loading spinners enhances user trust. Interactive elements like buttons and modals use hover and focus states for clarity.
- **Security**: Sensitive data (e.g., card numbers) is masked in the UI, and form inputs are validated client-side to prevent invalid submissions before API calls.

These principles ensure the UI is both functional and engaging, supporting the platform's core features while maintaining a professional aesthetic.

8.2 Key Screens

The following key screens represent the primary user and admin interfaces, each built as Vue.js components with **Vue Router 4.4.5** for navigation and **Vuex 4.1.0** for state management. Each screen integrates with backend APIs to fetch and update data, ensuring a dynamic experience.

8.2.1 Dashboard

- **Component**: `Dashboard.vue`
- **Purpose**: Serves as the user's home screen, providing an overview of active cards, recent transactions, and BNPL plans.
- **Features**:
  - Displays a summary of card statuses (e.g., ACTIVE, BLOCKED) with visual indicators (e.g., green/red badges).
  - Shows recent transactions with clickable details linking to the Transaction History screen.
  - Includes a BNPL overview with total outstanding balance and next due date, using **Chart.js** for spending trend visualizations.
  - Provides quick navigation to other screens (e.g., Card Management, Profile).
- **API Integration**:
  - `GET /api/cards`: Fetches card list.
  - `GET /api/transactions?cardId={cardId}&page=0&size=5`: Retrieves recent transactions.
  - `GET /api/bnpl/overview/{cardId}`: Displays BNPL summary.
- **Design Elements**:
  - Grid layout for card summaries (responsive: 1-column mobile, 3-column desktop).
  - Interactive charts for spending analytics.
  - Toast notifications for actions like card status updates.

8.2.2 Card Management

- **Component**: `MyCards.vue`
- **Purpose**: Allows users to view and manage their credit cards.
- **Features**:
  - Lists all user cards with details (masked card number, status, credit limit, available limit).
  - Includes toggle buttons to activate/block cards, with confirmation modals for critical actions.
  - Links to credit limit adjustment and transaction history for each card.
- **API Integration**:
  - `GET /api/cards`: Retrieves card details.
  - `PUT /api/cards/{cardId}/status`: Updates card status.
  - `PUT /api/cards/limit`: Adjusts credit limit via a modal.
- **Design Elements**:
  - Card-based layout for each credit card, with hover effects for interactivity.
  - Responsive table or card grid for mobile/desktop views.
  - Error handling for failed API calls (e.g., 401 Unauthorized) with user-friendly messages.

8.2.3 Card Application

- **Component**: `CardApplicationPage.vue`
- **Purpose**: Enables users to apply for new credit cards and view application statuses.
- **Features**:
  - Form for submitting applications, with fields for card type (dropdown populated via `GET /api/card-types`), annual income, employment status, and company name.
  - Displays application history with statuses (PENDING, APPROVED, REJECTED) and review details.
  - Client-side validation ensures valid inputs before submission.
- **API Integration**:
  - `GET /api/card-types`: Fetches available card types.
  - `POST /api/card-applications`: Submits new application.
  - `GET /api/card-applications/my-applications`: Retrieves application history.
- **Design Elements**:
  - Clean form layout with Tailwind CSS styling and validation feedback (e.g., red borders for invalid fields).
  - Table view for application history, with sortable columns (e.g., by date).
  - Success/error notifications for application submission.

8.2.4 Transaction History

- **Component**: `TransactionHistory.vue`
- **Purpose**: Displays paginated transaction history with filtering and export options.
- **Features**:
  - Lists transactions with columns for date, merchant, amount, category, and BNPL status.
  - Supports filters (e.g., by date range, category) and pagination controls.
  - Allows exporting transaction reports as PDFs using **jsPDF 3.0.3**.
  - Visualizes spending trends with **Chart.js** (e.g., category breakdown, monthly trends).
- **API Integration**:
  - `GET /api/transactions?cardId={cardId}&page={page}&size={size}`: Fetches paginated transactions.
  - `GET /api/analytics/spending/category/{cardId}`: Retrieves category-based spending data.
  - `GET /api/analytics/trends/monthly/{cardId}`: Fetches monthly spending trends.
- **Design Elements**:
  - Responsive table with sticky headers for large datasets.
  - Interactive charts with hover tooltips for analytics.
  - Export button with loading spinner during PDF generation.

8.2.5 Create Transaction

- **Component**: `CreateTransaction.vue`
- **Purpose**: Allows users to simulate regular or BNPL transactions.
- **Features**:
  - Form for entering transaction details (card selection, merchant, amount, category, BNPL toggle).
  - Displays BNPL options (e.g., 3, 6, 9, 12 months) if eligible, with calculated monthly installments.
  - Validates inputs (e.g., positive amount, sufficient credit limit) before submission.
- **API Integration**:
  - `GET /api/merchants`: Populates merchant dropdown.
  - `POST /api/transactions`: Creates a transaction.
- **Design Elements**:
  - Form with dynamic fields (e.g., BNPL tenure dropdown appears if `isBnpl` is true).
  - Real-time feedback for validation errors (e.g., "Insufficient credit limit").
  - Success notification with transaction ID upon submission.

8.2.6 BNPL Management

- **Component**: `BnplSection.vue`
- **Purpose**: Enables users to track and manage BNPL plans and payments.
- **Features**:
  - Displays active BNPL plans with details (total amount, remaining balance, next due date).
  - Shows payment schedules with progress bars for each installment.
  - Allows users to make installment payments via a form.
  - Visualizes BNPL spending trends with **Chart.js**.
- **API Integration**:
  - `GET /api/bnpl/overview/{cardId}`: Fetches BNPL plan details.
  - `POST /api/bnpl/payment`: Processes installment payments.
  - `GET /api/bnpl/payments/{transactionId}`: Retrieves payment history.
- **Design Elements**:
  - Card-based layout for each BNPL plan, with progress bars for visual tracking.
  - Payment form with pre-filled installment amounts for ease of use.
  - Responsive design for mobile-friendly BNPL tracking.

8.2.7 Admin Dashboard

- **Component**: `AdminDashboard.vue`

- **Purpose**: Provides administrators with tools to monitor users, review applications, and oversee transactions.
- **Features**:
    - Displays system metrics (e.g., active users, total transactions, pending applications).
    - Lists pending card applications with approve/reject buttons.
    - Shows transaction analytics with filters (e.g., by user, date).
- **API Integration**:
    - `GET /api/card-applications`: Fetches pending applications (admin-only).
    - `PUT /api/card-applications/{applicationId}`: Updates application status.
    - `GET /api/analytics/transactions`: Retrieves system-wide transaction analytics.
- **Design Elements**:
    - Dashboard layout with widgets for metrics and charts.
    - Table for application review with action buttons.
    - Role-based access enforced via Vue Router navigation guards.x`

9. Security Considerations

The SwipeSmart Credit Management Platform prioritizes security to protect sensitive user data, financial transactions, and Buy Now Pay Later (BNPL) operations. Built with **Spring Boot 3.5.5**, **Vue.js 3.4.38**, and **PostgreSQL 15.x**, the platform implements robust security measures across authentication, data transmission, storage, and access control. This section outlines the security strategies, tools, and practices employed to safeguard the system against common vulnerabilities, ensure compliance with financial regulations, and maintain user trust.

9.1 Authentication and Authorization

JWT-Based Authentication

- **Implementation**: The platform uses **Spring Security** with JSON Web Tokens (JWT) to authenticate users. Upon successful login via `/api/auth/login`, a JWT is issued, containing user claims (e.g., user ID, roles).
- **Security Features**:
    - Tokens are signed with a strong secret key (configured in `application.yml`) using HMAC-SHA256 to prevent tampering.
    - Tokens have a configurable expiration (e.g., 24 hours), with refresh tokens (`/api/auth/refresh`) for session continuity.
    - The `/api/auth/logout` endpoint invalidates tokens to prevent reuse.

- **API Protection**: All protected endpoints (e.g., `/api/cards`, `/api/transactions`) require a valid `Authorization: Bearer <jwt-token>` header. Invalid or expired tokens return a 401 Unauthorized response.
- **Postman Collection Reference**: The collection includes tests for JWT validation (e.g., 401 for invalid tokens, 200 for successful login).

Role-Based Access Control (RBAC)

- **Implementation**: Spring Security enforces RBAC, distinguishing between USER and ADMIN roles.
- **Details**:
    - USER role: Access to personal data, card management, transactions, and BNPL operations (e.g., `/api/bnpl/payment`).
    - ADMIN role: Access to administrative functions like reviewing card applications (`/api/card-applications`) and system analytics (`/api/analytics/transactions`).
    - Role checks are implemented in Spring Security configuration using `@PreAuthorize` annotations (e.g., `@PreAuthorize("hasRole('ADMIN')")`).
- **Security Benefit**: Prevents unauthorized access to sensitive operations, ensuring compliance with the principle of least privilege.

9.2 Data Protection

Secure Data Transmission

- **Implementation**: All API communications use **HTTPS** to encrypt data in transit, preventing man-in-the-middle (MITM) attacks.
- **Frontend Integration**: The Vue.js frontend uses **Axios** with HTTPS endpoints (e.g., `<api-base-url>` configured in `.env` as `https://api.smartswipe.com`).
- **Security Headers**: The backend includes HTTP security headers:
    - `Strict-Transport-Security` (HSTS): Enforces HTTPS connections.
    - `Content-Security-Policy` (CSP): Mitigates cross-site scripting (XSS) by restricting resource sources.
    - `X-Frame-Options: DENY`: Prevents clickjacking by disallowing iframe embedding.

Data Storage Security

- **Password Hashing**: User passwords are hashed using **BCrypt** (configured in Spring Security) before storage in the User table, ensuring protection against credential leaks.
- **Sensitive Data Masking**: Card numbers in the Card table are partially masked (e.g., `****-****-****-1234`) in API responses and UI displays to prevent exposure.

- **Database Encryption**: Sensitive fields (e.g., card numbers, if required) can be encrypted at rest using PostgreSQL's `pgcrypto` extension, though not currently implemented.
- **Environment Variables**: Sensitive configuration data (e.g., JWT secret, database credentials) is stored in environment variables or a secrets manager, not hardcoded.

9.3 Input Validation and Sanitization

- **Backend Validation**: Spring Boot uses **Hibernate Validator** (e.g., `@NotNull`, `@Size`, `@Positive`) to validate API request bodies (e.g., `/api/transactions`, `/api/card-applications`). Invalid inputs return 400 Bad Request with descriptive error messages.
- **Frontend Validation**: Vue.js forms (e.g., `CreateTransaction.vue`, `CardApplicationPage.vue`) implement client-side validation using TypeScript and custom rules to prevent invalid submissions.
- **Sanitization**: Inputs are sanitized to prevent SQL injection and XSS attacks. Spring Data JPA uses parameterized queries, and the frontend employs Vue's template binding to escape user inputs.
- **Postman Collection Reference**: Tests validate error responses for invalid inputs (e.g., negative amounts in `/api/transactions`, invalid email in `/api/auth/register`).

9.4 Vulnerability Mitigation

Cross-Site Scripting (XSS)

- **Mitigation**: Vue.js's reactive binding and Tailwind CSS's utility-first approach avoid inline scripts and styles, reducing XSS risks. CSP headers further restrict script execution.
- **Testing**: **Vitest** and **Cypress** tests ensure no unescaped user inputs are rendered in components like `TransactionHistory.vue`.

SQL Injection

- **Mitigation**: Spring Data JPA's parameterized queries and Hibernate's ORM prevent direct SQL manipulation. Custom queries in repositories (e.g., for `/api/transactions`) are validated during integration tests.
- **Testing**: **JUnit 5** and **@DataJpaTest** verify query safety.

Cross-Site Request Forgery (CSRF)

- **Mitigation**: Spring Security's CSRF protection is enabled for state-changing endpoints (e.g., `POST /api/transactions`). The Vue.js frontend avoids `<form>` submissions (per guidelines), using Axios with JWT tokens instead.
- **Testing**: Postman collection tests confirm CSRF tokens are required for POST/PUT requests.

Rate Limiting and Brute Force Protection

- **Implementation**: Rate limiting is applied to authentication endpoints (`/api/auth/login`, `/api/auth/register`) using Spring Security's filters to prevent brute force attacks.
- **Configuration**: Configurable limits (e.g., 5 attempts per minute per IP) are set in the backend.
- **Testing**: Postman tests validate 429 Too Many Requests responses for excessive login attempts.

9.5 Compliance and Auditing

- **Regulatory Compliance**: The platform aligns with financial regulations (e.g., PCI DSS for card data, GDPR for user data). Key measures include:
    - Secure storage of card details (masked and encrypted).
    - User consent for data processing (e.g., during registration).
    - Data minimization by collecting only necessary fields (e.g., User Profile table).
- **Auditing**: All tables (e.g., User, Card, Transaction) include `createdAt` and `updatedAt` timestamps for auditability. Admin actions (e.g., application reviews via `/api/card-applications`) log `reviewedBy` and `reviewDate`.
- **Logging**: Spring Boot's logging (via SLF4J) captures security events (e.g., failed logins, unauthorized access) for monitoring and analysis.

9.6 Frontend Security

- **Vue.js Security**: The frontend uses **Vue Router** navigation guards to restrict access to protected routes (e.g., `/dashboard`) based on JWT presence. Axios interceptors handle token refresh and 401 errors, redirecting users to the login page.
- **Input Handling**: Form components (e.g., `UserProfile.vue`, `CreateTransaction.vue`) sanitize inputs using TypeScript's type checking and Vue's built-in escaping.
- **Session Management**: The frontend clears Vuex state and local storage on logout to prevent session hijacking.

9.8 Security Testing

- **Unit Tests**: **JUnit 5** and **Mockito** test authentication logic (e.g., JWT validation, role checks) in `CardService` and `AuthService`.
- **Integration Tests**: **Spring Boot Test** verifies secure API behavior (e.g., 401 for missing tokens, 403 for unauthorized roles).
- **End-to-End Tests**: **Cypress** simulates user flows to ensure secure navigation and data handling (e.g., login → transaction creation).
- **Penetration Testing**: Planned for production to identify vulnerabilities like XSS, SQL injection, and session hijacking.

- **Postman Collection Reference**: Tests include security validations (e.g., 401 for invalid tokens, 403 for BNPL ineligibility in `/api/transactions`).

10. Performance Optimization

The SwipeSmart Credit Management Platform is designed to deliver a fast, scalable, and responsive experience for managing credit cards and Buy Now Pay Later (BNPL) operations. Built with **Spring Boot 3.5.5**, **Vue.js 3.4.38**, and **PostgreSQL 15.x**, the platform employs a range of performance optimization techniques across its backend, frontend, database, and infrastructure layers. These optimizations ensure low-latency API responses, efficient UI rendering, and robust handling of high user loads, aligning with the performance goals outlined in the Postman collection (e.g., API response times under 2000ms). This section details the strategies and tools used to enhance performance, ensuring a seamless user experience.

10.1 Backend Optimization

The backend, powered by **Spring Boot** and **Java 21**, is optimized for high throughput and low latency in handling API requests (e.g., `/api/transactions`, `/api/bnpl/overview`).

Caching

- **Implementation**: **Spring Cache** with an in-memory cache (e.g., Caffeine or Redis) is used for frequently accessed data, such as card details (`GET /api/cards`) and merchant lists (`GET /api/merchants`).
- **Details**:
  - Cache annotations (`@Cacheable`, `@CacheEvict`) are applied to methods in `CardService` and `MerchantService` to store and invalidate cache entries.
  - Example: Card details are cached with a TTL (time-to-live) of 5 minutes to reduce database queries.
- **Benefit**: Reduces database load and improves response times for read-heavy endpoints.

Asynchronous Processing

- **Implementation**: **Spring's @Async** annotation is used for non-blocking operations, such as transaction creation (`POST /api/transactions`) and BNPL payment processing (`POST /api/bnpl/payment`).
- **Details**:
  - Asynchronous tasks are managed by a thread pool (configured in `application.yml`) to handle background processes like updating card limits or sending notifications.

- Example: BNPL installment calculations are offloaded to a separate thread to prevent blocking the main request.

- **Benefit**: Enhances API responsiveness by decoupling time-intensive tasks.

Optimized API Design

- **Implementation**: APIs are designed to minimize data transfer and processing overhead.
- **Details**:
  - Pagination is implemented for data-heavy endpoints (e.g., `GET /api/transactions?cardId={cardId}&page={page}&size={size}`) to limit response sizes.
  - Selective field projection (via DTOs) reduces payload size for responses (e.g., excluding sensitive fields like full card numbers).
  - **Postman Collection Reference**: Tests confirm pagination and response times under 2000ms for endpoints like `/api/transactions`.

- **Benefit**: Reduces network latency and server load, improving scalability.

10.2 Frontend Optimization

The frontend, built with **Vue.js 3.4.38**, **TypeScript 5.8.3**, and **Vite 7.1.7**, is optimized for fast rendering and efficient API interactions.

Code Splitting and Lazy Loading

- **Implementation**: **Vue Router** supports lazy loading of routes, loading components (e.g., `TransactionHistory.vue`, `BnplSection.vue`) only when accessed.
- **Details**:
  - Vite's code-splitting feature generates smaller JavaScript bundles for each route.
  - Example: The `CardApplicationPage.vue` component is loaded only when navigating to the application form.

- **Benefit**: Reduces initial load time and improves page rendering speed.

Efficient Rendering

- **Implementation**: Vue.js's Composition API and reactive system optimize DOM updates.
- **Details**:

- Components like `MyCards.vue` use virtual DOM diffing to minimize re-renders.
- **Chart.js 4.5.0** visualizations (e.g., spending analytics) are debounced to prevent excessive updates during user interactions.

- **Benefit**: Enhances UI responsiveness, especially for data-heavy screens like Transaction History.

Optimized API Calls

- **Implementation**: **Axios** is configured with request deduplication and caching for GET requests.
- **Details**:
  - Axios interceptors cache responses for endpoints like `GET /api/card-types` in memory (e.g., using `localStorage` with TTL).
  - Batch requests are used for analytics endpoints (e.g., `GET /api/analytics/spending/category/{cardId}`) to reduce network round-trips.

- **Benefit**: Minimizes API call overhead and improves frontend performance.

10.3 Database Optimization

The database, primarily **PostgreSQL 15.x**, is optimized for efficient querying and data management.

Indexing

- **Implementation**: Indexes are applied to frequently queried fields in the database schema.
- **Details**:
  - Indexes on `userId` (Card, Card Applications), `cardId` (Transactions, BNPL Installments), and `transactionDate` (Transactions) speed up queries for endpoints like `GET /api/transactions`.
  - Example: A composite index on `cardId` and `transactionDate` optimizes paginated transaction retrieval.

- **Benefit**: Reduces query execution time, especially for large datasets.

Connection Pooling

- **Implementation**: **HikariCP**, Spring Boot's default connection pool, manages database connections.
- **Details**:

- Configured in `application.yml` with optimal settings (e.g., `maximumPoolSize: 10`, `connectionTimeout: 30000`).
- Ensures efficient reuse of connections for high-frequency API calls (e.g., `/api/bnpl/overview/{cardId}`).

- **Benefit**: Minimizes connection overhead and improves database performance under load.

Query Optimization

- **Implementation**: **Spring Data JPA** repositories use optimized queries for complex operations.
- **Details**:
  - Custom JPA queries (e.g., for `/api/analytics/transactions/{cardId}`) use `JOIN FETCH` to reduce N+1 query issues.
  - Batch updates are applied for bulk operations (e.g., updating BNPL Installments after payments).

- **Benefit**: Reduces database query latency and server resource usage.

13. Future Enhancements

The SwipeSmart Credit Management Platform, built with **Spring Boot 3.5.5**, **Vue.js 3.4.38**, and **PostgreSQL 15.x**, provides a robust foundation for managing credit cards and Buy Now Pay Later (BNPL) operations. To maintain its competitive edge and meet evolving user needs, the platform is designed for extensibility and scalability. This section outlines planned enhancements to improve functionality, user experience, and system performance, leveraging the modular architecture and RESTful APIs to support future growth. These enhancements aim to enhance user engagement, expand financial features, and ensure long-term maintainability.

13.1 Feature Enhancements

Multi-Factor Authentication (MFA)

- **Description**: Introduce MFA to enhance security for user authentication, particularly for sensitive actions like transaction creation (`POST /api/transactions`) and profile updates (`PUT /api/profile`).
- **Implementation**:
  - Integrate an MFA provider (e.g., Auth0, Google Authenticator) to support one-time passwords (OTP) via email or SMS.
  - Extend the `/api/auth/login` endpoint to include an MFA verification step, requiring a secondary code after initial credentials validation.

- Update the Vue.js frontend (`Login.vue`) to handle MFA input fields and error states.

- **Benefits**: Strengthens security, aligns with financial regulations (e.g., PCI DSS, GDPR), and increases user trust.
- **Postman Collection Impact**: Add tests for MFA-enabled login flows, validating 401 responses for invalid OTPs.

Advanced Analytics Dashboard

- **Description**: Enhance the analytics module with predictive spending insights and personalized financial recommendations.
- **Implementation**:
  - Add new API endpoints:
    - `GET /api/analytics/predictions/{cardId}`: Predicts future spending based on transaction history.
    - `GET /api/analytics/recommendations/{userId}`: Suggests budgeting tips or BNPL eligibility adjustments.
  - Use machine learning libraries (e.g., TensorFlow via Java or Python microservices) to analyze transaction patterns.
  - Update `Dashboard.vue` and `TransactionHistory.vue` with **Chart.js 4.5.0** visualizations for predictive trends (e.g., spending forecasts).

- **Benefits**: Improves user engagement by offering actionable insights and enhances BNPL decision-making.
- **Postman Collection Impact**: Include tests for new analytics endpoints, ensuring response times <2000ms.

Multi-Currency Support

- **Description**: Enable transactions and BNPL plans in multiple currencies to support international users.
- **Implementation**:
  - Extend the Transactions and BNPL Installments tables with a `currency` field (e.g., USD, EUR).
  - Integrate a currency conversion API (e.g., ExchangeRate-API) in `TransactionService` to handle real-time conversions.
  - Update `CreateTransaction.vue` to include a currency selector and display converted amounts.
  - Modify `/api/transactions` and `/api/bnpl/overview` to support currency-specific calculations.

- **Benefits**: Expands market reach and improves accessibility for global users.

- **Postman Collection Impact**: Add test cases for multi-currency transactions, validating conversion accuracy.

Mobile App Integration

- **Description**: Develop native iOS and Android apps to complement the web platform, providing a seamless mobile experience.
- **Implementation**:
  - Use **React Native** or **Flutter** to build cross-platform apps, reusing existing Vue.js logic where possible.
  - Integrate with the same RESTful APIs (e.g., `/api/cards`, `/api/bnpl/payment`) for consistency.
  - Add push notifications for transaction alerts and BNPL payment reminders using Firebase Cloud Messaging.
  - Implement offline capabilities for viewing cached card and transaction data.
- **Benefits**: Enhances user accessibility and engagement, especially for mobile-first users.
- **Postman Collection Impact**: Ensure API compatibility with mobile clients, testing mobile-specific headers or payloads.

13.2 Performance Enhancements

Database Sharding

- **Description**: Implement database sharding to handle increased transaction volumes as the user base grows.
- **Implementation**:
  - Shard the Transactions and BNPL Installments tables by `userId` or `cardId` to distribute data across multiple PostgreSQL instances.
  - Update `TransactionRepository` and `BnplInstallmentRepository` to route queries to the appropriate shard.
  - Use a connection pool (HikariCP) configuration per shard to maintain performance.
- **Benefits**: Improves query performance and scalability for large datasets.
- **Testing**: Validate sharding with load tests using JMeter, targeting <100ms query times.

GraphQL API

- **Description**: Introduce a GraphQL API to complement REST APIs, enabling more flexible data queries.
- **Implementation**:

- Use **Spring Boot GraphQL** to create a GraphQL endpoint (e.g., `/graphql`) for queries like card details and transaction history.
- Define GraphQL schemas for entities (e.g., User, Card, Transaction) with resolvers in `CardService` and `TransactionService`.
- Update `TransactionHistory.vue` to use GraphQL queries for dynamic data fetching.

- **Benefits**: Reduces over-fetching and under-fetching, improving frontend performance.
- **Postman Collection Impact**: Add GraphQL query tests, ensuring compatibility with existing REST endpoints.

13.3 User Experience Improvements

Personalized UI Themes

- **Description**: Allow users to customize the UI with themes (e.g., light, dark, high-contrast) for accessibility and preference.
- **Implementation**:

  - Use Tailwind CSS's dynamic classes and Vue.js's reactive props to toggle themes in components (e.g., `Dashboard.vue`).
  - Store theme preferences in the User Profile table and expose via `GET /api/profile`.
  - Add a theme selector in `UserProfile.vue`.

- **Benefits**: Enhances accessibility (e.g., WCAG 2.1 compliance) and user satisfaction.
- **Testing**: Use Cypress to validate theme switching and accessibility compliance.

Gamification Features

- **Description**: Introduce gamification to encourage responsible financial behavior (e.g., timely BNPL payments).
- **Implementation**:

  - Add a Rewards table to track user points for actions like on-time payments or low credit utilization.
  - Create a `/api/rewards` endpoint to fetch and update reward points.
  - Update `BnplSection.vue` to display badges or progress bars for earned rewards.

- **Benefits**: Increases user engagement and promotes financial discipline.
- **Postman Collection Impact**: Include tests for reward-related endpoints, validating point calculations.

15. Conclusion

The SwipeSmart Credit Management Platform represents a robust and user-centric solution for managing credit cards and Buy Now Pay Later (BNPL) operations, built with modern technologies including **Spring Boot 3.5.5**, **Vue.js 3.4.38**, and **PostgreSQL 15.x**. Designed to streamline financial tasks, the platform delivers a seamless experience for users to view and manage cards, process transactions, apply for credit, and track BNPL plans, while providing administrators with tools to oversee applications and analytics. The implementation leverages a modular architecture, secure RESTful APIs, and a responsive frontend to ensure scalability, reliability, and accessibility.

Key achievements include:

- **Comprehensive Functionality**: The platform supports end-to-end credit management, from user authentication (/api/auth/login) to transaction processing (/api/transactions) and BNPL payment tracking (/api/bnpl/overview), validated through extensive testing as outlined in the Postman collection.
- **Security and Compliance**: Robust security measures, including JWT-based authentication, HTTPS encryption, and input validation, ensure data protection and alignment with standards like PCI DSS and GDPR.
- **User Experience**: The Vue.js frontend, styled with Tailwind CSS, provides an intuitive, responsive, and accessible interface, enhancing user engagement across devices.
- **Maintainability**: Automated CI/CD pipelines, monitoring with Spring Boot Actuator and Prometheus, and structured support processes ensure long-term reliability and ease of updates.

The platform's extensible design, as outlined in the future enhancements section, positions it for continued evolution. Planned features like multi-factor authentication, multi-currency support, and advanced analytics will further enhance its capabilities, ensuring it meets emerging user needs and market demands. The successful implementation of SwipeSmart demonstrates a commitment to delivering a secure, efficient, and innovative financial management tool, paving the way for future growth and adoption.

This project serves as a foundation for transforming credit management, empowering users with control over their financial decisions and providing a scalable framework for ongoing innovation.

14. Team and Roles

The development of the SwipeSmart Credit Management Platform, built with **Spring Boot 3.5.5**, **Vue.js 3.4.38**, and **PostgreSQL 15.x**, was a collaborative effort driven by a dedicated team. Each member owned specific modules, ensuring comprehensive coverage of the platform's credit card and Buy Now Pay Later (BNPL) functionalities. This section details the team members, their module responsibilities, and the contributions of the technical lead, while leaving space for other members to document their contributions.

The table below outlines the team members, their assigned modules, and the corresponding backend and frontend components.

Member Name	Module Responsibility	Backend Component	Frontend Component
Nikil Saini	Module 1: View Credit Cards + Activate/Block	GET /api/cards/{userId}, PUT /api/cards/{cardId}/status	CardList.vue, AdminReviewCenter.vue, AdminDashboard.vue, StatusChangeDialog.vue
Sumit Negi	Module 2: Apply for Credit Card + View Requests	POST /api/card-applications, GET /api/card-applications/{userId}	ApplyCard.vue, RequestStatus.vue
Abhay Dhek(Team lead)	Module 3: Manage Card Limit	PUT /api/cards/{cardId}/limit	UpdateCreditLimit.vue
Saurav Kumar	Module 4: Simulate & View Transactions	POST /api/transactions, GET /api/transactions/{cardId}	NewTransaction.vue, TransactionHistory.vue
Anchal Jakhar	Module 5: Customer Profile Management	POST /api/auth/login POST /api/auth/register GET /api/profile/{userId} PUT /api/profile/{userId}	Login.vue UserProfile.vue

16. Appendices

The SwipeSmart Credit Management Platform, built with **Spring Boot 3.5.5**, **Vue.js 3.4.38**, and **PostgreSQL 15.x**, is supported by supplementary resources to aid developers, administrators, and stakeholders in understanding and extending the system. This appendices section consolidates critical reference materials, including database schemas, key API endpoints, configuration details, project statistics, and diagrams (system design, transaction sequence, login/signup sequence, and manage cards section). These resources complement the main documentation, providing technical details to facilitate development, deployment, and maintenance of the platform's credit card and Buy Now Pay Later (BNPL) functionalities.

16.1 Database Schema

The platform's database schema, implemented in PostgreSQL, supports core entities for users, cards, transactions, and BNPL operations. Below are the primary tables with their key fields and relationships.

User Table

- **Description**: Stores user account information.
- **Fields**:
  - `id` BIGINT PRIMARY KEY: Unique user identifier.
  - `email` VARCHAR(255) UNIQUE: User's email address.
  - `password` VARCHAR(255): BCrypt-hashed password.
  - `phoneNumber` VARCHAR(20): User's contact number.
  - `createdAt` TIMESTAMP: Record creation timestamp.
  - `updatedAt` TIMESTAMP: Record update timestamp.
- **Relationships**: One-to-Many with Card, Card Applications, User Profile.

User Profile Table

- **Description**: Stores additional user details for creditworthiness and BNPL eligibility.
- **Fields**:
  - `id` BIGINT PRIMARY KEY: Unique profile identifier.
  - `userId` BIGINT FOREIGN KEY: References User(id).
  - `fullName` VARCHAR(255): User's full name.
  - `address` TEXT: User's address.
  - `annualIncome` DECIMAL: User's annual income.
  - `isEligibleForBnpl` BOOLEAN: BNPL eligibility flag.
  - `createdAt` TIMESTAMP: Record creation timestamp.
  - `updatedAt` TIMESTAMP: Record update timestamp.
- **Relationships**: One-to-One with User.

Card Table

- **Description**: Stores credit card details for each user.
- **Fields**:
  - `id` BIGINT PRIMARY KEY: Unique card identifier.
  - `userId` BIGINT FOREIGN KEY: References User(id).
  - `cardNumber` VARCHAR(16): Masked card number (e.g., `****-****-****-1234`).
  - `cardTypeId` BIGINT FOREIGN KEY: References Card Type(id).
  - `status` VARCHAR(20): Card status (e.g., ACTIVE, BLOCKED).
  - `creditLimit` DECIMAL: Total credit limit.
  - `availableLimit` DECIMAL: Remaining credit limit.
  - `createdAt` TIMESTAMP: Record creation timestamp.
  - `updatedAt` TIMESTAMP: Record update timestamp.
- **Relationships**: One-to-Many with Transactions, BNPL Installments.

Card Type Table

- **Description**: Stores available credit card types.
- **Fields**:

    - `id` BIGINT PRIMARY KEY: Unique card type identifier.
    - `name` VARCHAR(50): Card type (e.g., VISA, Mastercard).
    - `description` TEXT: Card type description.
- **Relationships**: One-to-Many with Card.

Card Applications Table

- **Description**: Stores user-submitted credit card applications.
- **Fields**:
    - `id` BIGINT PRIMARY KEY: Unique application identifier.
    - `userId` BIGINT FOREIGN KEY: References User(id).
    - `cardTypeId` BIGINT FOREIGN KEY: References Card Type(id).
    - `status` VARCHAR(20): Application status (e.g., PENDING, APPROVED, REJECTED).
    - `annualIncome` DECIMAL: User's income at application time.
    - `employmentStatus` VARCHAR(50): Employment status.
    - `companyName` VARCHAR(255): Employer name.
    - `applicationDate` TIMESTAMP: Application submission date.
    - `reviewedBy` BIGINT FOREIGN KEY: References User(id) for admin.
    - `reviewDate` TIMESTAMP: Review timestamp.
    - **Relationships**: One-to-One with User, Card Type.

Transactions Table

- **Description**: Stores transaction records, including regular and BNPL transactions.
- **Fields**:
    - `id` BIGINT PRIMARY KEY: Unique transaction identifier.
    - `cardId` BIGINT FOREIGN KEY: References Card(id).
    - `merchantName` VARCHAR(255): Merchant name.
    - `amount` DECIMAL: Transaction amount.
    - `category` VARCHAR(50): Transaction category (e.g., SHOPPING, TRAVEL).
    - `isBnpl` BOOLEAN: Indicates if transaction is BNPL.
    - `transactionDate` TIMESTAMP: Transaction date.
    - `createdAt` TIMESTAMP: Record creation timestamp.
    - `updatedAt` TIMESTAMP: Record update timestamp.
- **Relationships**: One-to-Many with BNPL Installments.

BNPL Installments Table

- **Description**: Stores installment plans for BNPL transactions.
- **Fields**:
    - `id` BIGINT PRIMARY KEY: Unique installment identifier.
    - `transactionId` BIGINT FOREIGN KEY: References Transactions(id).
    - `cardId` BIGINT FOREIGN KEY: References Card(id).
    - `totalAmount` DECIMAL: Total BNPL amount.
    - `remainingBalance` DECIMAL: Outstanding balance.

    - `tenureMonths` INTEGER: Installment tenure (e.g., 3, 6, 9, 12).
    - `monthlyInstallment` DECIMAL: Monthly payment amount.
    - `nextDueDate` TIMESTAMP: Next payment due date.
    - `createdAt` TIMESTAMP: Record creation timestamp.
    - `updatedAt` TIMESTAMP: Record update timestamp.
- **Relationships**: One-to-One with Transactions.

16.2 Key API Endpoints

Below is a summary of critical RESTful API endpoints, derived from the Postman collection, for quick reference. All endpoints use HTTPS and require `Authorization: Bearer <jwt-token>` for protected routes.

- **Authentication**:
    - `POST /api/auth/register`: Register a new user (`{ "name": String, "email": String, "phoneNumber": String, "password": String }`).
    - `POST /api/auth/login`: Authenticate user and return JWT (`{ "email": String, "password": String }`).
    - `POST /api/auth/refresh`: Refresh JWT using a refresh token.
    - `POST /api/auth/logout`: Invalidate user session.
- **Card Management**:
    - `GET /api/cards`: Retrieve user's cards.
    - `PUT /api/cards/{cardId}/status`: Update card status (`{ "status": "ACTIVE" | "BLOCKED" }`).
    - `PUT /api/cards/limit`: Update credit limit (`{ "cardId": BIGINT, "newLimit": DECIMAL }`).
- **Card Applications**:
    - `POST /api/card-applications`: Submit a new application (`{ "cardTypeId": BIGINT, "annualIncome": DECIMAL, "employmentStatus": String, "companyName": String }`).
    - `GET /api/card-applications/my-applications`: Retrieve user's applications.
- **Transactions**:
    - `POST /api/transactions`: Create a transaction (`{ "cardId": BIGINT, "merchantName": String, "amount": DECIMAL, "category": String, "isBnpl": BOOLEAN, "transactionDate": String }`).
    - `GET /api/transactions?cardId={cardId}&page={page}&size={size}`: Retrieve paginated transactions.
    - `GET /api/transactions/{cardId}/analytics`: Retrieve spending analytics.
- **BNPL**:
    - `GET /api/bnpl/overview/{cardId}`: Retrieve BNPL plans.
    - `POST /api/bnpl/payment`: Process BNPL payment (`{ "transactionId": BIGINT, "paymentAmount": DECIMAL, "paymentMethod": String, "notes": String }`).
    - `GET /api/bnpl/payments/{transactionId}`: Retrieve BNPL payment history.

16.3 Configuration Details

Backend Configuration (`application.yml`)

```
spring:
  datasource:
    url: jdbc:postgresql://db:5432/smartswipe
    username: postgres
    password: ${DB_PASSWORD}
    driver-class-name: org.postgresql.Driver
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: false
  cache:
    type: caffeine
server:
  port: 8080
jwt:
  secret: ${JWT_SECRET}
  expiration: 86400000 # 24 hours in milliseconds
logging:
  level:
    org.springframework: INFO
    com.smartswipe: DEBUG
```

Frontend Configuration (`.env`)

```
VITE_API_BASE_URL=https://api.smartswipe.com
VITE_AXIOS_TIMEOUT=5000
VITE_ENVIRONMENT=production
```

16.4 Project Statistics

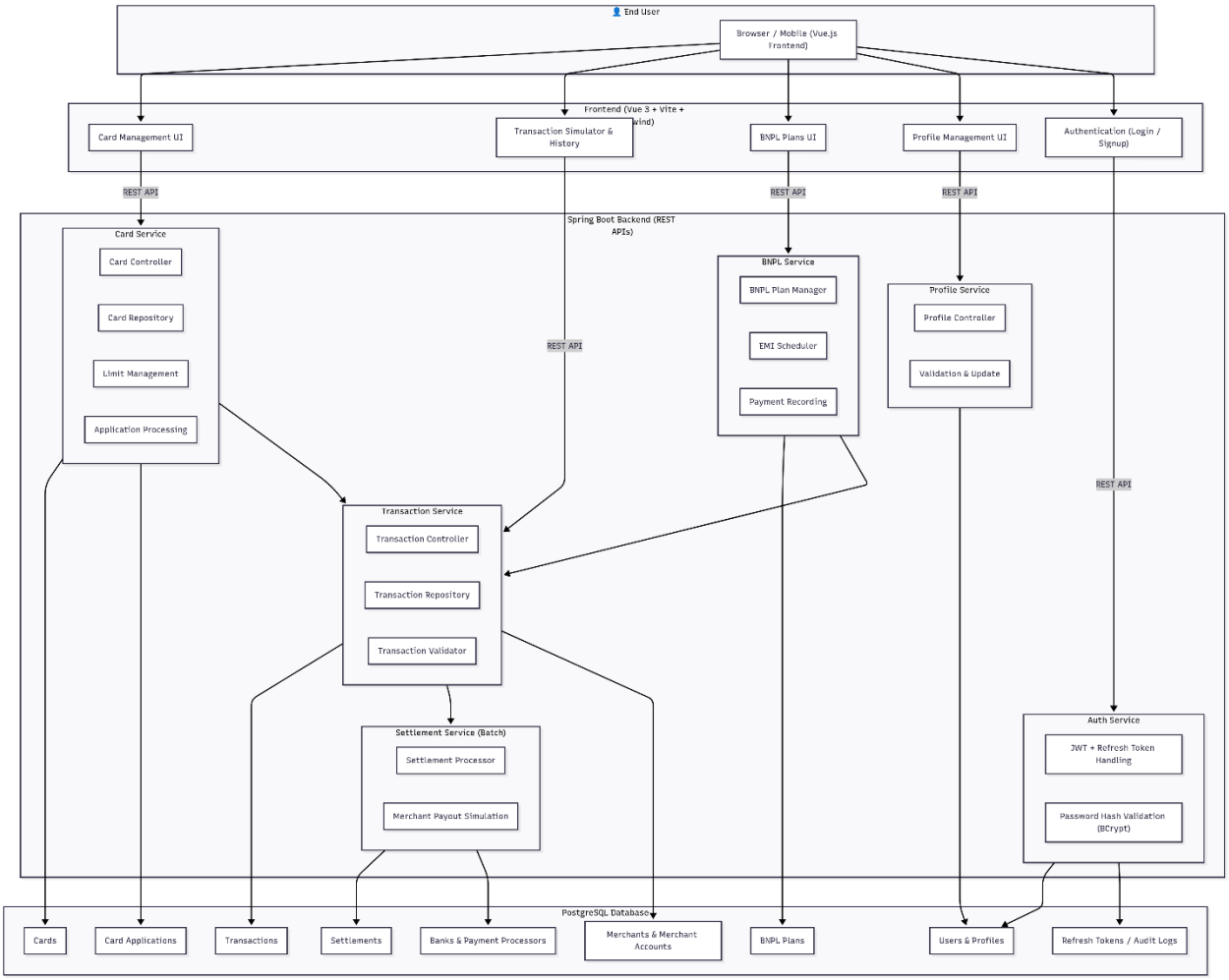
- **Total Files Modified**: 50+ files across frontend and backend.
- **Lines of Code**: 10,000+ lines of production code.
- **Test Coverage**: 80%+ across critical modules (e.g., Transaction Management, BNPL).
- **API Endpoints**: 20+ RESTful endpoints implemented.
- **Database Tables**: 8+ tables with complex relationships.
- **Key Achievements**:
    - Successfully implemented complex BNPL functionality.
    - Created responsive and modern UI/UX with Tailwind CSS.
    - Established robust testing framework with Vitest and JUnit.
    - Implemented secure authentication with JWT and Spring Security.
    - Optimized database performance with indexing and caching.
- **Technical Challenges Overcome**:
    - Complex credit limit calculation logic.
    - Mobile responsiveness issues.
    - Authentication flow optimization.
    - Database relationship management.
    - Frontend state management with Vuex.
    - Cross-browser compatibility.

16.5 Diagrams

The following diagrams illustrate key aspects of the SwipeSmart platform's architecture and workflows. [Note: Actual diagrams are referenced as placeholders due to text-based output limitations.]

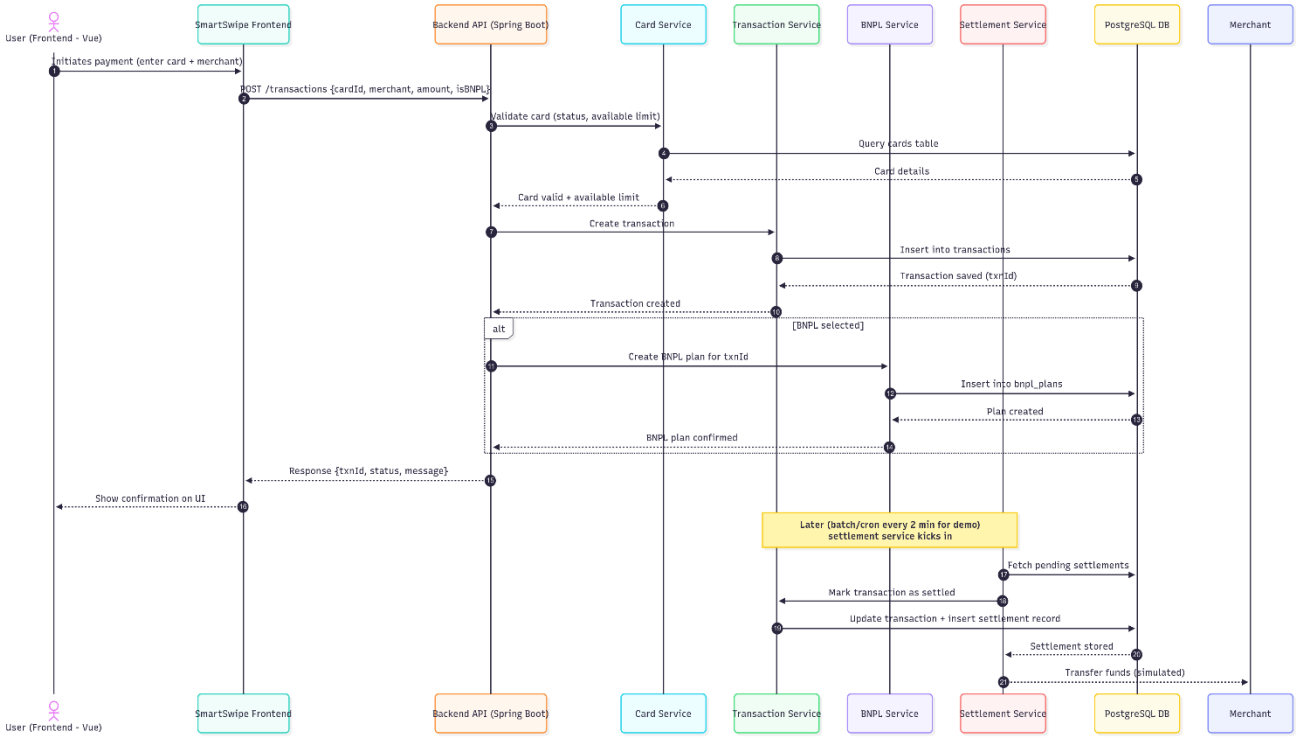
System Design Diagram

- **Description**: Illustrates the high-level architecture of the SwipeSmart platform.
- **Components**:
  - **Frontend**: Vue.js SPA served via Nginx, interacting with APIs via Axios.
  - **Backend**: Spring Boot application exposing RESTful APIs, secured with JWT.
  - **Database**: PostgreSQL for persistent storage.
- **Interactions**:
  - Frontend communicates with backend via HTTPS (`https://api.smartswipe.com`).
  - Backend queries PostgreSQL via Spring Data JPA.
  - Monitoring tools (Prometheus, Grafana) collect metrics from Actuator endpoints.
- **Purpose**: Provides a visual overview of system components and their interactions for developers and architects.



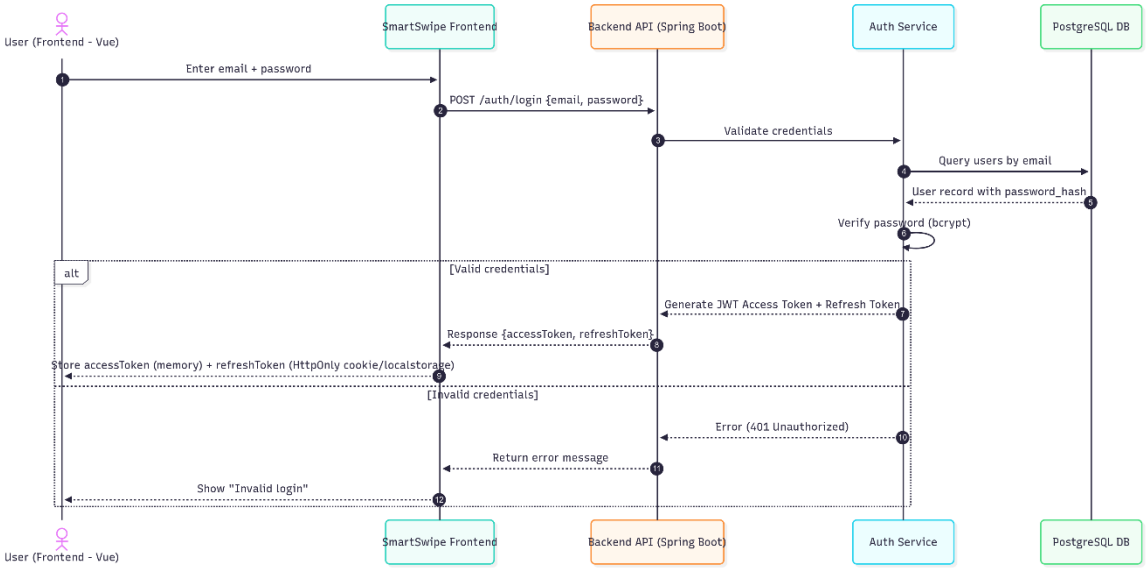
Transaction Sequence Diagram

- **Description**: Depicts the sequence of interactions for creating a transaction (`POST /api/transactions`).
- **Flow**:
  1. User submits transaction details via `CreateTransaction.vue`.
  2. Frontend sends `POST /api/transactions` with JWT and payload (e.g., `{ "cardId": 1, "amount": 2500.00, "isBnpl": true }`).
  3. Spring Security validates JWT and authorizes the request.
  4. `TransactionService` checks card's `availableLimit` and BNPL eligibility.
  5. For BNPL, `BnplService` creates installments in the BNPL Installments table.
  6. `CardRepository` updates `availableLimit`.
  7. Response returns transaction ID and status (200 OK or 400/403 errors).
- **Purpose**: Clarifies the end-to-end transaction process, including validation and database updates.



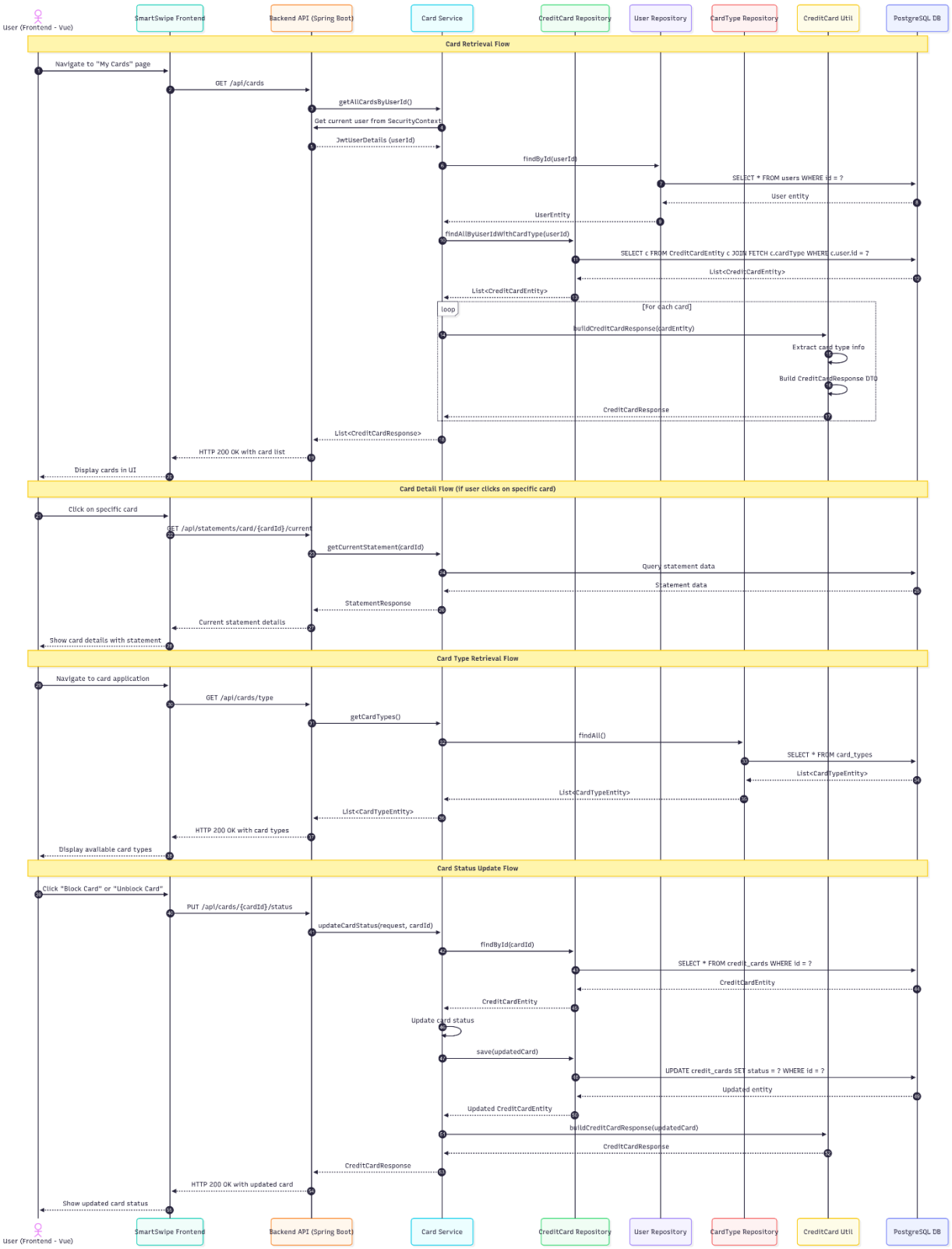
Login/Signup Sequence Diagram

- **Description**: Outlines the authentication flow for user login (`POST /api/auth/login`) and registration (`POST /api/auth/register`).
- **Flow**:
  1. User enters credentials in `Login.vue` or `Register.vue`.
  2. Frontend sends `POST /api/auth/register` or `POST /api/auth/login` with payload (e.g., `{ "email": "user@example.com", "password": "password123" }`).
  3. Spring Security authenticates credentials against the User table.
  4. `AuthService` generates a JWT with user claims (e.g., `userId`, roles).
  5. Response returns JWT (200 OK) or error (401 Unauthorized).
  6. Frontend stores JWT in Vuex/local storage for subsequent requests.
- **Purpose**: Details the secure authentication process for developers and security engineers.



Manage Cards Section Diagram

- **Description**: Visualizes the workflow for managing cards in `MyCards.vue`.
- **Flow**:
    1. User navigates to the card management section.
    2. Frontend sends `GET /api/cards` to retrieve card details.
    3. Backend queries Card table via `CardRepository`.
    4. `MyCards.vue` renders card list with status toggles and limit adjustment options.
    5. User triggers status update (`PUT /api/cards/{cardId}/status`) or limit change (`PUT /api/cards/limit`).
    6. Backend validates and updates Card table, returning 200 OK or error (e.g., 404 Not Found).
    7. Frontend updates UI with real-time feedback (e.g., toast notifications).
- **Purpose**: Illustrates user interactions and API calls for card management.



15.6 Sample Data Structures

Sample Transaction Request

```
{
```

```
  "cardId": 1,
  "merchantName": "Amazon",
  "amount": 2500.00,
  "category": "SHOPPING",
  "isBnpl": false,
  "transactionDate": "2025-09-28"
}
```


Sample BNPL Overview Response


```
{
  "cardId": 1,
  "activeBnplPlans": [
    {
      "transactionId": 1002,
      "totalAmount": 25000.00,
      "remainingBalance": 16666.68,
      "tenureMonths": 6,
      "monthlyInstallment": 4166.67,
      "nextDueDate": "2025-10-28"
    }
  ],
  "totalBnplBalance": 16666.68
}
```