SAHYADRI
COLLEGE OF ENGINEERING & MANAGEMENT
An Autonomous Institution
MANGALURU

Mini Project

Database Management System Lab – CS5227L5C

"Loan Management System"

Name and USN:

1.	Karthik Acharya	4SF23CS082
2.	M Rohith	4SF23CS094

Branch/Section: CSE / 5A

Faculty Incharge:

Ms. Chaithra S

Assistant Professor

Department of Computer Science & Engineering

SAHYADRI
COLLEGE OF ENGINEERING & MANAGEMENT
An Autonomous Institution
MANGALURU

Department of Computer Science & Engineering

CERTIFICATE

This is to certify that the mini project work entitled **"Loan Management System"** has been carried out by **Karthik Acharya (4SF23CS082) and M Rohith (4SF23CS094)** the bonafide students of Sahyadri College of Engineering & Management in partial fulfillment of the requirements for the V semester of Bachelor of Engineering in Computer Science and Engineering of Visvesvaraya Technological University, Belagavi during the year 2025 - 26. It is certified that all suggestions indicated for Internal Assessment have been incorporated in the report deposited in the departmental library. The project report has been approved as it satisfies the academic requirements in respect of project work prescribed for the said degree.

Course Coordinator **Ms. Chaithra S** Assistant Professor	**Dr. Mustafa Basthikodi** Professor & Head of Department

Name and Signature with Date

Examiner 1: _____

Examiner 2: _____

SAHYADRI
COLLEGE OF ENGINEERING & MANAGEMENT
An Autonomous Institution
MANGALURU

Department of Computer Science & Engineering

DECLARATION

We hereby declare that the entire work embodied in this Mini Project Report titled **"Loan Management System"** has been carried out by us at Sahyadri College of Engineering & Management, Mangaluru under the supervision of **Ms. Chaitra S,** in partial fulfillment of the requirements for the V semester of **Bachelor of Engineering** in **Computer Science and Engineering**. This report has not been submitted to this or any other University for the award of any other degree.

Karthik Acharya (4SF23CS082)

M Rohith (4SF23CS094)

ABSTRACT

This project presents a comprehensive Loan Management System built on a robust PostgreSQL database foundation, designed to streamline and automate the complete loan lifecycle from application to repayment. The system addresses critical challenges in financial institutions, including loan origination, borrower management, collateral tracking, repayment scheduling, and compliance auditing. By implementing advanced database features such as stored procedures, triggers, and optimized views, the platform ensures transactional integrity, data consistency, and operational efficiency.

Built using PostgreSQL as the database engine, the system leverages ACID compliance, row-level locking mechanisms, and comprehensive indexing strategies to handle concurrent operations reliably. The backend is powered by FastAPI, providing high-performance RESTful APIs with automatic documentation, while the frontend utilizes React with Vite for a responsive and modern user interface. The architecture employs SQLAlchemy as the ORM layer and Alembic for version-controlled database migrations, ensuring maintainable and scalable code.

The system implements sophisticated database patterns including role-based access control through ENUM types, automated business logic enforcement via triggers, and complex transaction handling through stored procedures. With features like automatic loan status updates, active loan limit enforcement, and real-time borrower portfolio analysis, the platform supports data-driven decision-making while maintaining strict compliance and audit trails.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1 Objectives

The primary objectives of this Loan Management System are:

1. Design Robust Schema: Create a comprehensive, well-defined relational schema that accurately models the loan management domain, including users, borrowers, loans, collateral, repayments, and audit trails.
2. Ensure Transactional Integrity: Implement stored procedures with row-level locking (SELECT FOR UPDATE) to guarantee ACID compliance and prevent race conditions during critical operations like loan disbursement and repayment processing.
3. Maintain Real-Time Consistency: Utilize database triggers for automatic business rule enforcement, including loan status updates and active loan limit restrictions.
4. Achieve Normalization: Structure all tables to adhere to Third Normal Form (3NF), eliminating redundancy and ensuring data integrity through proper foreign key relationships.
5. Enable Efficient Retrieval: Optimize database queries through strategic indexing and materialized views for application functions such as borrower portfolio analysis, overdue tracking, and loan performance metrics.
6. Implement Role-Based Security: Establish a multi-tier authentication system with distinct roles (admin, loan_officer, accountant) to control access and maintain operational security.

1.2 PostgreSQL

PostgreSQL serves as the robust relational backbone for the system, ensuring strict data integrity and consistency across all financial records through advanced features like ACID-compliant transactions and custom ENUM types. It employs strategic indexing and row-level locking to optimize performance and prevent concurrency conflicts during critical operations like loan approvals. Furthermore, the use of materialized views facilitates efficient real-time reporting, making PostgreSQL an ideal choice for handling the complex relationships, aggregations, and rigorous audit trails required in financial management.

1.3 FastAPI

FastAPI serves as a high-performance backend API layer bridging the React frontend and PostgreSQL database, leveraging Starlette and Pydantic to deliver speed comparable to Node.js while maintaining Python's simplicity. It ensures robust system architecture through automatic OpenAPI documentation, strict type safety for request validation, and clean dependency injection for secure resource management. Furthermore, its seamless integration with SQLAlchemy and native asynchronous capabilities enables the efficient handling of complex database queries and concurrent loan processing requests, guaranteeing both scalability and data integrity.

1.4 React

React powers the system's frontend interface, providing a highly responsive and efficient environment for users through its modular component architecture and real-time state management. Enhanced by Vite for optimized performance and React Router for seamless navigation, it ensures a fluid user experience that synchronizes instantly with the backend API without requiring page reloads. Additionally, the integration of Recharts transforms complex financial data into interactive visualizations, making it easier for administrators to analyze loan performance and repayment trends within a unified, maintainable dashboard.

CHAPTER 2
SYSTEM ANALYSIS

2.1 Existing System

Traditional loan management systems often suffer from several critical limitations:

- Manual Processing Bottlenecks: Loan approval workflows require multiple manual interventions, leading to processing delays and human error in calculations and documentation.
- Limited Concurrent Operations: Lack of proper transaction handling results in race conditions where multiple loan officers could potentially approve conflicting loans or update borrower records simultaneously.
- Poor Data Normalization: Redundant storage of borrower information across multiple loan records leads to update anomalies and inconsistent data.
- Inadequate Audit Trails: Insufficient logging mechanisms make it difficult to track who performed what actions, creating compliance and accountability issues.
- Static Reporting: Absence of real-time analytical views forces manual report generation, delaying decision-making processes.
- No Automated Business Rules: Critical constraints like active loan limits or automatic status transitions require manual monitoring and enforcement.

2.2 Proposed System

The proposed Loan Management System addresses these limitations through a comprehensive database-driven architecture:

- **Automated Workflow Processing:** Stored procedures encapsulate the entire loan approval lifecycle, ensuring that repayment generation, status updates, and ledger entries occur within a single atomic transaction to maintain consistency.
- **Concurrency Control:** Row-level locking (FOR UPDATE) ensures that when a loan officer accesses a pending loan for approval, no other user can simultaneously modify that record, preventing double-disbursement scenarios.

- **Normalized Schema Design:** Borrower information is stored once in the borrowers table, with loans referencing it via foreign keys. This eliminates redundancy and ensures that borrower updates (like address changes) automatically reflect across all associated loans.
- **Comprehensive Audit System**: Every significant action (loan creation, approval, disbursement, repayment recording) is automatically logged in the audit_logs table with user attribution and timestamp, supporting compliance requirements.
- **Real-Time Analytics:** Database views like v_borrower_portfolio provide instant access to aggregated borrower statistics (total loans, outstanding amounts, credit scores) without requiring complex join queries in application code.
- **Automated Business Logic:** Database triggers automatically enforce critical rules, such as restricting the number of active loans per borrower and auto-closing accounts once the outstanding balance reaches zero.
- **Role-Based Access Control:** The ENUM-based role system (admin, loan_officer, accountant) integrates with JWT authentication to restrict access to sensitive operations based on user privileges.

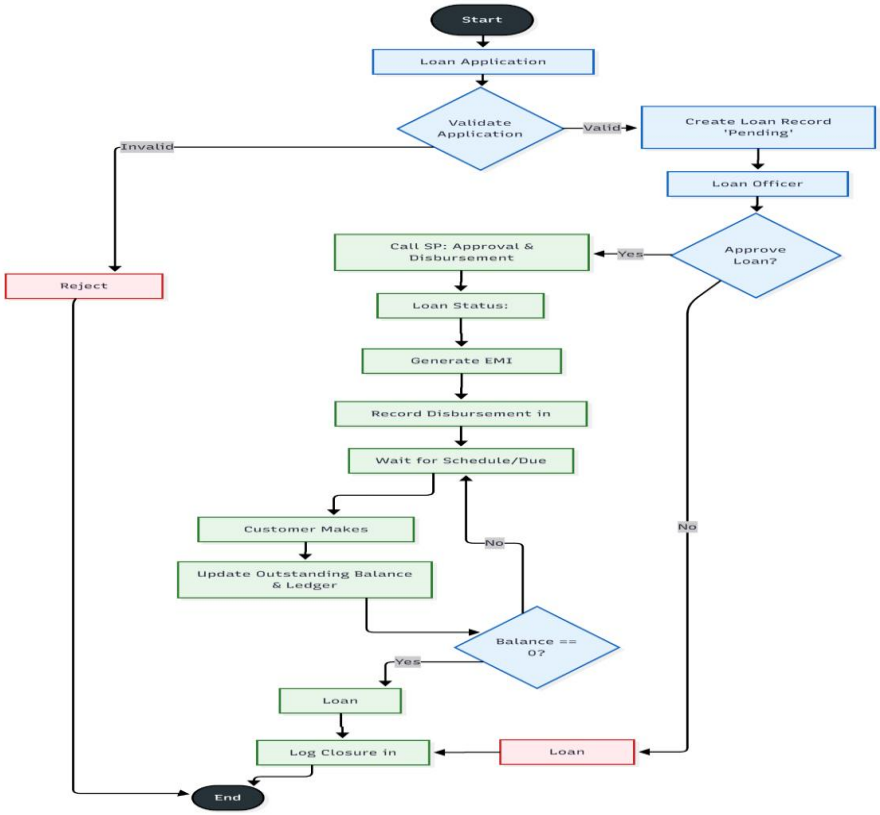


Figure 2.2: Flow Diagram

2.3 Feasibility Study

Aspect	Status	Rationale
Technical Feasibility	Highly Feasible	Built on proven technologies: PostgreSQL (stable database with 30+ years of development), FastAPI (mature Python framework), React (industry-standard frontend library). All components have extensive documentation and community support
Operational Feasibility	Highly Feasible	Directly addresses pain points in loan management workflows. The system's design aligns with standard banking operations including loan origination, collateral management, EMI tracking, and regulatory reporting.
Economic Feasibility	Feasible	PostgreSQL is open-source with no licensing costs. Python and JavaScript ecosystems are free. Primary costs are development time and infrastructure (cloud hosting), which are minimal compared to commercial database systems
Security Feasibility	Highly Feasible	JWT-based authentication, role-based access control, SQL injection prevention through parameterized queries, and comprehensive audit logging meet financial industry security standards..
Scalability Feasibility	Feasible	PostgreSQL handles millions of records efficiently with proper indexing. Horizontal scaling is possible through read replicas. The stateless FastAPI backend scales horizontally behind load balancers.

Table 2.3: Feasibility Study

CHAPTER 3
SYSTEM DESIGN

3.1 ER Diagram Entities

The Entity-Relationship model captures the conceptual structure of the loan management domain through the following key entities and their relationships:

- **Users:** Manages system access for admins and officers with role-based security.
- **Loan Types:** Defines loan products with specific interest rates, limits, and tenures.
- **Borrowers:** Stores customer profiles, income details, and credit scores.
- **Loans:** Tracks individual loan agreements, lifecycle statuses, and outstanding balances.
- **Collateral:** Records assets or documents pledged as security against specific loans.
- **Ledger:** Logs all financial movements including disbursements, penalties, and interest.
- **Repayments:** Manages EMI schedules, due dates, and payment tracking.
- **Receipts:** Stores official proof of payment records for specific repayment transactions.
- **Audit Logs:** Records user activities and system changes for security and compliance.

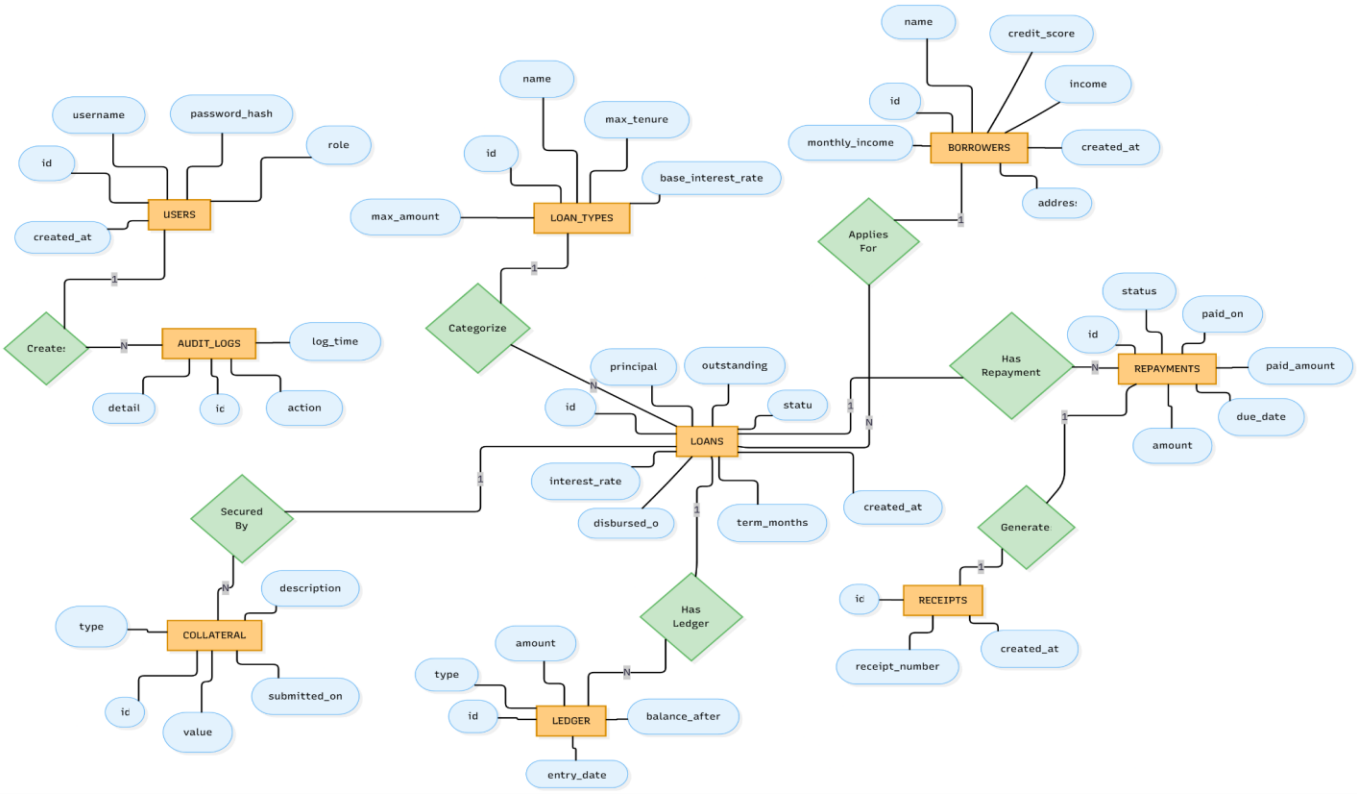


Figure 3.1: ER Diagram

3.2 Schema Diagram

The Relational Schema defines the physical PostgreSQL implementation, enforcing strict rules to ensure data consistency and performance.

- **Primary & Foreign Keys:** Each table uses SERIAL Primary Keys for identification, while Foreign Keys with cascade actions maintain integrity between loans, borrowers, and ledgers.
- **Unique Constraints:** Strict constraints on `receipt_number` and `username` prevent duplicate financial records and ensure unique user credentials.
- **Performance Indexes:** Strategic composite and partial indexes optimize high-frequency queries for active borrower loans, overdue repayments, and transaction history.
- **Data Integrity via ENUMs:** Custom data types strictly limit input values for User Roles and Loan Statuses, preventing invalid logic states.
- **Normalization:** The design adheres to Third Normal Form (3NF), minimizing data redundancy and ensuring a scalable, efficient structure.

Figure 3.2: Schema Diagram

CHAPTER 4
NORMALIZATION

4.1 First Normal Form (1NF)

The database design ensures atomicity through the following mechanisms:

- **Single-Valued Attributes:** Every column contains only atomic values. For example, the loans.principal stores a single monetary amount, not a comma-separated list of disbursement tranches.
- **No Repeating Groups:** Multiple repayments for a single loan are stored as separate rows in the repayments table rather than as repeated columns (repayment_1, repayment_2, etc.) within the loans table. This enables unlimited repayment records per loan.
- **Unique Row Identification:** Each table has a designated SERIAL PRIMARY KEY that uniquely identifies every row, enabling precise record retrieval and modification.
- **Consistent Data Types:** Monetary values consistently use NUMERIC(12,2) to prevent floating-point precision errors in financial calculations.
- The 1NF structure enables straightforward querying and eliminates the complexity of parsing multi-value fields, ensuring database operations remain efficient and reliable.

4.2 Second Normal Form (2NF)

The schema eliminates partial dependencies by ensuring all non-key attributes depend on the entire primary key:

- **Full Functional Dependency in Single-Key Tables:** In tables like loans, all attributes (principal, interest_rate, term_months, status, outstanding) depend fully on the primary key loan_id, not on any subset of it.
- **Proper Junction Table Design:** The receipts table establishes a one-to-one relationship with repayments through the unique constraint on repayment_id. The receipt_number and created_at depend on the complete primary key receipt_id, not partially on any composite key components.
- **Elimination of Transitive Storage:** Borrower details like name and address are stored only in the borrowers table. The loans table stores only the borrower_id foreign key, avoiding redundant storage of borrower information across multiple loan records.

- **Separation of Concerns:** The audit_logs table stores user action details separately from the users table, preventing the need to duplicate user information with every audit entry.
- This normalization level ensures that updating a borrower's contact information requires only a single UPDATE statement on the borrowers table, automatically reflecting across all associated loans.

4.3 Third Normal Form (3NF)

The design eliminates transitive dependencies where non-key attributes depend on other non-key attributes:

- **Direct Primary Key Dependency:** In the loans table, the interest_rate depends directly on loan_id, not on another non-key attribute like borrower_id. If interest rates varied by borrower characteristics, those would be calculated through the borrower_id foreign key relationship, not stored redundantly.
- **Elimination of Derived Data:** The outstanding balance in the loans table is maintained through ledger transactions rather than being derivable from other loan attributes. While technically calculable from repayment history, it's stored for performance optimization (a common denormalization for read-heavy queries).
- **Isolated Entity Attributes:** The loan_types table stores loan product characteristics (max_amount, max_tenure, base_interest_rate) that depend only on the loan_type_id, not on any other non-key attributes.
- **Independent Dimension Tables:** The collateral table stores collateral details (type, value, description) that depend solely on collateral_id, with the relationship to loans established through the loan_id foreign key.
- By achieving 3NF, the database minimizes update anomalies, reduces storage overhead, and maintains logical data independence. Changes to loan type policies (like modifying maximum tenure) require updates only to the loan_types table, automatically affecting all new loans without touching existing loan records.

CHAPTER 5
DATABASE IMPLEMENTATION

5.1 Table Structure

5.1.1 Users Table

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(128) UNIQUE NOT NULL,
    password_hash VARCHAR(256) NOT NULL,
    role role_enum NOT NULL DEFAULT 'loan_officer',
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

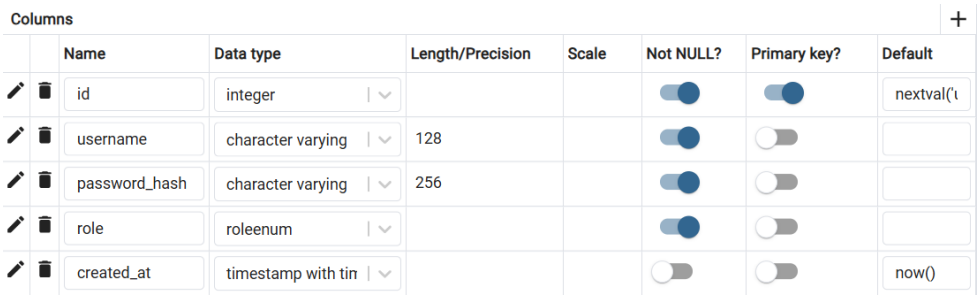


Figure 5.1.2: Users

5.1.2 Loan Type Table

```
CREATE TABLE loan_types (
    id SERIAL PRIMARY KEY,
    name VARCHAR(64) UNIQUE NOT NULL,
    max_amount NUMERIC(12, 2) NOT NULL,
    max_tenure INTEGER NOT NULL,
    base_interest_rate FLOAT NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

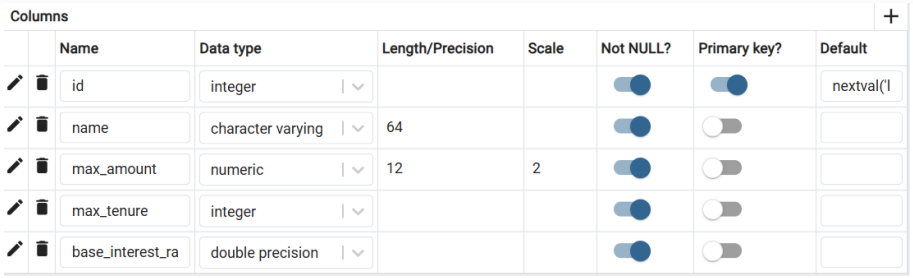


Figure 5.1.3: LoanTypes

5.1.3 Borrowers Table

```
CREATE TABLE borrowers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(256) NOT NULL,
    address TEXT,
    income NUMERIC(12, 2),
    monthly_income NUMERIC(12, 2),
    credit_score INTEGER,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

Columns

		Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
✎	🗑	id	integer			●	●	nextval('l
✎	🗑	name	character varying	256		●	○	
✎	🗑	address	text			○	○	
✎	🗑	income	numeric	12	2	○	○	
✎	🗑	monthly_income	numeric	12	2	○	○	
✎	🗑	credit_score	integer			○	○	
✎	🗑	created_at	timestamp with tin			○	○	now()

Figure 5.1.3: Borrowers

5.1.4 Loans Table

```
CREATE TABLE loans (
    id SERIAL PRIMARY KEY,
    borrower_id INTEGER NOT NULL,
    loan_type_id INTEGER,
    principal NUMERIC(12, 2) NOT NULL,
    interest_rate FLOAT NOT NULL,
    term_months INTEGER NOT NULL,
    disbursed_on TIMESTAMP WITH TIME ZONE,
    status loan_status_enum NOT NULL DEFAULT 'pending',
    outstanding NUMERIC(12, 2),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (borrower_id) REFERENCES borrowers(id) ON DELETE CASCADE,
    FOREIGN KEY (loan_type_id) REFERENCES loan_types(id) ON DELETE SET NULL
);
```

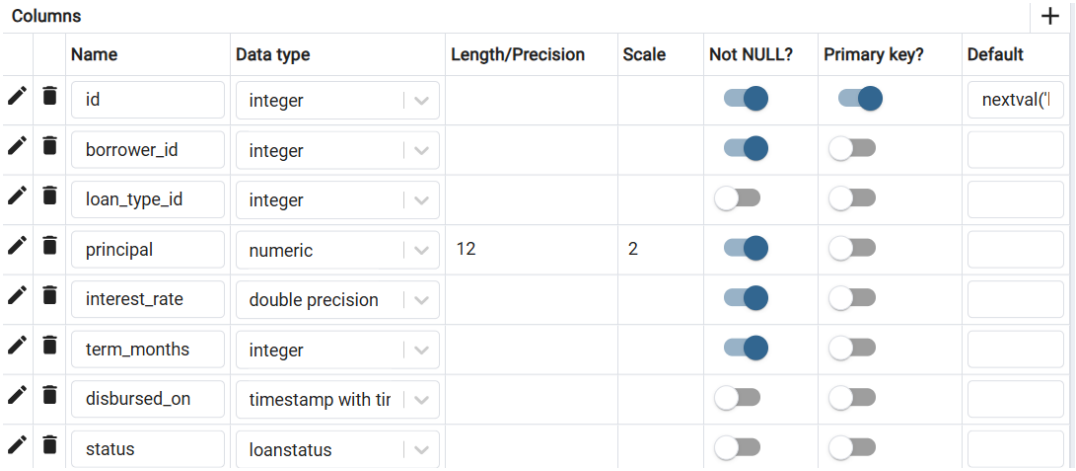


Figure 5.1.4: Loans

5.1.5 Repayments Table

```
CREATE TABLE repayments (
    id SERIAL PRIMARY KEY,
    loan_id INTEGER NOT NULL,
    due_date TIMESTAMP WITH TIME ZONE NOT NULL,
    amount NUMERIC(12, 2) NOT NULL,
    paid_amount NUMERIC(12, 2) NOT NULL DEFAULT 0,
    paid_on TIMESTAMP WITH TIME ZONE,
    status VARCHAR(32) NOT NULL DEFAULT 'due',
    FOREIGN KEY (loan_id) REFERENCES loans(id) ON DELETE CASCADE
);
```

Figure 5.1.5 : Repayments Table

5.1.6 Ledger Table

```
CREATE TABLE ledger (
    id SERIAL PRIMARY KEY,
    loan_id INTEGER NOT NULL,
    type VARCHAR(32) NOT NULL,
    amount NUMERIC(12, 2) NOT NULL,
    date TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    balance_after NUMERIC(12, 2),
    FOREIGN KEY (loan_id) REFERENCES loans(id) ON DELETE CASCADE
);
```

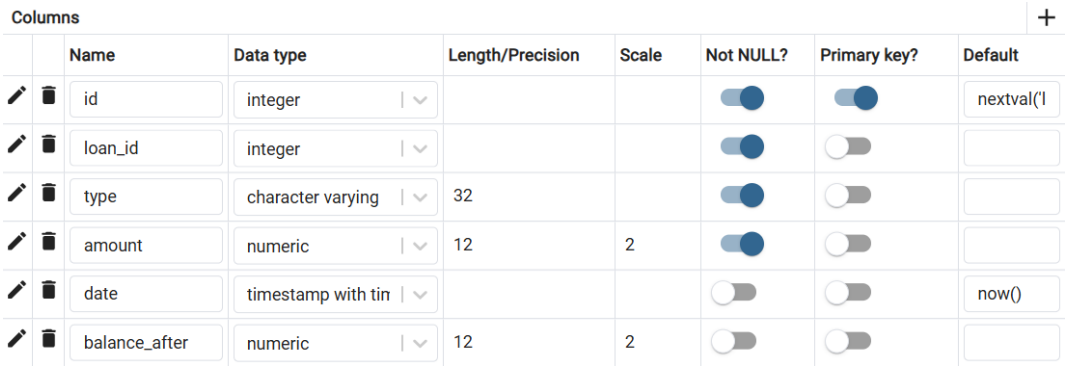
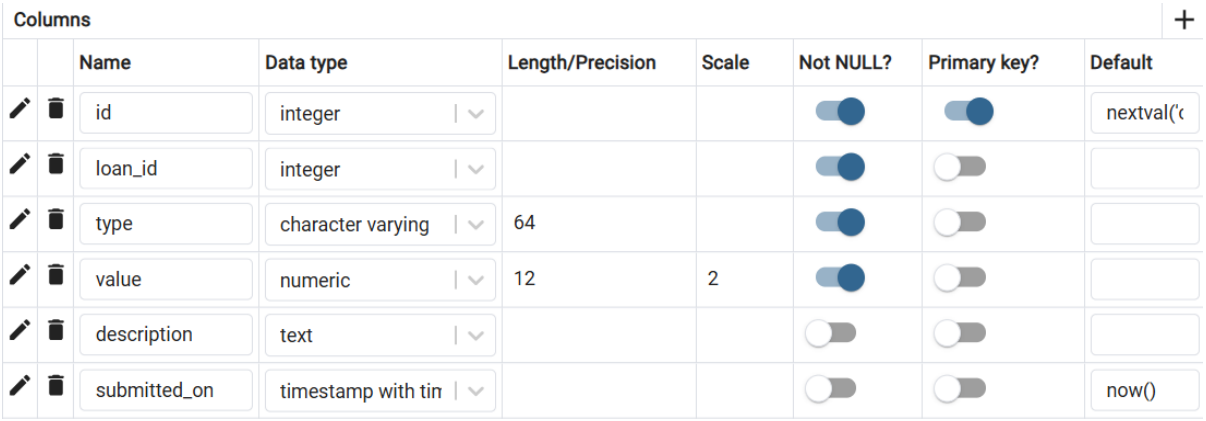


Figure 5.1.6 : Ledger Table

5.1.7 Collateral Table

```
CREATE TABLE collateral (
    id SERIAL PRIMARY KEY,
    loan_id INTEGER NOT NULL,
    type VARCHAR(64) NOT NULL,
    value NUMERIC(12, 2) NOT NULL,
    description TEXT,
    submitted_on TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (loan_id) REFERENCES loans(id) ON DELETE CASCADE
);
```



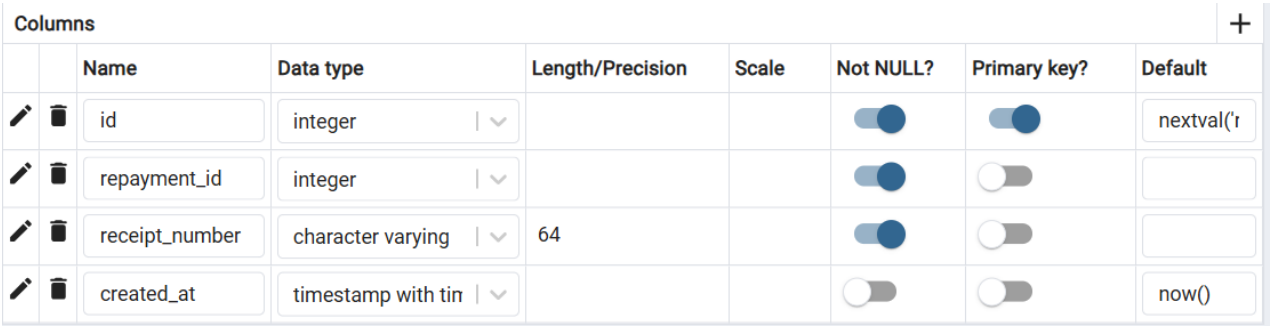
Columns

		Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
✎	🗑	id	integer			●	●	nextval('(
✎	🗑	loan_id	integer			●	○	
✎	🗑	type	character varying	64		●	○	
✎	🗑	value	numeric	12	2	●	○	
✎	🗑	description	text			○	○	
✎	🗑	submitted_on	timestamp with tin			○	○	now()

Figure 5.1.7 : Collateral Table

5.1.8 Receipts Table

```
CREATE TABLE receipts (
    id SERIAL PRIMARY KEY,
    repayment_id INTEGER UNIQUE NOT NULL,
    receipt_number VARCHAR(64) UNIQUE NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (repayment_id) REFERENCES repayments(id) ON DELETE CASCADE
);
```



Columns

		Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?	Default
✎	🗑	id	integer			●	●	nextval('r
✎	🗑	repayment_id	integer			●	○	
✎	🗑	receipt_number	character varying	64		●	○	
✎	🗑	created_at	timestamp with tin			○	○	now()

Figure 5.1.8: Receipts Table

5.1.9 Audit Logs

```
CREATE TABLE audit_logs (
    id SERIAL PRIMARY KEY,
    user_id INTEGER,
    action VARCHAR(64) NOT NULL,
    details TEXT,
    timestamp TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE SET NULL
);
```

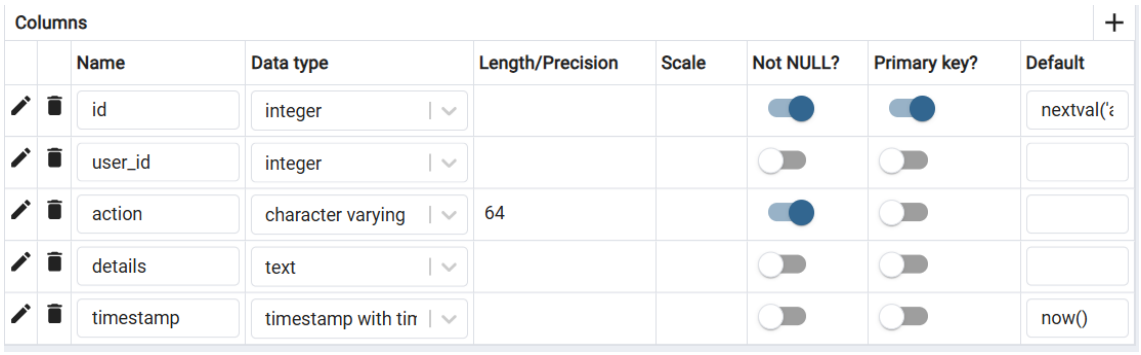


Figure 5.1.9: Audit Logs

5.2 Database Views

```
CREATE OR REPLACE VIEW v_borrower_portfolio AS
SELECT
    b.id AS borrower_id,
    b.name AS borrower_name,
    b.credit_score,
    COUNT(l.id) AS total_loans_count,
    COALESCE(SUM(l.principal), 0) AS total_principal_disbursed,
    COALESCE(SUM(l.outstanding), 0) AS total_outstanding_amount,
    COALESCE(SUM(l.principal) - SUM(l.outstanding), 0) AS
total_repaid_principal
FROM
    borrowers b
LEFT JOIN
    loans l ON b.id = l.borrower_id
GROUP BY
    b.id, b.name, b.credit_score;
```

Purpose: Provides instant borrower creditworthiness assessment by aggregating all loan data per customer.

Use Cases:

- Dashboard display of borrower financial standing
- Loan application evaluation (checking existing debt load)
- Risk analysis for portfolio management

```sql
CREATE OR REPLACE VIEW v_overdue_repayments AS
SELECT
    r.id AS repayment_id,
    l.id AS loan_id,
    b.name AS borrower_name,
    b.address AS borrower_address,
    r.due_date,
    r.amount AS amount_due,
    EXTRACT(DAY FROM (CURRENT_DATE - r.due_date)) AS days_overdue
FROM
    repayments r
JOIN
    loans l ON r.loan_id = l.id
JOIN
    borrowers b ON l.borrower_id = b.id
WHERE
    r.status IN ('due', 'overdue')
    AND r.due_date < CURRENT_DATE;
```

Purpose: Collections management tool for identifying missed payments requiring follow-up.

Benefits: Eliminates need for complex multi-table joins in application code, providing ready-to-use data for overdue reports and automated reminder systems.

5.3 Stored Procedures

```
CREATE OR REPLACE PROCEDURE sp_approve_and_disburse_loan(
    p_loan_id INT,
    p_user_id INT
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_loan_record RECORD;
    v_monthly_emi NUMERIC(10,2);
    v_total_interest NUMERIC(10,2);
    v_repayment_date DATE;
    v_disbursed_date TIMESTAMP;
    i INT;
BEGIN

    SELECT * INTO v_loan_record
    FROM loans
    WHERE id = p_loan_id
    FOR UPDATE;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Loan ID % not found', p_loan_id;
    END IF;

    IF v_loan_record.status != 'pending' THEN
        RAISE EXCEPTION 'Loan is not in pending status (Current: %)',
v_loan_record.status;
    END IF;

    v_disbursed_date := CURRENT_TIMESTAMP;
    UPDATE loans
    SET status = 'active',
        disbursed_on = v_disbursed_date,
        outstanding = principal
    WHERE id = p_loan_id;
```

```
    v_total_interest := (v_loan_record.principal *
v_loan_record.interest_rate * (v_loan_record.term_months::NUMERIC / 12)) /
100;
    v_monthly_emi := (v_loan_record.principal + v_total_interest) /
v_loan_record.term_months;

    FOR i IN 1..v_loan_record.term_months LOOP
        v_repayment_date := (v_disbursed_date + (i || '
month')::INTERVAL)::DATE;
        INSERT INTO repayments (loan_id, due_date, amount, paid_amount,
status)
        VALUES (
            p_loan_id,
            v_repayment_date,
            ROUND(v_monthly_emi, 2),
            0,  --
            'due'
        );
    END LOOP;

    INSERT INTO ledger (loan_id, type, amount, balance_after)
    VALUES (p_loan_id, 'disbursement', v_loan_record.principal,
v_loan_record.principal);

    INSERT INTO audit_logs (user_id, action, details)
    VALUES (p_user_id, 'LOAN_DISBURSED', 'Loan ID ' || p_loan_id || '
disbursed. Status: Active.');

END;
$$;
```

Critical Features:

- **FOR UPDATE Row Lock:** Prevents race condition where two loan officers could simultaneously approve the same loan
- **Atomic Transaction:** All steps (status update, schedule generation, ledger entry, audit log) commit together or rollback entirely
- **Automated Schedule Generation:** Eliminates manual EMI calculation and entry errors
- **Audit Trail:** Every disbursement is logged with user attribution for compliance
- **Usage:** `CALL sp_approve_and_disburse_loan(4, 9);` where 4 is loan_id and 9 is user_id

5.4 Triggers

```
CREATE OR REPLACE FUNCTION fn_enforce_active_loan_limit()
RETURNS TRIGGER AS $$
DECLARE
    v_active_count INT;
BEGIN
    SELECT COUNT(*) INTO v_active_count
    FROM loans
    WHERE borrower_id = NEW.borrower_id AND status = 'active';

    IF v_active_count >= 3 THEN
        RAISE EXCEPTION 'Borrower % already has % active loans. Limit is 3.',
NEW.borrower_id, v_active_count;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_enforce_active_loan_limit
BEFORE INSERT ON loans
FOR EACH ROW
EXECUTE FUNCTION fn_enforce_active_loan_limit();
```

Purpose: Business rule enforcement preventing borrowers from exceeding three concurrent active loans.

Behavior: Automatically blocks INSERT operations that would violate the limit, raising an exception before database modification.

CHAPTER 6
SCREENSHOTS

6.1 Login Page

The login page serves as the entry point to the Loan Management System, providing secure access control for authorized personnel. The authentication system implements JWT-based token management with role-based access control.

Key Features:

- Username and password authentication with secure validation
- Role-based login differentiating between Admin, Loan Officer, and Accountant
- Password encryption using bcrypt hashing algorithm
- Error handling for invalid credentials

The login interface features a clean, professional design with input validation and immediate feedback for authentication errors. Upon successful login, users are redirected to their role-specific dashboard.

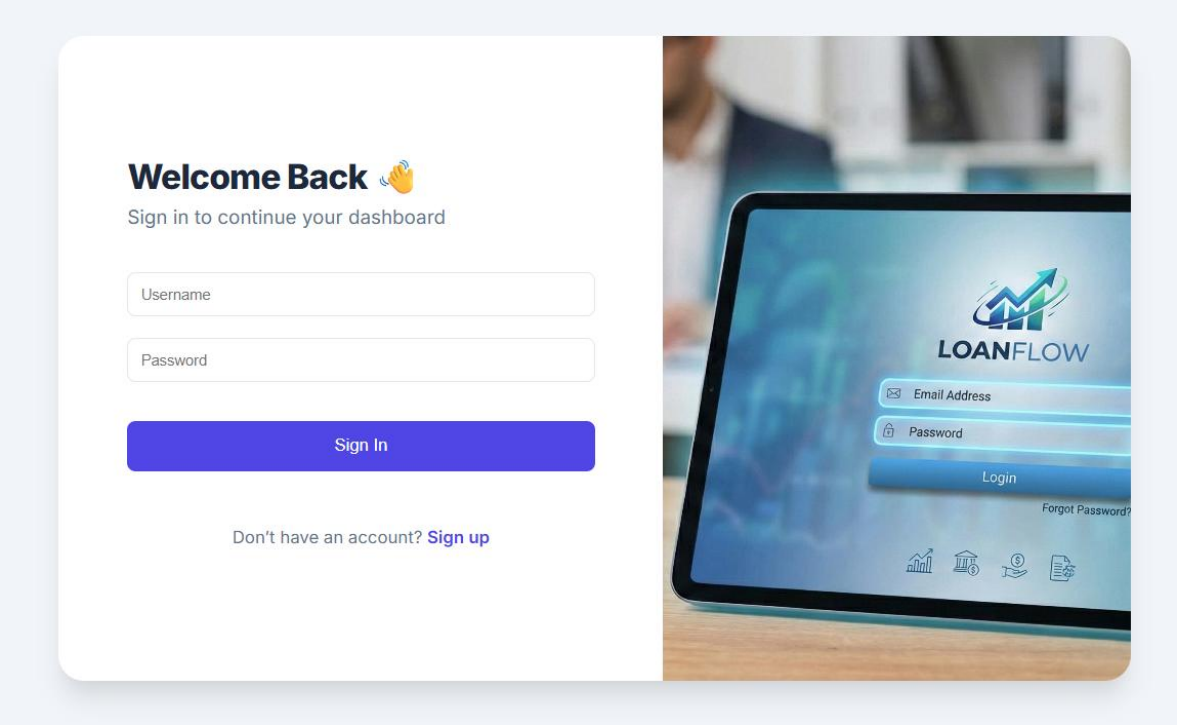


Figure 6.1: User Login Page

6.2 Dashboard Overview

The dashboard provides a comprehensive overview of the loan management system's key metrics and quick access to primary functions. The interface adapts based on user role, displaying relevant information and action buttons.

Key Features:

- Total active loans, pending approvals, and closed loans statistics
- Total disbursed amount and outstanding balance
- Quick action buttons for creating loans, adding borrowers, and recording payments
- Recent activity feed showing latest transactions
- Performance charts displaying loan distribution and monthly trends

The dashboard utilizes Recharts library for interactive data visualization, allowing users to drill down into specific metrics. Real-time updates ensure that displayed information reflects the current database state.

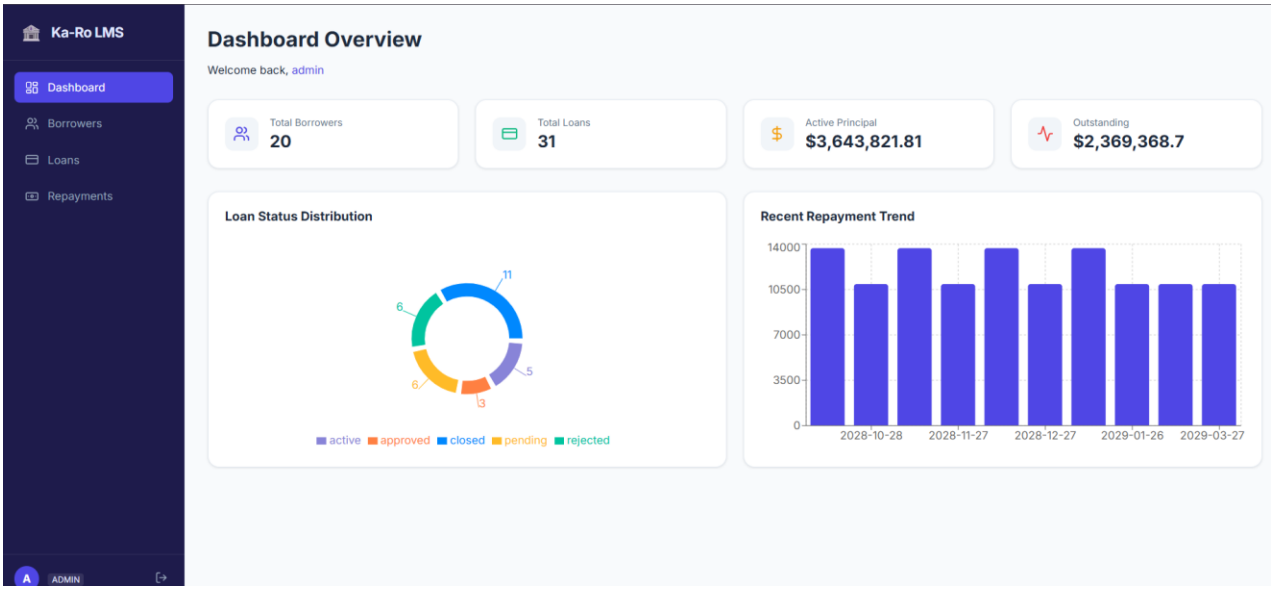


Figure 6.2: System Dashboard

6.3 Borrower Management

The borrower management interface allows loan officers to create, view, and update customer profiles with comprehensive financial information. This module serves as the foundation for loan application processing.

Key Features:

- Borrower registration form with personal and financial details
- Credit score tracking and visualization
- Complete loan history for each borrower
- Portfolio summary showing total borrowed amount and outstanding balance
- Document upload for ID proof, income proof, and address verification

The interface includes real-time validation to ensure data completeness and accuracy. The system automatically fetches existing borrower data for loan applications, eliminating duplicate data entry. The borrower portfolio view queries the v_borrower_portfolio database view for instant aggregate statistics.

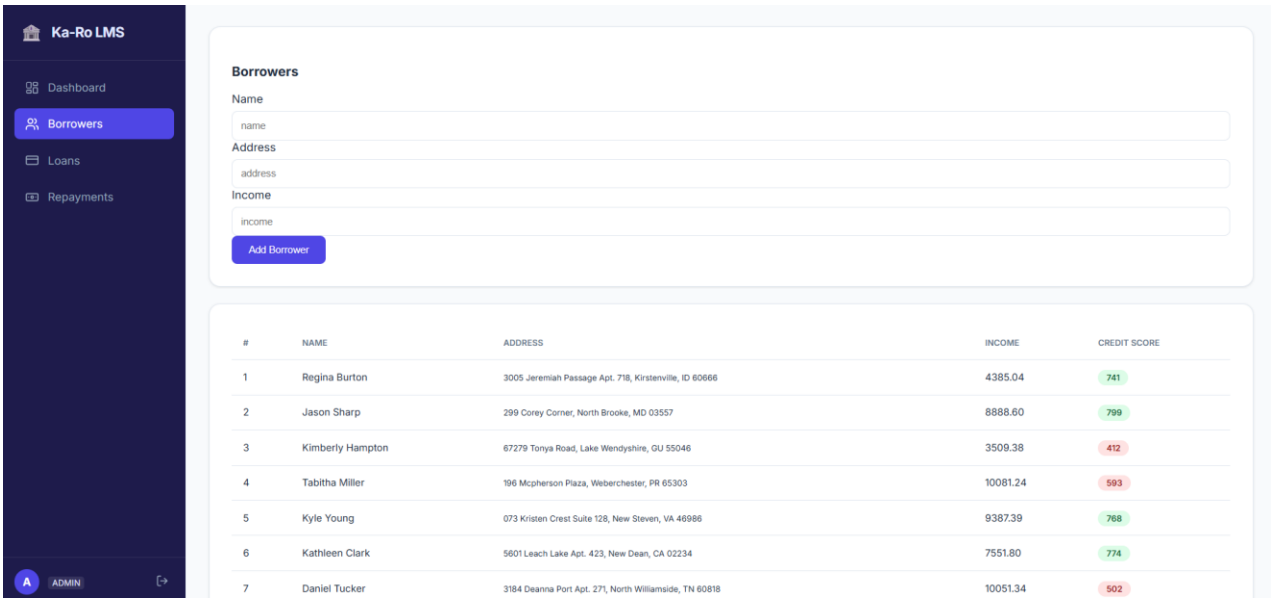


Figure 6.3: Borrower Registration Form

6.4 Loan Management

The loan application module streamlines the loan origination process from initial request through final approval and disbursement. This workflow integrates automated eligibility checks and database-level transaction management.

Key Features:

- Loan application form with borrower selection and loan type configuration
- Automated eligibility verification based on credit score and active loan count
- Pending loan approval queue for loan officers
- Single-click approval triggering the sp_approve_and_disburse_loan stored procedure
- Automatic EMI schedule generation with detailed repayment breakdown
- Real-time loan status tracking (Pending → Approved → Active → Closed)

When a loan officer approves a pending loan, the system executes the stored procedure which performs row-level locking, updates loan status, generates the complete repayment schedule, creates ledger entries, and logs the action—all within a single atomic transaction. The trg_enforce_active_loan_limit trigger prevents borrowers from exceeding three active loans.

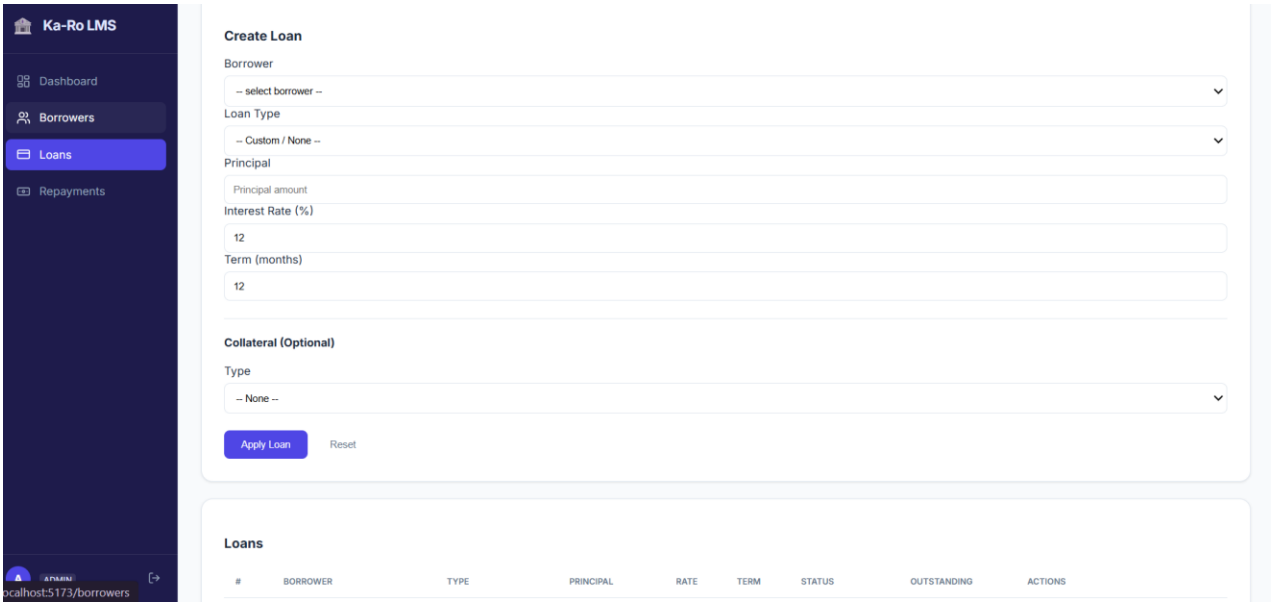


Figure 6.4.1: Loan Application Form

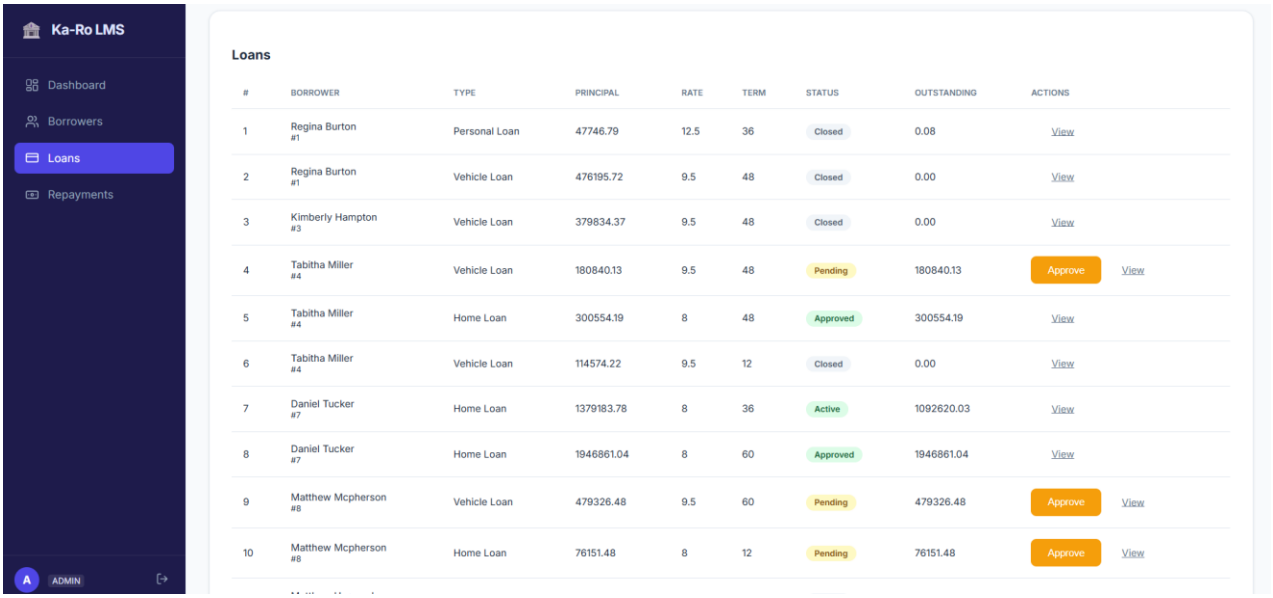


Figure 6.4.2: Pending Loan Approval Interface

6.5 Repayment Management

The repayment module enables accountants to record EMI payments, track payment history, and generate receipts. The interface integrates with the ledger system for complete financial tracking.

Key Features:

- Due and overdue repayment list with borrower details
- Payment recording form with amount validation
- Partial payment support with automatic status updates
- Automatic receipt generation with unique receipt numbers
- Real-time loan balance updates
- Automatic loan closure when outstanding balance reaches zero (via trg_auto_update_loan_status trigger)

The system validates payment amounts and provides warnings for discrepancies. The overdue payments view utilizes the v_overdue_repayments database view, which automatically calculates days overdue and provides complete borrower contact information for follow-up.

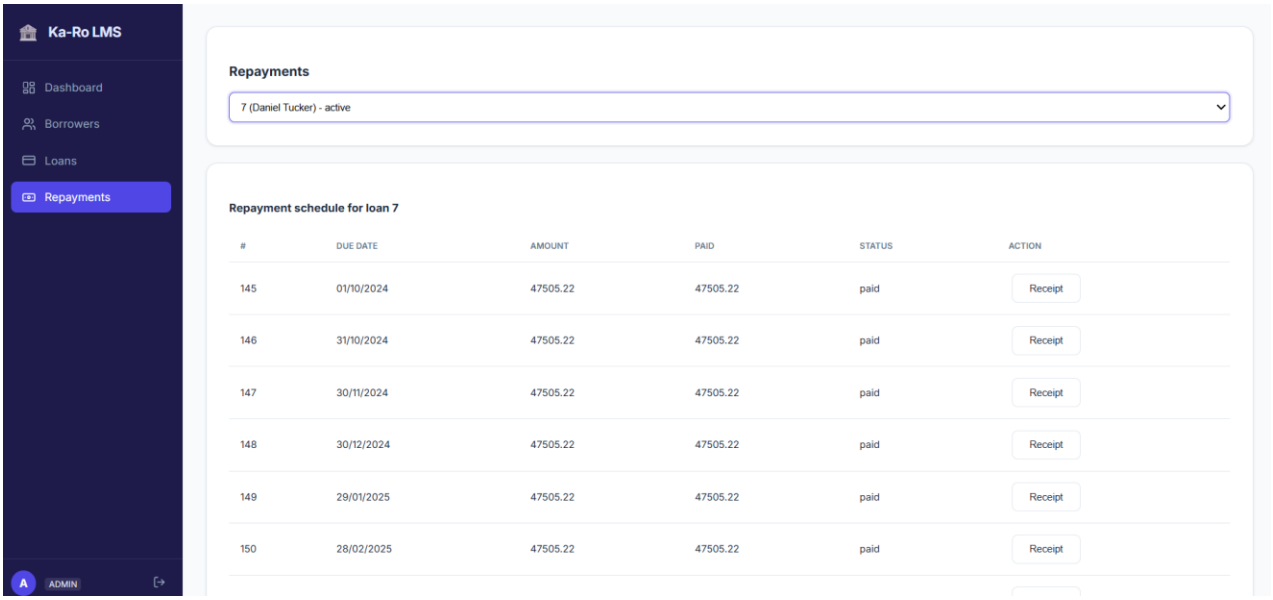


Figure 6.5.1: Repayment Recording Interface

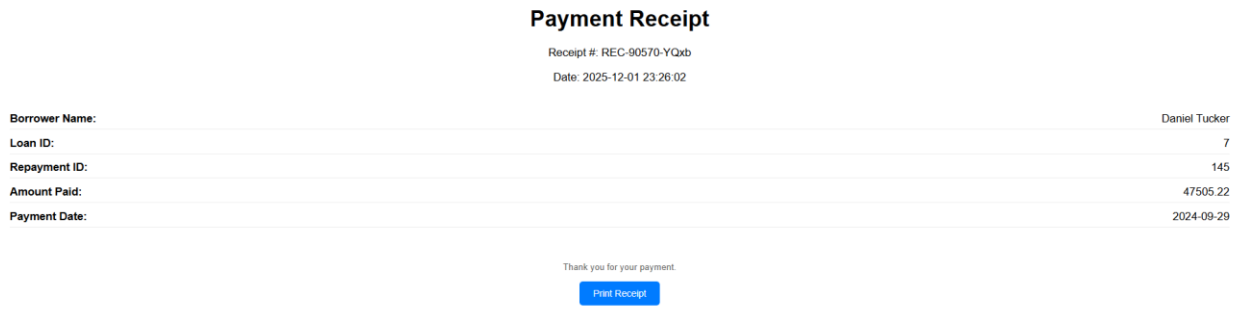


Figure 6.5.2: Payment Receipt Display

CHAPTER 7
CONCLUSION

7.1 Conclusion

This project delivers a production-ready Loan Management System built on a rigorously designed PostgreSQL architecture that ensures transactional integrity, operational efficiency, and regulatory compliance. By enforcing 3NF normalization, using ENUM types, and applying precise foreign key constraints with cascade rules, the system maintains data accuracy and eliminates redundancy. Complex loan workflows are handled through atomic stored procedures like sp_approve_and_disburse_loan, supported by row-level locking to guarantee safe concurrency and prevent race conditions in high-load environments.

Business logic is automated through intelligent triggers that enforce lending policies and maintain real-time portfolio accuracy, while optimized views streamline analytical queries for dashboards. The backend, powered by FastAPI, pairs with a responsive and type-safe React frontend to deliver high performance across the stack. With comprehensive indexing and audit logging, the system is both scalable and secure, ready to meet current needs and support future enhancements such as dynamic pricing or third-party credit bureau integrations.

7.2 Future Enhancements

1. Implement stored procedure for safe loan cancellation and refund processing, including automatic seat release and ledger reversal with audit trail
2. Introduce dynamic pricing calculation based on borrower credit score, loan type, and market interest rates through database functions
3. Add penalty calculation triggers for late payments with automated penalty charge insertion into ledger
4. Implement interest accrual mechanism with scheduled jobs to calculate and post daily interest charges to loans

CHAPTER 8
REFERENCES

1. Connolly, T., & Begg, C. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson. (Used as reference for relational database design principles and normalization theory)

2. PostgreSQL Global Development Group. (n.d.). *PostgreSQL Documentation - Version 16*. Retrieved from https://www.postgresql.org/docs/

3. FastAPI Documentation. (n.d.). *FastAPI - Modern, Fast (High-Performance) Web Framework*. Retrieved from https://fastapi.tiangolo.com/

4. Chen, P. P. (1976). The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems, 1*(1), 9–36. https://doi.org/10.1145/320434.320440