DEV OPPS

SRS DOCUMENT

DEMO 3

STOCKFELLOW

BRIGHT BYTE ENTERPRISES

TABLE OF CONTENTS

INTRODUCTION

Stokvels have been a vital part of South African communities for generations, providing a trusted mechanism for collective savings and financial support. These groups foster a sense of community and mutual aid, enabling individuals to achieve goals that might otherwise be out of reach. However, traditional stokvels face significant challenges: trust issues due to lack of transparency, manual record-keeping prone to errors, and delays in payments that erode confidence. StockFellow aims to modernize this cultural practice by delivering a secure, automated, and inclusive digital solution that preserves its communal essence while overcoming these obstacles.

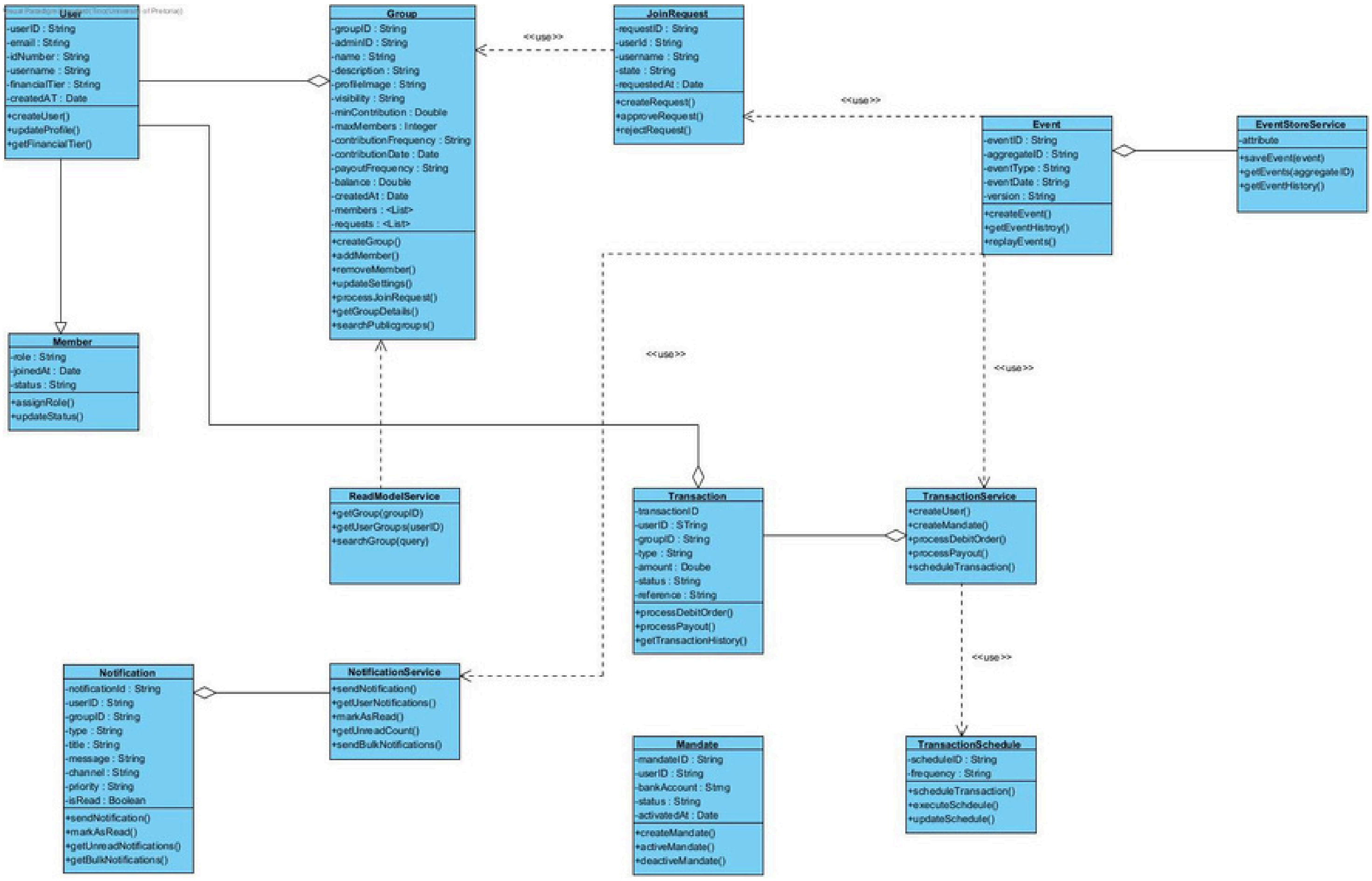
PROJECT DESCRIPTION

StockFellow is a modern financial solution designed to streamline and secure traditional stokvel savings groups. By leveraging automation, robust security, and open-source technologies, the platform addresses the core challenges of trust, efficiency, and compliance while introducing innovative features to enhance user experience.

Key Features

- **Secure User Registration & Verification:** Biometric authentication, ID verification, and credit scoring ensure safe onboarding, even for users without formal credit histories.
- **Automated Contributions & Payouts:** Scheduled debit orders and automated fund distribution eliminate delays, with bank API integration for seamless transactions.
- **Intelligent Group & Tier Management:** Users are assigned to affordability-based tiers using credit scoring, with automated promotions based on contribution consistency.
- **Fraud Detection & Security:** AI-driven fraud detection, multi-factor authentication (MFA), and real-time monitoring safeguard user funds and data.
- **Real-Time Notifications & Reporting:** SMS, email, and in-app alerts keep users informed about payments, tier changes, and group activities.
- **Compliance & Data Protection:** Adherence to POPIA, GDPR, and AML regulations, with encryption and audit logging for transparency.
- **Potential Enhancements:** AI-based risk assessment for emergency loans, blockchain for transparent transaction records, and USSD support for offline access.

DOMAIN MODEL



FUNCTIONAL REQUIREMENTS

R1: Set up a new account

- R1.1: Register a new user
  - **R1.1.1: Enter personal details:** The system shall prompt the user to input their username, full name, email address, phone number, ID number and password during registration..
  - **R1.1.2: Verify email address:** The system shall send a verification link to the provided email address and require the user to confirm it before proceeding.
  - **R1.1.3: Set up a secure password:** The system shall enforce password complexity rules (e.g., minimum length, special characters) and securely store the hashed password.
- R1.2: Verify user identity
  - **R1.2.1: Upload ID documents:** The system shall allow users to upload scanned copies or photos of government-issued ID documents for verification.
  - **R1.2.2: Perform biometric authentication:** The system shall authenticate users via WebAuthn using device biometrics and cryptographic key pairs stored securely on the device.
  - **R1.2.3: Confirm verification status:** The system shall display the verification status (e.g., pending, approved, rejected) and notify the user upon completion.
  - **R1.2.4: Require Multi Factor Authentication upon logging in:** The system shall send a one-time password (OTP) to the user’s registered email for verification.
- R1.3: Perform credit scoring
  - **R1.3.1: Collect financial information:** The system shall collect the user’s 6-month bank statement data, income sources, savings, investments, and existing financial obligations.
  - **R1.3.2: Analyse financial behaviour:** The system shall calculate a composite financial score based on: income stability, expense management, savings behaviour, and existing financial obligations. Red flags (e.g. high debt) may lower the tier and green flags (e.g. consistent savings) may raise it.

  - **R1.3.3: Assign affordability tier:** The system shall automatically assign the user to one of six tiers based on their financial score and risk indicators. Each tier defines recommended contribution, group size, payout frequency, and risk level.
  - **R1.3.3: Display tier assignment:** The system shall show the assigned tier to the user with a brief explanation and provide a way to request a review if needed.

R2: Manage a group/stokvel

- R2.1: Create a new stokvel group

  R2.1.1: Define group name and description: The system shall allow the group creator to input a unique name and a brief description for the stokvel group.

  R2.1.2: Set contribution amounts and schedules: The system shall enable the creator to specify the contribution amount and frequency, ensuring it falls within the recommended range for the user’s affordability tier.

  R2.1.3: Establish payout rules: The system shall allow the creator to define payout rules, such as the order of payouts and conditions for fund distribution.
- R2.2: Join an existing group
  - **R2.2.1: Search for available groups:** The system shall allow users to search and join groups that match their assigned affordability tier and preferred contribution amount as well as certain criteria like name and location.
  - **R2.2.2: Submit a join request:** The system shall allow users to send a request to join a group, which requires approval from group admin.
    - **R2.2.2.1: Review and accept debit mandate:** The system shall present the user with the group’s debit mandate agreement during the join process and require acceptance before the request is submitted.
    - **R2.2.2.2: Record mandate acceptance:** The system shall log the user's acceptance of the group mandate, including the timestamp and version of the mandate agreed to, for audit and legal purposes.

  - **R2.2.3: Await approval from group administrator:** The system shall notify the user once the group administrator approves or rejects their join request.
- R2.3: Manage group membership
  - **R2.3.1: Add new members:** The system shall enable group administrators to invite or approve new members to join the group. Administrators should consider tier compatibility when approving new members.
  - **R2.3.2: Remove existing members:** The system shall allow group administrators to remove members who violate group rules or fail to contribute.
  - **R2.3.3: Update member roles:** The system shall permit group administrators to assign or change roles (e.g., promote a member to co-administrator).

R3: Manage member contributions

- **R3.1: Set up debit orders**
  - **R3.1.1: Link bank account:** The system shall guide users through linking their bank account via a secure API for automated payments
  - **R3.1.2: Configure debit order amount and frequency:** The system shall allow users to set the exact amount and schedule for contributions, which must align with their assigned affordability tier range.
  - **R3.1.3: Confirm debit order setup:** The system shall send a confirmation notification once the debit order is successfully configured.
- **R3.2: Track contributions**
  - **R3.2.1: Record each contribution transaction:** The system shall log every successful contribution with details like amount, date, and transaction ID.
  - **R3.2.2: Display contribution history to the user:** The system shall provide a dashboard where users can view their past contributions in chronological order.
  - **R3.2.3: Calculate total contributions made:** The system shall compute and display the cumulative amount a user has contributed to their group, and result.

- R3.3: Handle missed contributions
    - **R3.3.1: Detect missed contributions:** The system shall identify when a scheduled contribution fails to process (e.g., due to insufficient funds).
        - **R3.3.1.1: Retry failed contributions up to 3 times:** The system shall automatically retry failed contributions up to three times, with a delay between each attempt (e.g., every 24 hours).
        - **R3.3.1.2: Log all retry attempts:** The system shall log each retry attempt for failed contributions, including the attempt number, timestamp, and result.
    - **R3.3.2: Send notification to the user:** The system shall alert the user via their preferred notification channel about the missed contribution.
        - **R3.3.2.1: Notify user of final failure:** The system shall notify the user when the final retry attempt fails, and inform them of any resulting consequences or next steps.
    - **R3.3.3: Apply penalties as per group rules:** The system shall automatically apply any penalties (e.g., fees or tier demotion) defined by the group.
        - **R3.3.3.1: Notify administrator of penalty enforcement:** The system shall notify the group administrator when penalties are applied due to missed contributions.

R4: Process member payouts

- R4.1: Schedule payouts
    - **R4.1.1: Determine payout dates based on group rules:** The system shall calculate payout dates according to the group's predefined schedule (e.g., every month, every 3 months).
    - **R4.1.2: Identify the next payout recipient:** The system shall select the next eligible member for a payout based on the group's rotation or rules.
    - **R4.1.3: Prepare payout transaction:** The system shall queue the payout transaction for processing on the scheduled date.

    - **R4.1.4: Daily payout eligibility check:** The system shall automatically check every day for payout-eligible cycles and initiate payout processing accordingly
- R4.2: Distribute funds
    - **R4.2.1: Verify recipient's bank account details:** The system shall confirm that the recipient's bank account is linked and valid before initiating the transfer.
    - **R4.2.2: Initiate fund transfer:** The system shall use a secure API to transfer the payout amount from the group's pooled funds to the recipient's account.
    - **R4.2.3: Confirm successful transfer:** The system shall verify the transaction's success and update the payout status accordingly.
- R4.3: Record payout transactions
    - **R4.3.1: Log payout details:** The system shall record the payout amount, recipient, date, and transaction reference for auditing.
    - **R4.3.2: Update group's financial records:** The system shall adjust the group's total funds and individual member balances after each payout.
    - **R4.3.3: Make transaction history available for audit:** The system shall store payout records in a secure, accessible format for transparency and compliance.
    - **R4.3.4: Display payout history to recipients:** The system shall allow users to view a list of all payouts they've received, including amount, date, and transaction status.

R5: Manage User Tiers

- R5.1: Assign users to affordability tiers
    - **R5.1.1: Evaluate user's affordability tier:** The system shall use the bank-statement-based financial assessment algorithm to evaluate the user's financial behaviour and risk profile.
    - **R5.1.2: Determine appropriate tier based on score:** The system shall assign the user to one of six affordability tiers based on the assessment and risk indicators.
    - **R5.1.3: Notify user of their tier assignment:** The system shall inform the user of their assigned tier and provide a brief explanation of the factors influencing the assignment.
- R5.2: Monitor contribution behaviour
    - **R5.2.1: Track consistency of contributions:** The system shall monitor whether users meet their contribution obligations on time and adjust risk/reliability scores to support potential tier adjustments.
    - **R5.2.2: Calculate reliability score:** The system shall compute a reliability score based on the user's contribution history (to support tier adjustments.
    - **R5.2.3: Check against promotion criteria:** The system shall compare the reliability score against thresholds for tier promotion.
- R5.3: Promote users to higher tiers
    - **R5.3.1: Identify users eligible for promotion:** The system shall automatically detect users who meet the criteria for a higher tier.
    - **R5.3.2: Update user's tier status:** The system shall upgrade the user's tier in the database and reflect the change in their profile.
    - **R5.3.3: Notify user of tier promotion:** The system shall send a congratulatory notification to the user about their tier upgrade.

R6: Provide security and fraud detection services

- R6.1: Implement multi-factor authentication (MFA)
    - **R6.1.1: Enable MFA for user accounts:** The system shall allow users to set up multi-factor authentication using email OTP or device-bound biometric verification.
    - **R6.1.2: Require MFA for login:** The system shall enforce MFA during the login process for all users.
    - **R6.1.3: Require MFA for fund transfers:** The system shall mandate MFA confirmation before processing any fund transfers or payouts.
- R6.2: Detect fraudulent activities
    - **R6.2.1: Analyze transaction patterns:** The system shall use AI algorithms to identify unusual patterns in contributions or payouts.
    - **R6.2.2: Flag suspicious activities:** The system shall mark transactions or user behaviors that deviate from normal patterns for review.
    - **R6.2.3: Alert system administrators:** The system shall notify administrators immediately when potential fraud is detected.
- R6.3: Monitor system in real-time
    - **R6.3.1: Set up real-time monitoring tools:** The system shall integrate monitoring software to track system performance and security metrics.
    - **R6.3.2: Track user activities:** The system shall log all user actions, especially those involving financial transactions.
    - **R6.3.3: Generate alerts for anomalies:** The system shall trigger alerts for any abnormal activities, such as multiple failed login attempts.

R7: Send notifications and reporting

- R7.1: Send real-time notifications
  - **R7.1.1: Configure notification preferences:** The system shall allow users to select which events trigger notifications (e.g., contributions, payouts).
  - **R7.1.2: Send notifications via SMS, email, or in-app:** The system shall deliver notifications through the user's preferred channel(s).
  - **R7.1.3: Ensure timely delivery of notifications:** The system shall prioritize notification delivery to ensure users receive updates promptly.
- R7.2: Generate reports
  - **R7.2.1: Collect data for reporting:** The system shall aggregate data on contributions, payouts, and group activities for reporting purposes.
  - **R7.2.2: Create customizable report templates:** The system shall provide templates for common reports (e.g., monthly contribution summaries) that users can personalize.
  - **R7.2.3: Allow users to download reports:** The system shall enable users to export reports in formats like PDF or CSV for offline access.
- R7.3: Customize notification preferences
  - **R7.3.1: Provide options for notification types:** The system shall let users choose which types of events they want to be notified about (e.g., only payouts).
  - **R7.3.2: Allow selection of notification channels:** The system shall offer multiple channels (SMS, email, in-app) and let users select their preferred ones.
  - **R7.3.3: Save user preferences:** The system shall store each user's notification settings and apply them consistently across the platform.

USER STORIES

Groups

- As a new user, I want to view a quick and interactive tutroial of the app when I first sign up, so that I understand how to use the key features before getting started.
- As a new user, I want to search for existing stokvel groups
- As a user, I want to send a request to join a stokvel group
- As a group admin, I want to view all pending membership requests, so that I can review and decide who to accept or reject.

Transactions

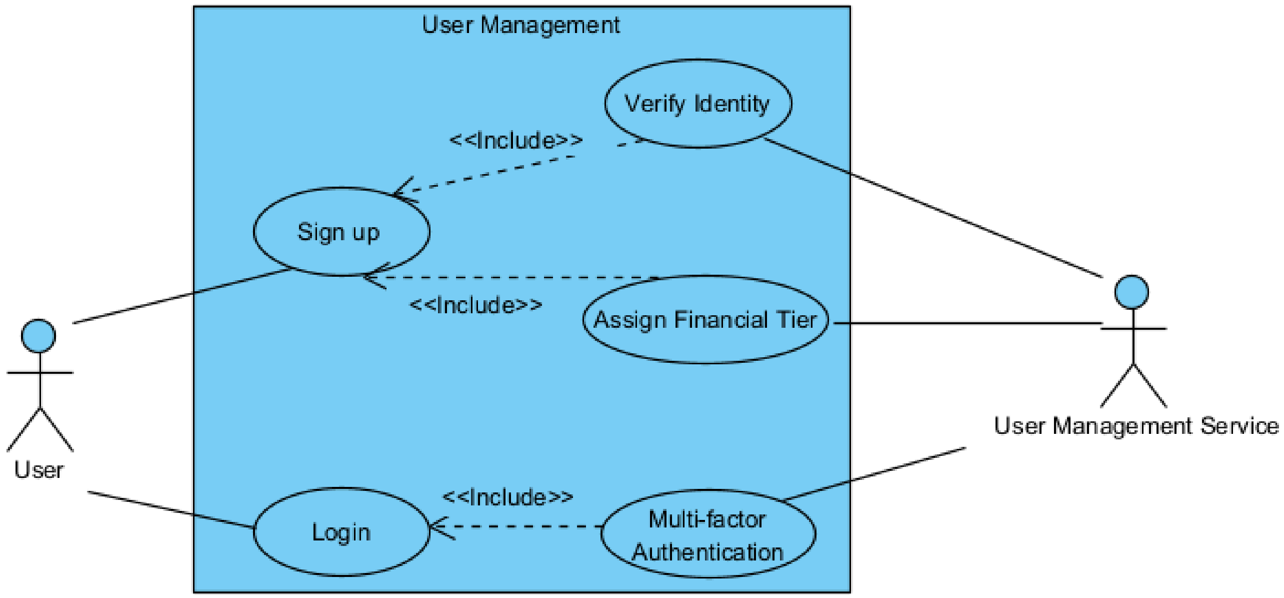
- As a user, I want to add a new payment card, set it to active, and be able to delete it.
- As a user, I want to view a history of my past contributions and payouts, so that I can keep track of my financial activity.

Notifications

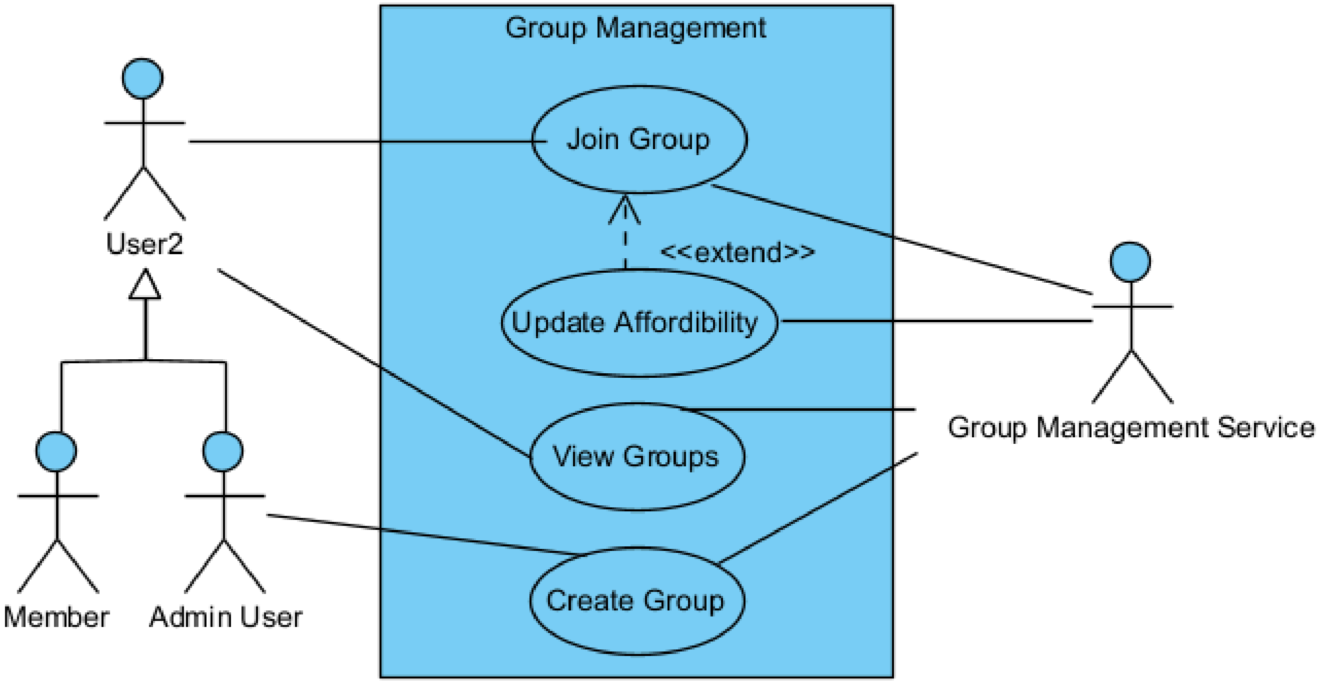
- As a user, I want to view a list of all past notifications in one place, so that I can review any missed alerts or messages.

USE CASE DIAGRAMS

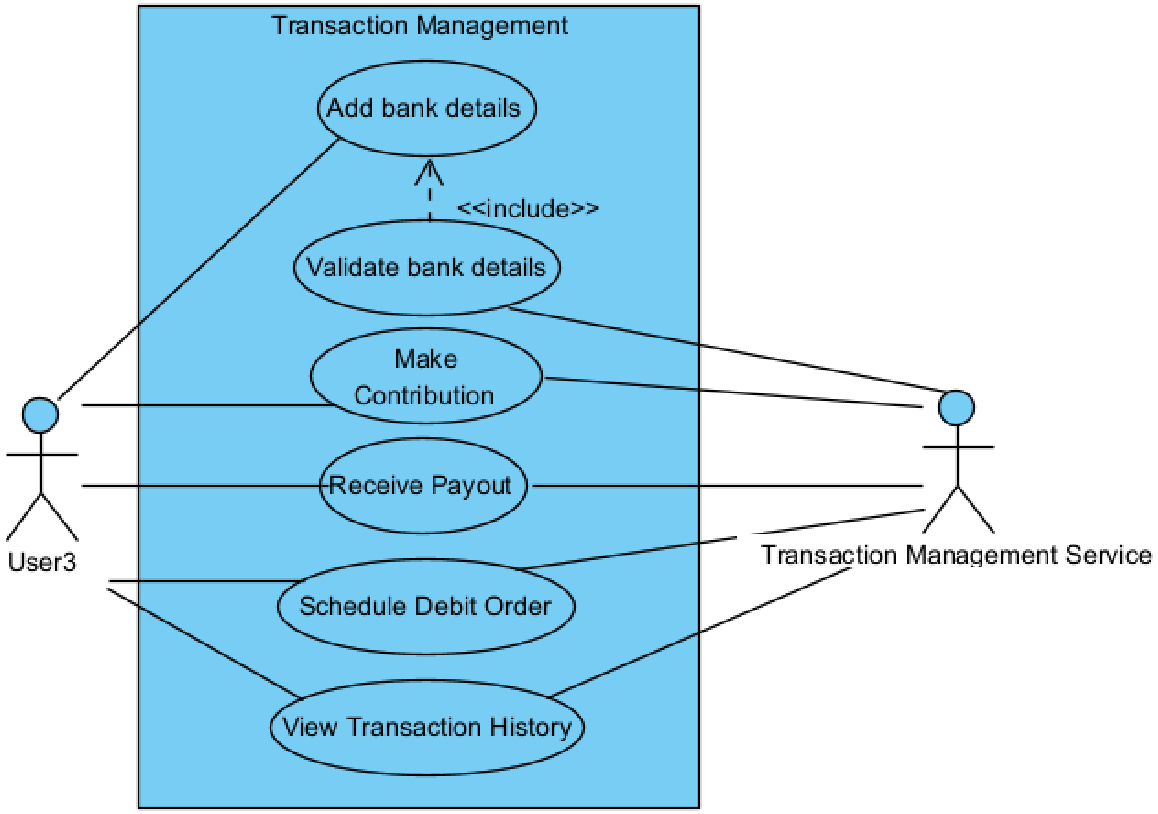
User Management



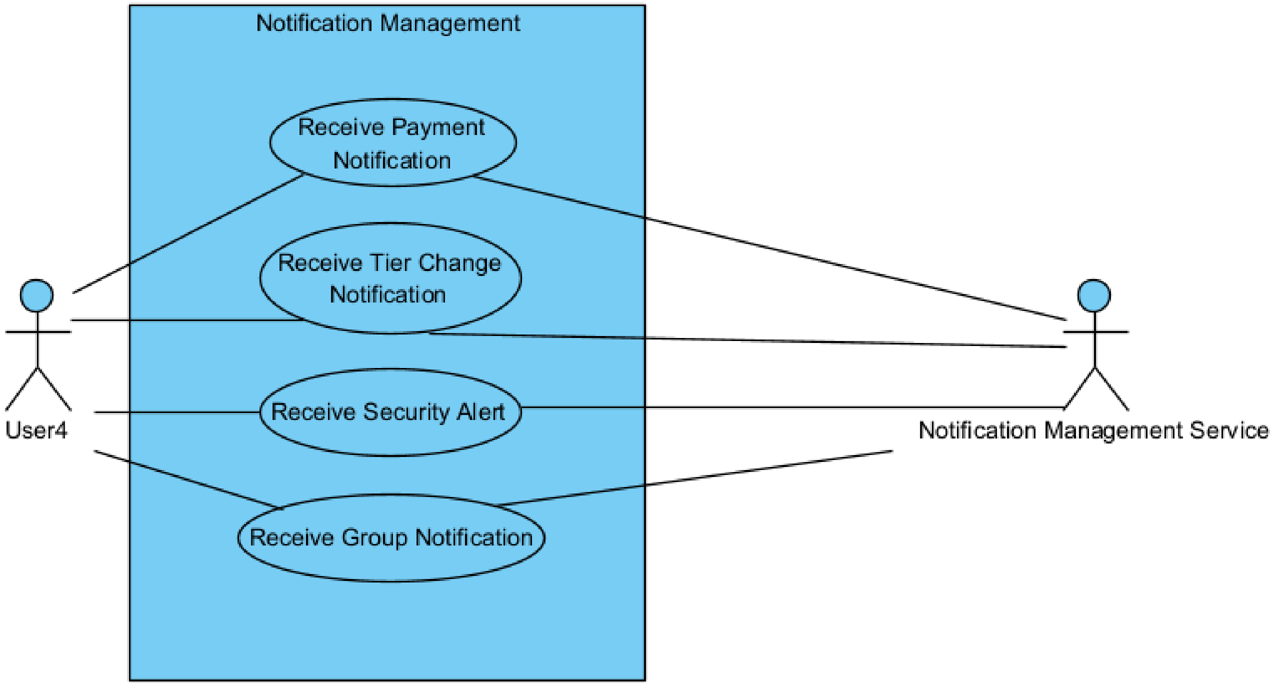
Group Management



Transaction Management



Notification Management


Notification Management
Receive Payment Notification
Receive Tier Change Notification
Receive Security Alert
Receive Group Notification
User4
Notification Management Service

ARCHITECTURAL REQUIREMENTS

INTRODUCTION

This document outlines the main quality requirements of the Stockfellow system and the architectural strategies and patterns used to address them. In addition the architectural constraints are stated as well as how the constraints are satisfied using the select architectural design.

ARCHITECTURAL DESIGN STRATEGY

The process to for designing our system architecture was influenced by factors such as:

- Our functional requirements and patterns/technologies required to meet them
- The vision and requirements and suggestions of our client
- The constraints of both the project as a whole and those related specifically to its architecture and design

QUALITY REQUIREMENTS

The quality requirements of our system are used to design an architecture to satisfy a variety of specific constraints and system needs, that are broadly satisfied by the quality requirements explained below.
Our team identified the following primary quality requirements of the Stockfellow system:

- **Security**
- **Consistency**
- **Usability**
- **Modularity**
- **Scalability**

3.1 Security:

Security was identified as the most important quality attribute of the Stockfellow system. As a fintech platform handling sensitive user information including ID numbers, contact details, and banking information, the system must protect against both data breaches and fraudulent activities.

To achieve this the system shall implement:

- OAuth security with JWT tokens
- RBAC(Role based access control)
- API rate limiting
- Multi-Factor Authentication
- Biometric Authentication

Source	User/API client request
Stimulus	Wants to access sensitive financial data and perform transactions securely
Response	• Authenticate and authorise all requests • Encrypt sensitive data • Log security events • Rate limit to prevent abuse
Response Measure	• All API endpoints protected with OAuth 2.0/JWT • Zero tolerance for unauthorized data access • Security audit logs maintained • Rate limiting prevents more than 100 requests per minute per user
Environment	Production environment with external threats and malicious actors
Artifact	Security components including Keycloak IAM, API Gateway with rate limiting, and database encryption

3.2 Consistency:

For financial transactions, ACID compliance is favored over speed to ensure that all financial transactions are correct and reliable for all users. Data integrity is paramount in financial systems.

- PostgreSQL for ACID-compliant transactions
- Database level constraints
- Synchronous processing of financial transactions to ensure data integrity

Source	User/System initiating financial transaction
Stimulus	Wants to perform financial transactions with guaranteed consistency
Response	• Ensure ACID compliance for all transactions • Maintain data integrity across services with single source of authority for critical data
Response Measure	• 100% transaction consistency • Database constraints prevent invalid states • Transaction rollback capability • Data synchronization across microservices
Environment	High-volume transaction processing with concurrent users
Artifact	PostgreSQL databases, transaction management components, and data validation services

2.3 Usability:

The system should be simple to understand and interact with for all users, especially those who lack a technologically savvy background. Clear navigation and helpful guidance are essential.

- Simple Intuitive UI/UX
- Clear help documentation and messages/notifications

Source	Mobile app users
Stimulus	Want to intuitively interact with the financial platform
Response	• Provide simple, intuitive UI/UX • Clear help documentation • Helpful error messages and notifications • Accessible design patterns
Response Measure	• User satisfaction through usability testing • Task completion rates above 90% • Help documentation accessibility • Mobile-responsive design compliance
Environment	Mobile application interface with diverse user base
Artifact	Mobile app UI screens, forms, buttons, navigation menus, notifications, and help documentation

2.4 Modularity:

The system should be modular to allow for separate development and updates of components. This approach enables flexibility for future platform support such as USSD integration.

- Domain-driven design with boundaries for each service
- Independent databases for loose coupling

Source	Development team
Stimulus	Wants to develop, deploy, and scale system components independently
Response	• Implement domain-driven design with clear service boundaries • Provide independent databases for loose coupling • Enable separate deployment pipelines
Response Measure	• Services can be deployed independently • Domain boundaries clearly defined • Database per service pattern implemented • API contracts maintained for service communication
Environment	Development and deployment stages of the system
Artifact	Individual microservices, service APIs, and deployment configurations

2.5 Scalability:

The system should handle thousands of concurrent users and perform under load, particularly when large batches of transactions are processed.

- Horizontal scaling of individual microservices
- Caching strategies for frequently access data

Source	High user load/peak usage periods
Stimulus	System must handle thousands of concurrent users and transaction batches
Response	• Horizontal scaling of microservices • Implement caching strategies • Load balancing across instances
Response Measure	• Support for 10,000+ concurrent users • Response time under 2 seconds for 90% of requests • Auto-scaling based on load metrics (subject to hardware constraints) • Cache hit ratio above 80% for frequently accessed data
Environment	Production environment during peak usage and high transaction volumes
Artifact	Load balancers, caching layers, and horizontally scalable microservice instances

ARCHITECTURAL STRATEGIES

The following list of architectural strategies will be used to address the quality requirements as mentioned above:

1. **Microservices Architecture**
2. **Event-Driven Architecture**
3. **API Gateway Pattern**
4. **Database per Service**
5. **Security-First Design**

Microservices Architecture

To achieve **modularity** and **scalability**, the system employs a microservices architecture. This pattern allows multiple instances of individual services to be deployed to handle increased load. By decomposing the system into focused, domain-specific services, the architecture enables independent development, deployment, and scaling of components.

The quality requirement of **modularity** is addressed with microservices, as each service is decoupled from others, meaning adding or changing functionality requires only modifications to specific services. This allows for easier maintenance, testing, and independent scaling of system components.

Event-Driven Architecture

To improve **scalability** and **consistency** while maintaining loose coupling, the Event-Driven Architecture pattern is implemented using ActiveMQ message queues. This pattern allows asynchronous communication between services while ensuring reliable message delivery and event ordering for critical financial operations.

API Gateway Pattern

Security is addressed using the API Gateway pattern, which provides a single entry point for all client requests. The gateway handles authentication, authorization, rate limiting, and request routing, ensuring consistent security policies across all services.

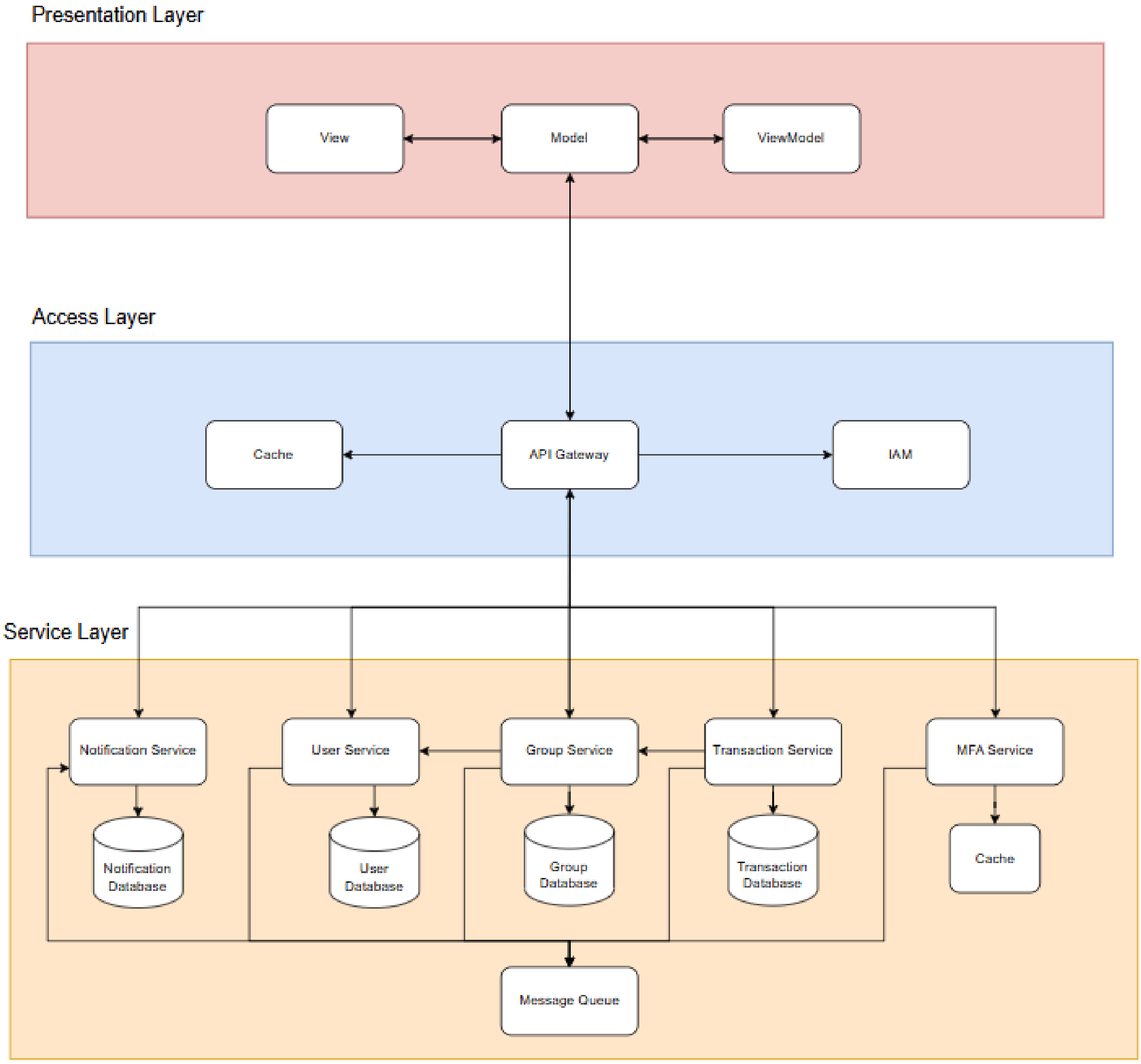
Database per Service

The Database per Service pattern supports **modularity** and **consistency** by ensuring that each microservice owns its data and maintains clear boundaries. This pattern prevents tight coupling between services while allowing each service to choose the most appropriate data storage technology for its needs.

Security-First Design

To address the critical **security** requirements, a security-first design approach is employed throughout the architecture. This includes OAuth 2.0/JWT authentication, role-based access control (RBAC), data encryption, and comprehensive audit logging.

SYSTEM ARCHITECTURAL DIAGRAM



ARCHITECTURAL DESIGN PATTERN

- **Event-Driven Architecture**: Using ActiveMQ for async communication
- **Database per Service**: Each microservice owns its data
- **API Gateway Pattern**: Single entry point for clients

ARCHITECTURAL CONSTRAINTS

- **Client Constraints:**
  - **Microservice Pattern**(suggested)
  - **Event-Driven Architecture via message queue**(suggested)
  - **Keycloak IAM**(suggested)
- **Legal:**
  - **POPIA** compliance

TECHNOLOGY CONSIDERATIONS

- **Frontend:**
- React: enables simultaneous development of web and mobile UI
- **Backend:**
- Node.js: familiar, faster to develop
- Java: more mature ecosystem particularly for fintech and IAM systems. More robust frameworks for microservice architecture (Spring Boot)
- **Message Queue(Event-driven):**
- ActiveMQ: better integration with Java specific programs, steeper learning curve and less resources
- RabbitMQ: Smaller learning curve, less tailored to Java specifically
- **IAM**
- Keycloak: default for frameworks like spring boot and rich ecosystem and support
- **Payment Gateway Integration**
- Ozow
- Payfast
- DirectDebit

SERVICE CONTRACTS

User Registration

Service Contract Name: registerUser

Parameters: { firstName: string, lastName: string, email: string, phone: string, password: string, nationalId: string }

Pre-conditions:

- The user must not have an existing account
- All information must be provided and validated
- They must provide a valid and verifiable South African ID

Post-conditions:

- Verification process initiated via Keycloack

Actors: User

Scenario: The user accesses the registration page, enters their details, and submits the form. The verification process is initiated.

User Login

Service Contract Name: loginUser

Parameters: { email: string, password: string }

Pre-conditions:

- The user must have an existing account
- Credentials must be provided and correct

Post-conditions:

- Multi-factor authentication initiated
- User session begins
- User is navigated to dashboard

Actors: User

Scenario: The user accesses the login page, enters their details, the system verifies them and establishes a session.

Create Stokvel Group

Service Contract Name: createStokvelGroup

Parameters: { name: string, description: string, creatorId: string, private: boolean }

Pre-conditions:

- The user must have an existing account
- The user must have a good enough financial tier

Post-conditions:

- Contribution amount and frequency is set.
- New group created with rules and parameters.
- Creator of the group is set as admin.

Actors: Verified User

Scenario: The user accesses the stokvels page and creates a new stokvel group with specified parameters.

Join Stokvel Group

Service Contract Name: joinStokvelGroup

Parameters: { userId: string, groupId: string }

Pre-conditions:

- The user must have a good enough financial tier to meet the group requirements.
- Group must have space for a new user.

Post-conditions:

- Group admin is notified.
- If admin approves, the user is added to list of group members.

Actors: Verified User

Scenario: The user accesses the stokvels page, requests to join a group.

TECHNOLOGY REQUIREMENTS

- **Frontend:** React.js with Tailwind CSS for a responsive, intuitive UI accessible on web and mobile devices, can be catered to users on mid-range smartphones and low-bandwidth networks, critical for rural accessibility.
- **Backend:** Node.js with Express.js for a fast, scalable server-side application.
- **Database**: PostgreSQL for structured financial data (e.g., transactions) and MongoDB for flexible user and group data.
- Authentication: Keycloak for secure identity management with OAuth2 and MFA support.
- **Messaging:** ActiveMQ for asynchronous event-driven processing (e.g., payments, notifications).
- Document Storage: Alfresco for secure management of verification documents.
- **API**: RESTful API for integration with banks, payment gateways, and possibly blockchain services.
- Search and Analytics: ElasticSearch enables fast querying and analytics for real-time reporting
- **AI Frameworks:** TensorFlow powers fraud detection and credit scoring with robust machine learning capabilities.
- **Hosting:** AWS Free Tier, scalable and cost-effective for a student project, with security features for financial data.
- **Version Control:** GitHub, essential for collaborative development and code management.
- **CI/CD:** GitHub Actions, automates testing and deployment, ensuring quality and efficiency along with aiding the dev-ops process
- **Testing**: Jest and Cypress, provides comprehensive unit, integration and end-to-end testing to maintain reliability.