DEV OPPS

ARCHITECTURAL REQUIREMENT SPECIFICATION

DEMO 3

STOCKFELLOW

BRIGHT BYTE ENTERPRISES

TABLE OF CONTENTS

Table of contents 1

Introduction 2

Architectural Design Strategy 2

Quality Requirements 2

- Security 3
- Consistency 4
- Usability 11
- Modularity 11
- Scalability 12

Architectural Styles/Patterns 12

Architectural Patterns 13

Design Patterns 13

Architectural Strategies 14

Architectural Diagram 21

Chosen Technologies 23

Architectural Constraints

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

Our team identified the following primary quality requirements of the Stockfellow system:

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.

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

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

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

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	Simple intuitive UI, with support and documentation for non-tech-savvy users

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

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 STYLES/PATTERNS

On a high-level, our system is structurally organized by the following architectural styles and patterns:

- Microservices Architecture
- Event-Driven Architecture
- Layered Architecture
- MVVM Architecture

Microservice 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.

Layered Architecture

The StockFellow system employs a three-tier layered architecture promoting modularity and security by organizing components into distinct logical layers with clear separation of concerns.

Presentation Layer serves as the user interface tier. This layer handles user interactions, input validation, and presentation logic while remaining independent of business rules and data access concerns.

Access Layer acts as the intermediary tier, primarily implemented through the API Gateway, which manages all communication between the presentation and service layers. This layer handles cross-cutting concerns including authentication, authorization, request routing, rate limiting, and response aggregation. The access layer enforces consistent security and provides a unified interface that hides the complex underlying architecture.

Service Layer contains the core business logic and data management components, implemented as domain-specific microservices (User Service, Transaction Service, Group Service, etc.) along with their associated databases.

This layered separation ensures that changes in one layer have minimal impact on others, supporting the modularity quality requirement while establishing clear security boundaries and enabling independent development and testing of each tier.

MVVM Architecture

The Model-View-ViewModel (MVVM) architecture pattern separates the presentation layer from business logic, supporting **usability** and **modularity**. In StockFellow, React components serve as the View layer, service classes act as ViewModels handling business logic and state management, while backend microservices represent the Model layer containing data and core business rules. This separation enables independent development of UI components and business logic while maintaining clear data flow.

ARCHITECTURAL PATTERNS

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

- API Gateway Pattern
- Database per Service
- Circuit Breaker Pattern
- Service Discovery

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.

Circuit Breaker Pattern

The Circuit Breaker pattern enhances **consistency** and system resilience by preventing cascading failures when downstream services become unavailable. When a service experiences repeated failures, the circuit breaker opens, immediately returning errors instead of attempting failed calls. This protects the overall system integrity and ensures that financial transaction consistency is maintained even when individual services face issues.

Service Discovery

Service Discovery supports **scalability** and **modularity** by enabling services to locate and communicate with each other dynamically. As new service instances are deployed or scaled horizontally, the discovery mechanism automatically registers and deregisters services, allowing the system to adapt to changing loads without manual configuration changes.

DESIGN PATTERNS

The design patterns show below are use in combination to create a functional, scalable system:

- DTO Pattern
- Dependency Injection
- Factory Pattern
- Observer Pattern

DTO Pattern

Data Transfer Objects facilitate consistency and security by providing a standardized format for data exchange between services and layers. DTOs ensure that only necessary data is transmitted, reducing security exposure while maintaining data integrity across service boundaries.

Dependency Injection

Dependency Injection promotes modularity and testability by decoupling components from their dependencies. This pattern allows services to be easily tested in isolation and enables flexible configuration of different implementations for various environments.

Factory Pattern

The Factory pattern supports modularity by centralizing object creation logic and abstracting the instantiation process. This enables the system to create appropriate service instances based on runtime conditions while maintaining loose coupling between components.

Observer Pattern

The Observer pattern, implemented through event-driven communication, enhances scalability by enabling asynchronous, loosely-coupled interactions between services. Services can subscribe to relevant events without direct dependencies on event publishers.

ARCHITECTURAL TACTICS

- Event Sourcing
- Message Queuing
- Pipes and filters - Filters in API Gateway
- ORM for databases
- Ring fence resources
- Hmac from paystack

Event Sourcing

Event sourcing ensures **consistency** and provides complete audit trails by storing all state changes as immutable events. This approach enables system recovery, regulatory compliance, and maintains perfect transaction history for financial operations.

Message Queuing

Asynchronous message queuing improves **scalability** by decoupling service interactions and enabling systems to handle variable loads. Messages can be processed when resources are available, preventing system overload during peak periods.

Ring Fence Resources

Resource isolation protects **consistency** and **scalability** by preventing resource contention between different system components. Critical financial operations are guaranteed sufficient resources regardless of other system activities.

HMAC Authentication

Hash-based Message Authentication Codes enhance **security** by ensuring message integrity and authenticity in API communications. This prevents tampering and unauthorized access to sensitive financial data.

Mapping of Tactics to Quality Attributes

Quality Attribute	Supporting Tactics
Security	HMAC Authentication, Pipes and Filters (API Gateway), Ring Fence Resources
Consistency	Event Sourcing, ORM for databases, Message Queuing
Scalability	Message Queuing, Ring Fence Resources, Horizontal scaling
Modularity	ORM abstraction, Event Sourcing, Service boundaries

ARCHITECTURAL STRATEGIES

- Security-First Design
- Consistency over Availability
- Horizontal Scaling
- Loose Coupling

Security First Design

Security considerations have been integrated into every architectural decision from the start of the project, rather than being added as an afterthought. This ensures the security quality requirement by establishing security as a primary design constraint that influences all system components and interactions.

Consistency over Availability

This approach supports the consistency quality requirement by choosing strong consistency guarantees even if it means occasional system unavailability. This does not imply a disregard for availability, as we aim to maximize both consistency and availability however, where a trade-off is necessary, consistency will be prioritized

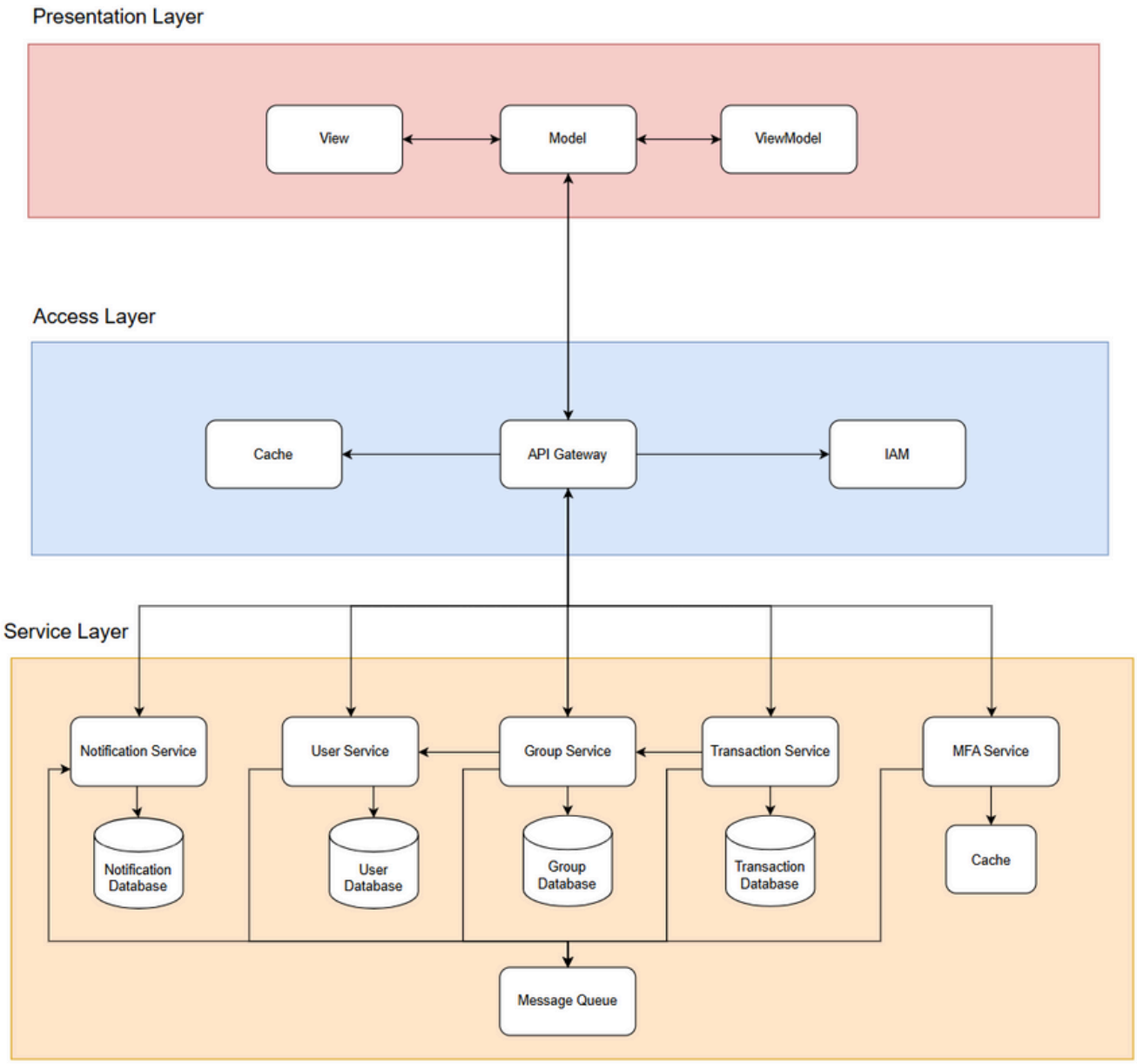
Horizontal Scaling

Horizontal Scaling addresses the scalability quality requirement by enabling the system to handle increased load through the addition of more service instances rather than upgrading individual components. This strategy involves designing services to be stateless and independently deployable, allowing multiple instances to run simultaneously.

Loose Coupling

Loose Coupling minimizes dependencies between system components, supporting modularity and scalability. This ensures that changes to one component have minimal impact on others, enabling independent development, testing, and deployment.

ARCHITECTURAL DIAGRAM



TECHNOLOGY CONSIDERATIONS

Frontend

React.js with Tailwind CSS

Pros:

- Enables faster UI development compared to Flutter's widget-based system, supporting quicker time-to-market
- Lower Bandwidth Usage
- Leverages existing JavaScript knowledge

Cons:

- Slower web-based performance for intensive operations
- Limited native device features compared to platform-specific development approaches

Flutter + Material Design

Pros:

- Single codebase for iOS, Android, and web deployment
- Built-in Material Design components ideal for financial apps
- Hot reload for faster development

Cons:

- Dart language has smaller developer pool
- Less mature ecosystem with steeper learning curve

Kotlin/Swift

Pros:

- Performance and platform optimization
- Full access to all device capabilities
- Better biometric authentication support

Cons:

- Requires separate codebases
- Higher development and maintenance costs with longer time to market

Backend

Spring Boot(Java)

Pros:

- Enterprise maturity with proven track record in fintech with robust security frameworks and extensive ecosystem, superior to Node.js for enterprise-scale financial systems
- Microservices ecosystem with better integration with chosen technologies (ActiveMQ, Keycloak) compared to Node.js
- Performance stability with multi-threaded architecture that handles CPU-intensive tasks more effectively than Node.js single-threaded model

Cons:

- Slower development cycles compared to Node.js rapid prototyping
- Higher memory footprint and startup times
- Requires Java expertise, whereas Node.js would leverage existing JavaScript knowledge

Node.js + Express

Pros:

- Consistent JavaScript in frontend and backend
- Fast development
- Large ecosystem for financial libraries

Cons:

- Less mature for enterprise level systems compared to Spring
- Memory leaks possible
- Single threaded nature limits CPU-intensive tasks
- Less compatible with other selected technologies like ActiveMQ

.NET

Pros:

- Excellent scalability
- Mature financial ecosystem
- Built-in security features

Cons:

- Learning curve is steeper with the team having limited exposure to .NET
- Smaller open-source community compared to Spring/Java

Database

PostgreSQL

Pros:

- ACID Compliance: Full transactional integrity essential for financial operations, unlike MongoDB's historical consistency limitations
- Advanced Query Capabilities: Superior complex query support and financial reporting features compared to MySQL's simpler feature set

1. Regulatory Compliance: Better audit trail and data integrity features for POPIA/GDPR compliance compared to both alternatives

Cons:

- Horizontal Scaling Complexity: More challenging to scale horizontally compared to MongoDB's distributed architecture
- Schema Rigidity: Less flexibility for evolving data models compared to MongoDB's schema-less approach
- Performance Overhead: ACID compliance introduces performance costs compared to MySQL's optimized read operations

MongoDB

Pros:

- Flexible schema for evolving requirements
- Excellent for document storage (user profiles, transactions)
- Horizontal scaling capabilities
- Good performance for read-heavy workloads

Cons:

- ACID transactions limitations (improved in recent versions)
- Memory intensive
- Less mature ecosystem for financial reporting
- Potential data consistency issues

MySQL

Pros:

- Mature and widely adopted
- Excellent performance for read operations
- Large community and ecosystem
- Lower learning curve

Cons:

- Less advanced features compared to PostgreSQL
- Weaker support for complex queries
- Replication can be complex

Cache

Redis

Pros:

- In-memory database with SQL support
- Better integration with Spring Boot
- Distributed computing capabilities
- ACID transactions support

Cons:

- More complex setup and configuration
- Higher memory requirements
- Smaller community compared to Redis

Apache Ignite

Pros:

- In-memory database with SQL support
- Better integration with Spring Boot
- Distributed computing capabilities
- ACID transactions support

Cons:

- More complex setup and configuration
- Higher memory requirements
- Smaller community compared to Redis

Hazelcast

Pros:

- Strong Java ecosystem integration
- Built-in clustering and distribution
- Good for microservices architectures

Cons:

- Resource intensive
- Complex configuration for optimal performance

IAM (Identity and Access Management)

KeyCloak

Pros:

- Cost Effectiveness: Open-source solution with no per-user licensing costs, unlike Auth0's scaling expenses or AWS Cognito's usage-based pricing
- Data Sovereignty: Full control over user data location and infrastructure, critical for POPIA compliance compared to third-party cloud solutions
- Customization Control: Complete customization capabilities for specific fintech requirements, superior to AWS Cognito's limited configuration options

Cons:

- Infrastructure Overhead: Requires self-hosting and maintenance compared to fully managed solutions like AWS Cognito
- Feature Development: May lack some out-of-the-box features available in Auth0's enterprise offering
- Operational Complexity: Requires internal expertise for scaling and troubleshooting compared to managed alternatives' support services

AWS Cognito

Pros:

- Fully managed service (no infrastructure overhead)
- Seamless AWS ecosystem integration
- Built-in social login providers
- MFA support out of the box

Cons:

- Vendor lock-in to AWS
- Limited customization options
- Can become expensive at scale
- Less control over user data location

Auth0

Pros:

- Enterprise-grade security features
- Extensive customization options
- Rich dashboard and analytics
- Strong social login integration
- Excellent documentation and support
- Multi-factor authentication built-in

Cons:

- Costly for large user bases
- Third-party dependency
- Data residency concerns
- Feature limitations on free tier

Messaging

ActiveMQ

Pros:

- Spring Boot Integration and simplified configuration
- Provides familiar programming model for Java developers
- Moderate learning curve and resource requirements

Cons:

- Lower throughput capabilities compared to Kafka
- Smaller community and fewer learning resources
- Less advanced event streaming and replay capabilities

Apache Kafka

Pros:

- Excellent for high-throughput scenarios
- Built-in partitioning and replication
- Strong durability guarantees
- Event streaming capabilities

Cons:

- Complex setup and configuration
- Resource intensive
- Steep learning curve

RabbitMQ

Pros:

- Easy to set up and configure
- Excellent routing capabilities
- Strong community support
- Good management interface

Cons:

- Single point of failure without clustering
- Lower throughput compared to Kafka
- Erlang-based (unfamiliar)

Document Processing

Python + PdfPlumber

Pros:

- Rapid development with pre-built libraries and simple syntax
- Machine learning integration for fraud detection
- Financial document specialization: financial and regulatory document processing tools,

Cons:

- Technology stack inconsistency
- Slower execution for simple document operations

Java-based Solution (Apache Tika + iText)

Pros:

- Consistent technology stack with Spring Boot backend
- Excellent PDF manipulation features
- Good integration with existing Java infrastructure

Cons:

- Less flexible than Python for ML/AI tasks
- Fewer pre-built models for document analysis
- More verbose code for data processing tasks

Node.js + Sharp/Tesseract

Pros:

- Async processing suitable for high loads
- Good integration with React Native frontend
- NPM ecosystem for document tools

Cons:

- Less mature than Python for document AI
- Limited machine learning capabilities
- Fewer specialized financial document tools

Deployment

Docker

Pros:

- Simple container orchestration and deployment
- Lower learning curve for development teams
- Cost-effective for initial development and testing

Cons:

- Limited scaling capabilities for production workloads
- Manual management of container health and networking
- Less suitable for high-availability requirements

Kubernetes (EKS/GKE)

Pros:

- Excellent auto-scaling for microservices architecture
- Production-grade orchestration with self-healing capabilities
- Industry standard for enterprise container management

Cons:

- Steep learning curve and operational complexity
- Higher infrastructure costs and resource overhead
- Requires specialized DevOps expertise for management

Docker Swarm

Pros:

- Easier setup and management compared to Kubernetes
- Native Docker integration with familiar commands
- Good balance between simplicity and orchestration features

Cons:

- Limited ecosystem compared to Kubernetes
- Fewer advanced features for complex deployments
- Smaller community and less third-party tool support

Payment Service Provider

Paystack

Pros:

- Competitive transaction fees suitable for budget-conscious stokvel operations
- Comprehensive sandbox environment and clear documentation
- Strong presence in African markets with local payment method support

Cons:

- Fewer alternative payment methods than international providers
- Limited availability outside African markets for future expansion

Ozow

Pros:

- Most competitive transaction fees for high-volume stokvel transactions
- South African specialized banking integration and regulatory compliance

Cons:

- Registration requirements and limited sandbox access
- Scalability Concerns: Smaller provider with smaller infrastructure

Payfast

Pros:

- Easy sandbox registration and straightforward API implementation
- Flexible test environment without funding requirements

Cons:

- API quality issues
- Frequent bugs and inconsistent performance
- Inadequate technical resources compared to competitors' comprehensive guides

ARCHITECTURAL CONSTRAINTS

StockFellow's architecture is subject to the following constraints that were taken into consideration when choosing patterns, and technologies to be used:

Client Specification

Certain aspects of the system were specified in the client specification of the project such as the desire for the system to be architectured using microservices. Whilst most of these details in the specification were to act as a guideline, our team aimed to meet the brief where possible.

Budget

A primary constraint of the system is the lack of a specific budget. This will influence the reliance on open source integration such as Keycloak. In addition, the deployment environment will need to be carefully select to ensure the system runs on hardware that can handle the desired load while keeping costs to a minimum.

Team Expertise and Time

Do to the limited time available to develop the system, the familiarity and relative learning curve of software and languages will be taken into consideration. This ensures that maximum time is spent developing the system and limit time spent learning new architectures and languages.

Compliance

StockFellow handles and stores sensitive information/data of it's customers. For this reason the system must comply with the following regulations:

- POPIA (Protection of Personal Information Act)
- AML (Anti-Money Laundering)
- GDPR (General Data Protection Regulation)