EUSBC Technical Design Specification (TDS)

Version: 1.0

Date: 14/01/2026

Author: Egbie Uku

Technical Design Specification

EUSBC Virtual Bank

1. Introduction

1.1 Purpose

This document describes the technical architecture and design decisions for the **EUSBC Virtual Bank** application. It explains how the system is structured internally and how it implements the functional and non-functional requirements defined in the Software Requirements Specification (SRS).

The purpose of this document is to provide developers, reviewers, and maintainers with a clear understanding of the system's architecture, technology choices, module responsibilities, and internal workflows to support development, testing, and future enhancement.

1.2 Scope

This document focuses on:

- System architecture and design patterns
- Technology stack and supporting libraries
- Backend and frontend structure
- Domain modules and their responsibilities
- Transaction and business logic orchestration
- Background processing and automation
- Project structure as an implementation reference

This document does **not** redefine system requirements and does **not** replace the SRS. It assumes familiarity with the functional and non-functional requirements described therein.

1.3 Intended Audience

This document is intended for:

- Software developers
- System designers and architects
- Security engineers
- Technical reviewers
- Future maintainers of the system
- Independent learners and hobbyists seeking to understand the system design

2. System Architecture Overview

The EUSBC Virtual Bank is a **web-based application** implemented using the Django framework. The system follows a **modular monolithic architecture**, where functionality is separated into domain-focused modules while remaining within a single deployable application.

2.1 Architectural Pattern

The backend follows Django's **Model–View–Template (MVT)** architectural pattern:

- **Models** represent real-world domain entities such as users, wallets, bank accounts, cards, and transactions.
- **Views** handle HTTP requests, orchestrate application logic, and return responses.
- **Templates** render server-side HTML views using reusable layouts and partials.

Django’s Object-Relational Mapper (ORM) is used to define and manage database models, allowing database schemas to be expressed as Python classes rather than raw SQL. Schema changes are managed through Django’s migration system to ensure consistency between code and database state.

2.2 Modular Design

The system is organised into **domain-specific modules**, each responsible for a distinct area of functionality, such as authentication, user profiles, wallets, bank accounts, cards, transactions, dashboards, and notifications.

This modular approach:

- Improves separation of concerns
- Enhances maintainability
- Allows individual domains to evolve independently
- Supports future feature expansion without large-scale refactoring

2.3 Business Logic and Transaction Handling

Core financial operations such as adding funds, transferring balances, and deducting funds are centralised within a dedicated **Transaction Service layer**. This service acts as a shared domain service used by wallet, bank account, and card modules to ensure that all monetary operations follow consistent validation, security checks, and balance update rules.

Critical financial operations are designed to execute within **atomic database transactions** to ensure:

- Consistency of balances
- Prevention of partial updates
- Protection against race conditions that could result in invalid states

Models may enforce basic domain invariants, while complex transaction orchestration is handled by dedicated service modules to keep views thin and logic reusable.

2.4 Background Processing

The system supports asynchronous and scheduled operations through a background task processor. This enables features such as:

- Scheduled and recurring transfers
- Deferred processing of time-based operations
- Future extensibility for interest calculation and notifications

Background tasks are decoupled from request–response cycles to improve performance and reliability.

3. Technology Stack & Rationale

This section describes the technologies used to implement the EUSBC Virtual Bank application and the rationale behind each choice.

3.1 Programming Language

Python is used as the primary programming language due to its readability, mature ecosystem, and strong support for web development and financial applications. Python enables rapid development while maintaining code clarity and long-term maintainability.

3.2 Backend Framework

Django is used as the core backend framework. Django provides a robust and secure foundation for building data-driven web applications, including:

- A built-in authentication system
- An object-relational mapper (ORM)
- A powerful migration framework
- Strong security features (e.g. CSRF protection, password hashing)

Django's architecture aligns well with the system's requirements for transactional consistency, user authentication, and structured domain modelling.

3.3 Background Task Processing

Django-Q is used to handle background tasks and scheduled operations. This allows the system to process time-based and asynchronous workflows outside of the main request–response cycle.

Typical use cases include:

- Scheduled or recurring fund transfers
- Deferred processing of system events
- Future extensibility for interest calculation and automated notifications
- This approach improves system responsiveness and scalability.

3.4 Frontend Technologies

The frontend is implemented using HTML, CSS, and JavaScript, with server-side rendering provided by Django templates.

- **HTML** is used for structured markup and layout
- **CSS** provides global and component-level styling
- **JavaScript** enables client-side interactivity and asynchronous requests

The Fetch API is used to perform asynchronous operations where dynamic updates are required without full page reloads.

3.5 Styling and Assets

Font Awesome is used for icons and visual indicators. The library is downloaded and bundled directly within the application rather than loaded via a CDN to ensure deterministic builds and reduce reliance on external services.

Google Fonts, such as **Poppins**, are used to enhance visual consistency and readability across the user interface.

3.6 Security and Authentication Libraries

The application integrates **django-2fa-recovery-codes** to support two-factor authentication recovery workflows. This enhances account security while providing users with safe fallback mechanisms should they forget their login password..

Email-based communication (such as verification and notifications) is handled through a dedicated library **EmailSender,** a library that allows emails to be sent e.g verification email, changed password confirmation, etc.

4. Backend Design

This section describes the internal backend design of the EUSBC Virtual Bank application, including how data is modelled, how business logic is organised, and how backend components interact to ensure correctness, security, and maintainability.

4.1 Backend Architecture

The backend is implemented using the Django framework and follows a layered design built on top of Django's Model–View–Template (MVT) pattern.

At a high level, the backend is structured into:

- **Domain models**, representing core business entities
- **Views**, handling HTTP requests and coordinating workflows
- **Forms**, managing input validation and user-submitted data
- **Service and utility layers**, encapsulating complex business and transaction logic

This layered approach ensures a clear separation of concerns and supports long-term maintainability.

4.2 Data Modelling and Persistence

Django's Object-Relational Mapper (ORM) is used to define all persistent entities within the system. Each model represents a real-world concept within the banking domain, such as users, wallets, bank accounts, cards, and transaction logs.

The ORM provides:

- Declarative model definitions using Python classes
- Automatic database table generation
- Relationship management (e.g. one-to-one and one-to-many)
- Database-agnostic persistence

Schema changes are managed using Django's migration framework, ensuring that model changes remain synchronised with the underlying database schema across environments.

4.3 Model Responsibilities and Domain Invariants

Models are responsible for representing domain entities and enforcing **basic domain invariants**, such as:

- Ownership relationships between users and financial entities
- Structural constraints (e.g. one wallet per user)
- Field-level validation and consistency rules

Models may include small, entity-specific helper methods where appropriate. However, **complex business workflows and financial transactions are intentionally not implemented directly within views or scattered across models**.

4.4 Business Logic and Service Layer

To maintain consistency and reduce duplication, complex financial operations are centralised within dedicated **service and utility modules**. These services encapsulate business rules and coordinate interactions between multiple domain entities.

Examples of responsibilities handled by the service layer include:

- Adding and deducting funds
- Transferring balances between wallets, bank accounts, and cards
- Validating transaction amounts and account states
- Enforcing security checks such as PIN verification

By routing all monetary operations through a shared service layer, the system ensures that:

- Business rules are applied consistently
- Views remain thin and focused on request handling
- Transaction logic can be reused and tested independently

4.5 Transactional Consistency and Atomic Operations

All balance-modifying operations are designed to execute within **atomic database transactions**. This guarantees that financial operations either complete fully or fail without leaving the system in an inconsistent state.

This design prevents:

- Partial balance updates
- Negative balances caused by race conditions
- Inconsistent transaction histories

Atomic transactions are a critical requirement for systems that manage financial data, even when operating in a simulated environment.

4.6 Validation and Error Handling

Input validation is performed at multiple layers to ensure system robustness:

- Forms validate user-submitted data
- Services validate business constraints and entity states
- Transactions validate limits and sufficient balances before execution

Errors are handled gracefully and communicated clearly to users, while failures are recorded for auditing and diagnostic purposes.

4.7 Audit Logging

All significant actions, particularly authentication events and financial operations, are recorded in a dedicated transaction log. This supports:

- Traceability of system actions
- Debugging and issue investigation
- Audit and reporting requirements

Transaction logs are designed to be immutable records of system events and are not directly modifiable by users.

5. Domain Modules

This section describes the core domain modules of the EUSBC Virtual Bank application. Each module is responsible for a specific functional area and encapsulates related models, views, forms, templates, and supporting logic.

The system is organised by **business domain rather than technical layer**, allowing functionality to remain cohesive and easier to maintain as the application grows.

5.1 Authentication Module

Responsibility

The Authentication module manages user authentication, registration, and password lifecycle operations. It ensures that only authenticated users can access protected system functionality. This module acts as the entry point for all user access into the system.

Key Features

- User login and logout
- User registration
- Password reset and password change workflows
- Authentication-specific base templates

Components

- Templates:
    - `login.html`
    - `register.html`
    - `new_password.html`
    - `change_password.html`
- Views handling authentication-related requests
- Integration with Django's authentication system
- Support for two-factor authentication recovery codes

5.2 User and Profile Module

Responsibility
The User module manages user identity, profiles, and account-related operations that are not directly tied to financial transactions.

Key Features

- Custom user model
- User profile creation and updates
- Profile-related forms and views
- User-specific URLs and access control

Components

- Models: Custom User model, Profile model
- Forms: Profile update forms
- Views: Profile display and modification
- URLs: `/profile/<username>/id/`

5.3 Wallet Module

Responsibility
The Wallet module manages user wallets and wallet balance operations.

Key Features

- Automatic wallet association per user
- Wallet balance management
- Adding funds to a wallet

- Serving as a source or destination for transfers

Components

- Models: Wallet
- Views: Wallet funding and display
  - `add_funds_wallet()`
- URLs:
  - `/wallet/add/fund/`
  -

5.4 Bank Module

Responsibility
The Bank module manages user bank accounts and bank-related financial operations.

Key Features

- Bank account funding
- Transfers between bank accounts and wallets
- Transfers between bank accounts and cards
- Enforcement of transaction rules and limits

Components

- Models: Bank Account
- Forms:
  - `fund_form`
  - `transfer_funds_form`
- Views handling bank-related actions

- URLs:
  - `/fund/<username>/account/`
  - `/fund/<username>/wallet/`
  - `/transfer/from/bank/to/wallet/<bank_id>/`
  - `/transfer/from/bank/to/card/<bank_id>/`

5.5 Card Module

Responsibility

The Card module manages credit card lifecycle and card-based transactions.

Key Features

- Manual card creation
- Card blocking and unblocking
- Card funding
- Card deletion with balance constraints
- Card-based transfers

Components

- Models: Card
- Forms:
  - `add_new_card_form.html`
  - `remove_card_form.html`
- Views: Card management and transfer handling
- URLs:
  - `/card/<username>/add/new/`

    - `/card/<username>/block/`
    - `/card/<username>/transfer/`
    - `/card/<username>/delete/`

5.6 Transaction Services Module

Responsibility
The Transaction Services module encapsulates all business logic related to monetary operations. It acts as a shared service layer used by the Wallet, Bank, and Card modules.

Key Features

- Centralised handling of all balance mutations
- Validation of transaction amounts and entity states
- Coordination of multi-entity transfers
- Enforcement of transactional consistency

Components

- Utility functions such as:
    - `fund_account`
    - `fund_wallet`
    - `transfer_from_bank_to_wallet`
    - `transfer_from_wallet_to_bank`
    - `transfer_from_bank_to_card`
    - `transfer_from_wallet_to_card`

5.7 Dashboard Module

Responsibility
The Dashboard module provides users with an aggregated view of their financial data and activity.

Key Features

- Display of balances
- Overview of recent activity
- User-specific metrics

Components

- Models: Dashboard-related data
- Forms: `dashboard_form`
- Views: Dashboard rendering logic
- URLs: `/dashboard/`

5.8 Notification Module

Responsibility
The Notification module manages system and user notifications.

Key Features

- Transaction-related notifications
- Account and system event notifications
- Delivery and display of notifications

Components

- Models: Notification
- Views: Notification handling and delivery

5.9 Shared View Helpers

Responsibility
Shared view helper utilities provide reusable logic that reduces duplication across views.

Key Features

- Common validation helpers
- Shared response handling
- Utility functions used by multiple modules

These helpers support consistency and cleaner view implementations.

6. Transaction and Business Logic Layer

This section describes how financial operations are processed within the EUSBC Virtual Bank application. All monetary actions are handled through a dedicated business logic layer to ensure consistency, correctness, and maintainability.

Rather than embedding complex logic directly inside views, the application centralises transaction rules within a reusable service layer. This layer acts as the **single source of truth** for financial behaviour across the system.

6.1 Purpose of the Transaction Layer

The Transaction and Business Logic Layer is responsible for:

- Ensuring all balance updates follow the same rules
- Validating transaction inputs and system state
- Coordinating multi-entity operations (e.g. bank → wallet)
- Preventing inconsistent or partial updates
- Enforcing system constraints and assumptions

6.2 Design Approach

The application adopts a **service-oriented design** for financial logic.

Key principles:

- Views remain thin and focused on request handling
- Business rules are isolated from presentation logic
- Transaction services are reusable across multiple modules
- All monetary changes pass through a controlled interface

6.3 Transaction Services (Utility Layer)

Transaction logic is implemented using a set of dedicated service functions located in a shared utility module.These functions encapsulate the full lifecycle of a transaction, including validation, balance updates, and persistence. Each service is responsible for managing exactly one type of transaction.

Core Services

- `fund_account(username)`
- `fund_wallet(username)`
- `transfer_from_bank_to_wallet(bank_id)`
- `transfer_from_wallet_to_bank(wallet_id)`
- `transfer_from_bank_to_card(bank_id)`
- `transfer_from_wallet_to_card(wallet_id)`

6.4 Transaction Flow

This design prevents scenarios such as funds being deducted from one entity without being credited to another. A typical transaction follows a structured sequence:

1. **Authentication Check**
   The user must be authenticated before initiating any transaction.

2. **Ownership Validation**
   The system verifies that the user owns the source and destination entities involved in the transaction.

3. **Input Validation**
   - Amount must be greater than zero
   - Amount must not exceed available balance

    - Destination entity must be active and valid

4. **Business Rule Enforcement**
   System constraints are applied, such as:

    - Maximum number of cards
    - One wallet and one bank account per user
    - Blocked cards cannot receive or send funds

5. **Balance Update**
   Balances are updated in memory in a controlled manner.

6. **Persistence**
   Updated balances are saved to the database as a single logical operation.

7. **Notification Trigger**
   A notification event may be generated upon successful completion.

6.5 Atomicity and Data Consistency

Although the system operates with virtual currency, transactional integrity remains critical.

To preserve consistency:

- Balance updates are treated as atomic operations
- Transfers involving multiple entities are completed as a single logical unit
- Partial updates are avoided to prevent inconsistent states

This design prevents scenarios such as funds being deducted from one entity without being credited to another.

6.6 Validation Rules

The following validation rules apply to all financial operations:

- Transaction amounts must be positive
- Source entities must have sufficient balance
- Users may only operate on entities they own
- Cards marked as blocked cannot participate in transactions
- Entities must exist and be active at the time of transaction

Validation logic is centralised within the transaction services to ensure consistency.

6.7 Error Handling Strategy

If a transaction fails at any stage:

- No balances are modified
- The user receives a clear error message
- The system remains in a consistent state

6.8 Extensibility

The Transaction and Business Logic Layer is designed to support future enhancements, including:

- Scheduled transfers
- Recurring transactions
- Multi-currency support
- External banking integrations (future versions)

By isolating business rules, these features can be added without major changes to the user interface or core modules

Models

7. Data Model Design

This section describes the core data models used within the EUSBC Virtual Bank application, their responsibilities, and the relationships between them. The data model is designed to accurately represent real-world financial entities while enforcing system constraints at the database level.

The application uses Django's Object-Relational Mapper (ORM) to define models as Python classes, which are translated into relational database tables via migrations.

7.1 Design Principles

The data model is guided by the following principles:

- Clear ownership of all financial entities
- Strong relational integrity between models
- Explicit constraints enforced at the model level
- Separation of data representation and business logic
- Extensibility for future features

Each model represents a single responsibility and maps directly to a real-world concept within the banking domain.

7.2 User Model

This table summarises the fields in the `User` model, including their types, character lengths, default values, and behaviour notes.

User Model					
Fields	**Char Length**	**Type**	**Required**	**Is unique**	**Notes**
id	N/A	Integer	Yes	N/A	Automatically incremented
username	80	String	Yes	Yes	The username associated with the user model
email	200	email	Yes	Yes	The email associated with the user model
is_staff	N/A	Boolean	Yes	N/A	Determines if the user is a staff member
is_superuser	N/A	Boolean	Yes	N/A	Determines if the user is a superuser
is_admin	N/A	Boolean	Yes	N/A	Determines if the user is an admin
created_on	N/A	Datetime	No	N/A	Automatically created when the model is created

User Model					
last_updated	N/A	Datetime	No	N/A	Automatically updated whenever the model is saved
last_login	N/A	Datetime	No	N/A	Automatically updated when the model is saved