Wallet API – Technical Documentation

1. Project Overview

This project implements a Wallet API that allows users to:

Register and authenticate

Manage their wallet balance

Perform peer-to-peer (P2P) transfers

Purchase services

Track transaction history

Receive notifications for every financial operation

The system focuses on:

Clean code

Correct business logic

Security

Safe handling of financial transactions

2. Main Features Implemented Authentication & Authorization

Token-based authentication using Laravel Sanctum.

Role-based access control (user, admin).

Authorization handled using Laravel Policies for:

Transactions (confirm, cancel)

Services (create, update, delete, restore)

This approach was chosen because it is:

Clear and business-oriented

Easy to extend

Keeps authorization logic outside controllers

Wallet Management

A wallet is automatically created for each user upon registration using an Event/Observer.

Initial balance is set to 0.

Wallet balance updates use atomic database operations (increment / decrement).

Transactions System

All financial operations are stored in the transactions table with:

type: charge, transfer, service_purchase

status: pending, completed, cancelled

Operations supported:

Charge wallet

P2P transfer (pending → confirm / cancel)

Service purchase

Critical operations are wrapped in DB::transaction() to ensure:

Atomicity

Data consistency

Rollback on failure

Wallet rows are locked using lockForUpdate() during transfers to prevent race conditions.

Services Management

CRUD operations for services.

Soft deletes implemented to allow restore.

Only authorized users (admin) can manage services.

Users can purchase services only if they have sufficient balance.

Transaction History

Paginated transaction history.

Supports filters:

type

status

date range

Users can only view their own transactions (sender or receiver).

Notifications

In-app notifications using Laravel Notification system.

Triggered after:

Wallet charge

Transfer confirm/cancel

Service purchase

Notification data stored in notifications table (polymorphic).

Optional email notifications implemented using Jobs and Queue (if enabled).

3. **Database Design Decisions**

Entities:

Users

Wallets (1:1 with users)

Transactions (linked to sender and receiver)

Services

Notifications (polymorphic)

Key decisions:

Enum fields for type and status to enforce valid states.

Polymorphic relation (reference_id, reference_type) for future extensibility.

Indexes on foreign keys for performance.

Soft deletes for services.

4. **Security Considerations**

CSRF protection enabled.

SQL Injection prevention via Eloquent ORM.

Authorization enforced using Policies.

Input validation via Form Requests.

Idempotency handling to avoid duplicate financial operations.

Database transactions used for all balance updates.

5. Code Structure

Controllers: Handle HTTP requests only.

Service classes: Handle business logic (transfer, purchase).

Policies: Handle authorization logic.

Form Requests: Handle validation and authorization.

Notifications: Handle user alerts.

Jobs & Queue: Handle email notifications asynchronously.

This separation improves:

Maintainability

Readability

Testability

6. API Documentation

A Postman collection is provided containing:

All endpoints

Example requests and responses

Authentication headers

Error cases

The collection can be imported directly into Postman.

7. Main Design Decisions Summary

Used Policies instead of token abilities for authorization.

Used database transactions for financial safety.

Implemented wallet creation via Observer/Event.

Used polymorphic relations for future expansion.

Focused on clean business logic instead of extra features.

8. Conclusion

This project demonstrates:

Proper handling of financial business logic

Secure API design

Clean architecture

Attention to edge cases and consistency

The implementation prioritizes correctness and clarity over feature quantity, as required by the task.