Technical Assessment: Atomic Wallet & Transaction System

Overview

The goal of this assessment is to build a secure, thread-safe internal credit transfer system. This task evaluates your proficiency with Django's ORM, database integrity, and asynchronous task processing.

Deadline: Monday 02:00 PM PKT

Target Stack: Django, Django REST Framework, PostgreSQL, Celery, Redis.

1. Functional Requirements

A. Data Modeling

- **Wallet:** Create a model linked to the User that maintains a balance.
- **Transaction:** Create an audit log model to record the sender, receiver, amount, and timestamp for every successful transfer.

B. The Transfer API

Build a DRF-based endpoint (POST /api/wallet/transfer/) that accepts:

- receiver_id (Integer)
- amount (Decimal)

Business Rules:

- Transfers must be **Atomic**: If any part of the process fails (e.g., logging the transaction), the balance changes must roll back.
- Transfers must be **Thread-Safe**: The system must prevent "double-spending." If a user with $10 attempts two simultaneous $10 transfers, only one must succeed.
- Balances must never be negative.

C. Background Processing (Celery)

- Upon a successful transfer, trigger an asynchronous task to generate a simple "Transaction Receipt" (text or PDF).
- The receipt should be saved to the media storage or a specific directory for audit purposes.

2. Technical Expectations

- **Database Integrity:** Use appropriate database-level locking (e.g., select_for_update)

and transactions (transaction.atomic) to ensure data consistency.
- **Accuracy:** Use DecimalField for all currency-related data.
- **Optimization:** Ensure queries are efficient and indices are applied where necessary.
- **Testing:** Provide a test suite, specifically including a test case that simulates concurrent requests to verify the locking mechanism works.

3. Submission Guidelines

1. **Repository:** Provide a GitHub repository link.
2. **Documentation:** Include a README.md with:
    - Setup instructions (how to run migrations, Celery, and the server).
    - A brief explanation of how you handled the concurrency/race-condition problem.
3. **Environment:** Use a .env file for sensitive configurations.
4. **Prompt Engineering:** Also add prompts.txt which will list all the prompts you gave to the AI agent you used(Also mention the agent and the platform e.g. Claude Code - Opus 4.5).

Evaluation Criteria

- **Correctness:** Does it handle edge cases (insufficient funds, invalid users)?
- **Concurrency:** Does it successfully block race conditions?
- **Code Quality:** Is the code "Pythonic" and follow PEP 8 standards?
- **Async Implementation:** Is Celery configured correctly with Redis?
- **AI Augmentation:** Were your prompts atomic or molecular?