Project ATTN: Refactoring Report - Phase 3

Subject: Unifying Data Persistence and Finalizing the Core Architecture
Date: June 28, 2025
Status: Planning

1. Introduction and Goals

With the successful completion of Phase 2, the application's service layer is now fully adapted to the new, efficient data architecture. The core business logic correctly utilizes the structured Redis models, and the inefficient sweep_users cron job has been successfully decommissioned.

The final and most critical objective of this refactoring project is to permanently resolve the "Data Integrity Race Condition" identified in the initial analysis. This will be achieved by overhauling the data persistence mechanism, unifying the separate cron jobs into a single, atomic operation, and seamlessly integrating this task into the main application's lifecycle.

Goals for This Phase:

1. **Guarantee Data Integrity:** Completely eliminate the foreign key violation errors by creating a single, unified cron task that ensures user data is persisted before their associated attendance records.
2. **Simplify Architecture:** Remove the last remaining redundant cron job (sweep_attendances_task in its old form) to create a more linear, predictable, and maintainable data flow.
3. **Automate and Integrate:** Embed the new, unified persistence task directly into the main application using apscheduler, ensuring it runs automatically without requiring a separate process.

2. Planned Changes and Tasks

2.1. Unify Data Persistence Logic (cron.py)

This task will consolidate the separate, conflicting persistence jobs into a single, robust function.

- **sweep_attendances_task Function:**
    - **Current State:** Periodically saves attendance records from Redis to PostgreSQL, creating a race condition with the now-removed sweep_users_task.
    - **New State:** This function will be refactored into a new, idempotent function named unified_persistence_task. Its logic will be completely rewritten to perform the following sequence for each completed lesson found in Redis:

1. **Upsert Users First:** It will iterate through the AttendanceRecordRedis list. Using the enriched data (which now includes the student's full name), it will perform an "upsert" operation (e.g., INSERT ... ON CONFLICT DO NOTHING) for each student into the PostgreSQL Users table. This guarantees that the user record exists *before* the next step.
2. **Insert Attendance Data:** Only after all user records for the session are confirmed to exist in the database will it proceed to insert the Attendances and AttendanceRecords data.
3. **Add Robust Error Handling:** The processing for each individual lesson's data will be wrapped in a try...except block to ensure that a single corrupted record does not halt the entire persistence task.

2.2. Integrate Cron Job into Main Application (main.py)

This task ensures the new persistence logic runs reliably as part of the main application.

- **FastAPI Application Lifecycle:**
  - **Current State:** The cron jobs are designed to be run as a separate process.
  - **New State:** We will use apscheduler to run the unified_persistence_task in the background of the main application.
    - In main.py, the AsyncIOScheduler will be imported and instantiated.
    - Inside the @app.on_event("startup") event handler, the unified_persistence_task will be added to the scheduler and configured to run at a regular interval (e.g., every 5 minutes).
    - The scheduler will be started within the same startup handler, ensuring the task begins running as soon as the application is live.