Phase 1: Database Operations & API Implementation Guide

Overview

This guide addresses the backend API implementation for the business module. The business module uses a separate SQLite database (`business.db`) and requires proper CRUD operations for inventory, assets, and transactions.

Context

- The application has two databases: `personal_finance.db` and `business.db`
- Business transactions should be automatically created from inventory sales and asset purchases
- All API endpoints are currently returning sample data or have placeholder implementations

1.1 Inventory API Implementation

File: `blueprints/business/routes.py`

A. Fix GET `/api/inventory/current` endpoint

Current Issue: Returns sample data instead of database data **Required Changes**:

1. Remove the sample data generation in `loadSampleCurrentInventory()`
2. Query the `business_inventory` table for items where `is_active = 1` and `listing_status != 'sold'`
3. Return proper JSON response with all inventory fields

B. Implement POST `/api/inventory` endpoint

Purpose: Create new inventory item **Required Validations**:

- All fields required except `description` (optional)
- `sku` must be unique (auto-generate if not provided)
- `listing_price` must be greater than 0
- `category` must be from allowed list or newly added category
- Default `listing_status` to 'inventory'
- Set `date_added` to current date

C. Implement PUT `/api/inventory/<sku>` endpoint

Purpose: Update existing inventory item **Required Logic**:

1. Fetch item by SKU
2. Validate that item exists and is not sold
3. Update only provided fields
4. Update `updated_at` timestamp
5. Return success response with updated item

D. Implement DELETE `/api/inventory/<sku>` endpoint

Purpose: Delete inventory item **Required Logic**:

1. Check if item exists
2. Verify item is not sold (sold items should not be deletable)
3. Perform hard delete from database
4. Return success response

E. Implement POST `/api/inventory/<sku>/sell` endpoint

Purpose: Mark item as sold and create revenue transaction **Required Logic**:

1. Accept `sold_price` in request body (required)
2. Update inventory item:
    - Set `listing_status` = 'sold'
    - Set `sold_price` = provided price
    - Set `sold_date` = current date
3. Create business transaction:
    - `transaction_type` = 'Income'
    - `amount` = sold_price
    - `category` = 'Sales Revenue'
    - `description` = "Sale: {item brand} {item type}"
    - `date` = current date
4. Return success response with transaction ID

1.2 Asset API Implementation

File: `blueprints/business/routes.py`

A. Implement GET `/api/assets` endpoint

Purpose: Fetch all business assets **Required Logic**:

1. Query `business_assets` table
2. Include both active and disposed assets
3. Return JSON array with all asset fields

B. Implement GET `/api/assets/<id>` endpoint

Purpose: Get single asset details **Required Logic**:

1. Fetch asset by ID
2. Return 404 if not found
3. Return complete asset details

C. Implement PUT `/api/assets/<id>` endpoint

Purpose: Update asset information **Required Validations**:

- `name` required
- `asset_category` required
- `asset_type` required
- `purchase_date` required
- `purchase_price` required and > 0
- `description` optional **Required Logic**:

1. Fetch existing asset
2. Update provided fields
3. Update `updated_at` timestamp
4. Return updated asset

D. Implement POST `/api/assets/<id>/dispose` endpoint

Purpose: Mark asset as disposed **Required Logic**:

1. Accept `disposal_date` in request body
2. Update asset:
    - Set `is_active` = 0
    - Set `disposal_date` = provided date
3. Optionally accept `disposal_value`
4. Return success response

E. Implement DELETE `/api/assets/<id>` endpoint

Purpose: Delete asset **Required Logic**:

1. Check if asset exists
2. Perform hard delete
3. Return success response

1.3 Transaction API Enhancement

File: `blueprints/business/routes.py`

A. Enhance POST `/api/transactions` endpoint

Current State: Already implemented but needs validation **Required Enhancements**:

1. When creating expense transaction from asset purchase:
   - Auto-set `category` = 'Equipment & Supplies' or asset category
   - Link to asset if applicable
2. Validate required fields based on transaction type
3. Ensure `amount` is positive for expenses

1.4 Database Initialization Updates

File: `blueprints/business/utils.py`

A. Remove Sample Data

In function `initialize_business_data()`:

1. Remove the sample inventory items creation
2. Remove the sample assets creation
3. Keep only the category initialization
4. Remove sample transactions

B. Add Inventory Category Support

Required Changes:

1. Create function to add new inventory categories dynamically
2. Ensure new categories can be added via API
3. Maintain list of default categories

1.5 Database Schema Considerations

Inventory Table Updates

Ensure the `business_inventory` table has these fields:

- `listing_status` (inventory, listed, sold, kept)
- `sold_price` (nullable, set when sold)
- `sold_date` (nullable, set when sold)
- `date_added` (required, defaults to current date)

Business Transaction Table

Ensure proper linking:

- Add `source_type` field (inventory_sale, asset_purchase, manual)
- Add `source_id` field (reference to inventory SKU or asset ID)

Testing Checklist

1. Test creating inventory item with all fields
2. Test creating inventory item with only required fields
3. Test updating inventory item
4. Test deleting unsold inventory item
5. Test marking item as sold with custom price
6. Verify transaction is created when item is sold
7. Test all asset CRUD operations
8. Verify data persistence across server restarts

Error Handling Requirements

1. Return proper HTTP status codes (200, 201, 400, 404, 500)
2. Return consistent error response format: `{"success": false, "error": "message"}`
3. Log all errors to console with detailed information
4. Validate all numeric inputs
5. Sanitize string inputs to prevent SQL injection

Notes for Implementation

- All monetary values should be stored as DECIMAL(10,2)
- All dates should be stored in ISO format (YYYY-MM-DD)
- Use database transactions for operations that modify multiple tables
- Implement proper connection closing to prevent database locks
- Consider adding database indexes for frequently queried fields (SKU, status)