**ERP Lite: Mini Inventory Management System - Project Overview**

This document offers an overview of the ERP Lite project, outlining the approach taken, key technical decisions, and a high-level technical breakdown as part of the "Build Your Mini ERP Lite Module" assignment.

**1. Project Approach & Philosophy**

The development of ERP Lite was driven by a philosophy focused on **efficiency, usability, and maintainability**, particularly with future analytical needs in mind.

**1.1. Schema Design for Efficiency and Analysis**

A primary decision point was the design of the database schema. My approach focused on:

* **Normalization for Integrity:** I ensured data integrity by separating concerns into Supplier, Product, and "Transaction" tables, minimizing redundancy.
* **Less Dimensionality for Analysis:** While normalized, the schema was kept intentionally simple and focused on the core entities needed for inventory management. This "less dimensional" approach simplifies future data analysis, reporting, and potential machine learning model integration. Fewer well-defined columns make data extraction and transformation for visualization and predictive modeling easier.
* **Snapshotting Transaction Price:** I decided to include unit\_price directly in the "Transaction" table. This approach ensures that historical reports on sales value and profit are accurate, even if the product's current unit\_price changes over time. This is crucial for precise financial analysis and trend visualization.

**1.2. Iterative Development, Debugging, and Error Handling**

The development process was iterative, emphasizing rigorous testing and strong error handling. This involved:

* **Incremental Feature Building:** I implemented features one step at a time, such as adding products, then editing, then deleting, followed by transactions, and then reports.
* **Rigorous Debugging:** I addressed Flask's werkzeug.routing.exceptions.BuildError by correcting URL generation. I resolved jinja2.exceptions.UndefinedError by explicitly converting date strings to datetime objects in Python before passing them to templates. I tackled sqlite3.OperationalError by correctly quoting the Transaction table name to avoid SQL keyword conflicts.
* **Stock Management Logic:** I implemented complex stock adjustment logic for add\_transaction, edit\_transaction, and delete\_transaction routes. This required careful consideration of all scenarios, including sales reducing stock and purchases increasing it, as well as the inverse for edits and deletions to maintain accurate inventory levels.
* **Deployment-Related Issues:** I managed PermissionError during database file deletion on server restart by reordering operations to ensure a smooth development experience before transitioning to persistent deployment strategies.

**2. Key Technical Decisions**

The following technology stack and specific choices were made to meet the project objectives:

* **Backend Framework: Python Flask**
  + **Decision:** I chose Flask for its lightweight nature, simplicity, and flexibility. It allows rapid development and full control over components, with a strong community and extensive libraries like sqlite3, making it ideal for this mini ERP system.
* **Database: SQLite3**
  + **Decision:** This option is perfect for a mini system. Its file-based nature means no separate database server setup, simplifying development and initial deployment. Its SQL compliance supports powerful queries for reporting without the complexity of larger RDBMS systems in a small-scale context.
* **Frontend Styling: Bootstrap 5**
  + **Decision:** Bootstrap provided a responsive, modern, and consistent UI out of the box, significantly speeding up frontend development without needing deep CSS expertise.
* **Frontend Charting: Chart.js**
  + **Decision:** I used this powerful yet easy-to-use JavaScript library to create dynamic and interactive data visualizations directly in the browser, which perfectly suits my reporting needs.
* **AI Integration: Together AI (Llama 3.3)**
  + **Decision:** I integrated this for the optional bonus task due to its powerful language models and its free-tier availability, making advanced AI capabilities accessible without incurring costs. I specifically selected the meta-llama/Llama-3.3-70B-Instruct-Turbo-Free model for its performance in structured output generation.

**3. Technical Overview**

**3.1. Core Functionalities**

* **Dashboard:** Displays key inventory metrics and low-stock alerts.
* **Product Management:** Facilitates CRUD operations for products, including supplier associations.
* **Supplier Management:** Supports basic CRUD operations for suppliers.
* **Transaction Management:** Records purchases and sales, with automatic stock level adjustments. It includes robust edit and delete functionalities that correctly reverse and reapply stock changes.
* **Reporting:** Visualizes critical business data through various charts.
* **AI Reorder Suggestion:** Intelligently recommends reorder quantities based on historical data and current stock.

**3.2. Database Schema Highlights**

As detailed in the code documentation on Canvas, the SQLite database utilizes Supplier, Product, and "Transaction" tables.

* The unit\_price column in the Transaction table captures the price at the time of the transaction, which is vital for accurate historical financial analysis.
* ON DELETE CASCADE and ON DELETE SET NULL constraints ensure referential integrity for related data.

**3.3. Backend (app.py) Architecture**

* **Modular Design:** Helper functions like get\_all\_products and get\_transaction\_by\_id manage database interactions, keeping routes clean.
* **Context Management:** Flask's g object and teardown\_appcontext ensure proper handling of database connections.
* **Data Transformation:** I converted database date strings to datetime objects in Python to ensure proper Jinja2 template rendering.
* **AI Integration (suggest\_reorder\_quantity\_from\_ai):**
  + Constructs a detailed text prompt for the AI based on live inventory data.
  + Communicates with Together AI's chat.completions.create endpoint, requesting a json\_object response format for reliable parsing.
  + Parses the AI's JSON output for reorder\_quantity and reasoning.
  + Includes comprehensive try-except blocks to handle API errors, empty responses, and JSON decoding issues gracefully, ensuring the application remains stable even if AI responses are unexpected.

**3.4. Frontend (HTML/JS/CSS)**

* **Template Inheritance:** base.html provides a consistent layout, and other pages like dashboard.html and products.html extend it, promoting reusability and maintainability.
* **Dynamic Data Rendering:** Jinja2 templating renders dynamic data fetched from the Flask backend.
* **Interactive Charts:** reports.html utilizes Chart.js to create:
  + **Sales Over Time (Units):** Displays units sold monthly.
  + **Net Sales Value Over Time:** Shows monthly revenue from sales.
  + **Net Profit Over Time:** A crucial financial indicator, showing monthly profit/loss with dynamic coloring for immediate understanding.
  + **Top Selling Products:** A pie chart for quick product performance overview.
  + **Purchase vs. Sale Ratio:** A bar chart for transaction type distribution.
* **AJAX for AI Feature:** JavaScript fetches AI suggestions asynchronously, providing a smooth user experience without full page reloads.

**3.5. Deployment Strategy**

The application is designed for deployment on cloud platforms, with **Render.com** selected as the free-tier option.

* **Ease of Deployment:** Render's Python buildpack simplifies the process.
* **Ephemeral Database:** For the free tier, the SQLite database (database.db) is stored ephemerally, meaning data resets on service restarts due to Render's infrastructure choices for free services.
* **Production Readiness (Future):** While it is suitable for demonstration, a truly production-ready version would switch to a persistent database solution, such as PostgreSQL, which Render offers as a paid service, and manage API keys through environment variables for better security.