**ERP Lite: Mini Inventory Management System - Code Documentation  
solution :** [**https://github.com/anshuldhamankar/Inventory-Management-System-ERP-**](https://github.com/anshuldhamankar/Inventory-Management-System-ERP-)

**Video -** [**https://drive.google.com/file/d/1PVQW1cnkYJfP4CeFXt0-0O9Kma3EXpEg/view?usp=sharing**](https://drive.google.com/file/d/1PVQW1cnkYJfP4CeFXt0-0O9Kma3EXpEg/view?usp=sharing)

This document provides a detailed overview of the ERP Lite application's codebase, covering its database schema, backend logic, frontend structure, and the integration of external APIs. (Computer Store For example)

**1. Project Overview**

ERP Lite is a Flask-based web application designed for small-scale inventory management. It offers functionalities to manage products, suppliers, and transactions, along with reporting tools and an AI-powered reorder suggestion system.

**2. Database Design and Schema (SQLite3)**

The application utilizes SQLite3 as its database, which is a lightweight, file-based relational database management system. SQLite is highly efficient for embedded applications and local data storage, making it an excellent choice for a "mini" system like ERP Lite. Its file-based nature simplifies deployment and management, and its SQL compliance allows for complex queries, enabling efficient data retrieval and analysis for reporting purposes.

The database consists of three main tables: Supplier, Product, and "Transaction".

**2.1. Supplier Table**

Stores information about product suppliers.

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Type** | **Constraints** | **Description** |
| id | INTEGER | PRIMARY KEY AUTOINCREMENT | Unique identifier for the supplier. |
| name | TEXT | NOT NULL, UNIQUE | Name of the supplier. Must be unique. |
| contact\_email | TEXT |  | Email address for contacting the supplier. |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | Timestamp when the record was created. |
| updated\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | Timestamp when the record was last updated. |

**2.2. Product Table**

Stores details about individual products in the inventory.

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Type** | **Constraints** | **Description** |
| id | INTEGER | PRIMARY KEY AUTOINCREMENT | Unique identifier for the product. |
| name | TEXT | NOT NULL | Name of the product. |
| sku | TEXT | NOT NULL, UNIQUE | Stock Keeping Unit, a unique identifier for the product. |
| stock\_quantity | INTEGER | NOT NULL | Current quantity of the product in stock. |
| unit\_price | REAL | NOT NULL | Price per unit of the product. |
| supplier\_id | INTEGER | FOREIGN KEY REFERENCES Supplier (id) ON DELETE SET NULL | Foreign key linking to the Supplier table. If a supplier is deleted, this field for associated products will be set to NULL. |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | Timestamp when the product record was created. |
| updated\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | Timestamp when the product record was last updated. |

**2.3. "Transaction" Table**

Records all purchase and sale transactions. Note that "Transaction" is quoted because TRANSACTION is a reserved SQL keyword in some database systems, preventing syntax errors.

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Type** | **Constraints** | **Description** |
| id | INTEGER | PRIMARY KEY AUTOINCREMENT | Unique identifier for the transaction. |
| product\_id | INTEGER | NOT NULL, FOREIGN KEY REFERENCES Product (id) ON DELETE CASCADE | Foreign key linking to the Product table. If a product is deleted, all its associated transactions are also deleted. |
| quantity | INTEGER | NOT NULL | Quantity of product involved in the transaction. |
| type | TEXT | NOT NULL, CHECK(type IN ('purchase', 'sale')) | Type of transaction: either 'purchase' or 'sale'. |
| date | TEXT | NOT NULL | Date of the transaction (stored as interminable-MM-DD string). |
| unit\_price | REAL | NOT NULL | The unit price of the product at the time of the transaction (snapshot). This is crucial for accurate historical reporting, even if the current product price changes. |

**3. Backend Logic (app.py)**

The app.py file contains the Flask application's core logic, including database interactions, route definitions, and AI integration.

**3.1. Application Setup & Database Management**

* **app = Flask(\_\_name\_\_)**: Initializes the Flask application.
* **app.secret\_key = 'your\_secret\_key\_here'**: Essential for secure sessions and flash messages.
* **DATABASE = 'database.db'**: Defines the SQLite database file name.
* **get\_db()**: Connects to the SQLite database. It ensures a single database connection per request context and sets row\_factory to sqlite3.Row for dictionary-like access to query results.
* **close\_connection(exception)**: Closes the database connection when the application context tears down.
* **init\_db()**: Initializes the database schema. It uses CREATE TABLE IF NOT EXISTS to ensure tables are only created if they don't already exist, preserving data across server restarts. This is called once when the application starts (if \_\_name\_\_ == '\_\_main\_\_':).

**3.2. Helper Functions (Database Operations)**

* **get\_all\_products()**: Fetches all products, including their associated supplier names. Crucially, it converts created\_at and updated\_at timestamp strings from the database into Python datetime objects, enabling proper formatting in templates.
* **get\_product\_by\_id(product\_id)**: Retrieves a single product by its ID, also performing datetime object conversion for timestamps.
* **get\_all\_suppliers()**: Fetches all suppliers and calculates the count of products associated with each, useful for the Suppliers page.
* **get\_supplier\_by\_id(supplier\_id)**: Retrieves a single supplier by ID.
* **get\_all\_transactions()**: Fetches all transactions, including the product name, SKU, and current price, converting the date string to a datetime object.
* **get\_transaction\_by\_id(transaction\_id)**: Retrieves a single transaction by ID, along with relevant product details, and converts the date.

**3.3. AI Integration (suggest\_reorder\_quantity\_from\_ai)**

This function powers the AI-driven reorder suggestion feature.

* **Purpose:** To recommend an optimal quantity for a product reorder based on its current stock and recent sales/purchase trends.
* **Logic:**
  1. Retrieves current product details.
  2. Calculates total sales and purchases for the given product over the last 30 days by querying the "Transaction" table.
  3. Constructs a detailed prompt for the AI, incorporating product name, SKU, current stock, and 30-day transaction history. It also specifies desired output format (JSON).
  4. **Together AI API Call:**
     + Uses the together Python client.
     + The API key is hardcoded as this isnt a production level application
     + Utilizes the meta-llama/Llama-3.3-70B-Instruct-Turbo-Free model, chosen because it's a powerful model *free to use* within Together AI's offering (subject to their terms and usage limits).
     + Sets response\_format={"type": "json\_object"} to explicitly request JSON output from the AI, improving parsing reliability.
  5. Parses the AI's JSON response, extracting reorder\_quantity (integer) and reasoning (string).
  6. Includes robust error handling for API call failures, empty responses, or invalid JSON output from the AI, providing informative messages.

**3.4. Routes (Web Pages & API Endpoints)**

* **/ & /dashboard (GET)**: Renders the dashboard page, displaying summary statistics like total products, total stock value, number of low-stock products, and total transactions.
* **/products (GET)**: Renders the products page, listing all products and providing a form to add new ones. It also loads all suppliers for the product association dropdown.
* **/add\_product (POST)**: Handles the submission of the "Add New Product" form. Inserts a new product into the Product table with validation for input types and uniqueness (SKU).
* **/edit\_product/<int:product\_id> (GET, POST)**:
  + **GET:** Renders the edit product form, pre-filling it with the data of the specified product ID.
  + **POST:** Processes updates to a product's details.
* **/delete\_product/<int:product\_id> (POST)**: Deletes a product. Includes a check to prevent deletion if associated transactions exist, guiding the user to delete transactions first.
* **/suppliers (GET)**: Renders the suppliers page, listing all suppliers and a form to add new ones.
* **/add\_supplier (POST)**: Handles adding a new supplier, with validation for unique names.
* **/delete\_supplier/<int:supplier\_id> (POST)**: Deletes a supplier. Only allowed if no products are currently associated with that supplier.
* **/transactions (GET)**: Renders the transactions page, showing a history of all transactions and a form to record new ones.
* **/add\_transaction (POST)**: Records a new transaction (purchase or sale). Crucially, it automatically updates the stock\_quantity of the associated product based on the transaction type. It also includes validation to prevent sales if stock is insufficient.
* **/edit\_transaction/<int:transaction\_id> (GET, POST)**:
  + **GET:** Renders the edit transaction form, pre-populating with existing data.
  + **POST:** Handles updates to a transaction. This is complex as it first *reverts* the original transaction's stock impact, then applies the *new* transaction's stock impact (even if the product changes), ensuring accurate stock levels.
* **/delete\_transaction/<int:transaction\_id> (POST)**: Deletes a transaction. It automatically *reverts* the stock quantity of the associated product to its state before the transaction occurred. Includes checks to prevent negative stock.
* **/reports (GET)**: Renders the reports page, gathering data for various analytical charts.
* **/api/suggest\_reorder/<int:product\_id> (GET)**: An API endpoint that calls the suggest\_reorder\_quantity\_from\_ai helper function and returns the AI's reorder suggestion as a JSON response.

**4. Frontend (HTML Templates & JavaScript)**

The frontend is built using standard HTML5, styled with Bootstrap 5 for responsiveness and modern aesthetics, and uses Chart.js for data visualization.

**4.1. base.html**

* The foundational template, defining the basic HTML structure, Bootstrap CSS/JS imports, and a responsive navigation bar.
* Includes a flash message area to display success or error notifications from the backend.
* Defines {% block title %} and {% block content %} for child templates to extend.
* Includes Chart.js and bootstrap.bundle.min.js for global availability.

**4.2. dashboard.html**

* **Features:** Provides a high-level overview of the inventory.
* **Visible Features:**
  + **Summary Cards:** Displays Total Products, Total Stock Value, Low Stock Products count, and Total Transactions in prominent, colored cards.
  + **Low Stock Products Table:** A table listing products currently running low on stock (stock\_quantity < 10), highlighting critical items.

**4.3. products.html**

* **Features:** Central hub for product management.
* **Visible Features:**
  + **"Add New Product" Form:** Allows users to input product name, SKU, unit price, stock quantity, and select an existing supplier from a dropdown.
  + **"All Products" Table:** Displays a sortable table of all products with their ID, Name, SKU, Price, Stock, Supplier, and Creation Date.
  + **Actions Column:** For each product, there are "Edit" and "Delete" buttons.
    - "Edit" button (<a> tag) links to the edit\_product page for that specific product.
    - "Delete" button (<form> with POST) triggers the delete\_product route, with a confirmation prompt.
  + **"Suggest Reorder" Button:** A new button for each product that triggers a modal to display AI-powered reorder suggestions.
  + **AI Reorder Suggestion Modal:** A Bootstrap modal (#reorderSuggestionModal) that appears when "Suggest Reorder" is clicked. It shows a loading spinner while fetching data and then displays the Suggested Quantity and Reasoning from the AI. Includes error handling for UI.

**4.4. edit\_product.html**

* **Features:** Dedicated page for modifying an existing product's details.
* **Visible Features:**
  + A pre-filled form with the current details of the product being edited (Name, SKU, Unit Price, Stock Quantity, Supplier).
  + "Update Product" button to save changes.
  + "Cancel" button to return to the products list.

**4.5. suppliers.html**

* **Features:** Manages supplier information.
* **Visible Features:**
  + **"Add New Supplier" Form:** Allows adding supplier Name and Contact Email.
  + **"All Suppliers" Table:** Lists all suppliers with their ID, Name, Email, and the Products Supplied count.
  + **Actions Column:** For each supplier, a "Delete" button is available. It's conditionally enabled/disabled based on product\_count (can only delete if product\_count == 0), with a confirmation prompt.

**4.6. transactions.html**

* **Features:** Records and tracks all inventory movements.
* **Visible Features:**
  + **"Add New Transaction" Form:** Allows users to select a Product, specify Type (Purchase/Sale), Quantity, Unit Price (at transaction time), and Date.
  + **"Transaction History" Table:** Displays a detailed list of all transactions including ID, Product, Type, Quantity, Price, and Date. Rows are color-coded (green for purchase, red for sale) for easy visual distinction.
  + **Actions Column:** For each transaction, there are "Edit" and "Delete" buttons.
    - "Edit" button links to the edit\_transaction page for that specific transaction.
    - "Delete" button (form with POST) triggers the delete\_transaction route, with a confirmation prompt that warns about stock adjustments.

**4.7. edit\_transaction.html**

* **Features:** Dedicated page for modifying an existing transaction.
* **Visible Features:**
  + A pre-filled form with the current details of the transaction being edited (Product, Type, Quantity, Unit Price, Date).
  + The "Product" dropdown helpfully shows the current stock of each product.
  + "Update Transaction" button to save changes.
  + "Cancel" button to return to the transactions history.

**4.8. reports.html**

* **Features:** Provides visual analytics and insights into inventory performance.
* **Visible Features:**
  + **Inventory Summary Cards:** Repeats key metrics like Total Inventory Value, Low Stock Products count, and Suppliers Count.
  + **Products Running Low Table:** A quick table summarizing low-stock products.
  + **Chart Section:** Contains multiple interactive charts powered by Chart.js:
    - **📈 Sales Over Time (Units Sold):** A line graph showing the total units sold each month over the last 6 months.
    - **🥇 Top Selling Products:** A pie chart illustrating the distribution of the top 5 selling products by quantity.
    - **💸 Net Sales Value Over Time:** A line graph tracking the total revenue generated from sales each month over the last 6 months.
    - **💰 Net Profit Over Time:** A bar chart displaying the net profit (sales revenue - purchase cost) for each month. Bars are dynamically colored green for profit and red for loss.
    - **📊 Purchase vs Sale Ratio:** A bar chart comparing the total count of purchase transactions vs. sale transactions.

**5. Technologies Used**

* **Backend Framework:** Python 3.x with Flask
* **Database:** SQLite3
* **Frontend Libraries:**
  + HTML5
  + Bootstrap 5 (CSS Framework for responsive design and components)
  + Chart.js (JavaScript library for data visualization)
* **AI Integration:** Together AI (using the meta-llama/Llama-3.3-70B-Instruct-Turbo-Free model) for intelligent reorder quantity suggestions. This API is chosen for its powerful capabilities and available free-tier usage for development and testing purposes.

**6. Deployment**

This application can be easily deployed to cloud platforms. A recommended free option for deploying Flask applications is **Render.com**.

**6.1. Deployment on Render.com (Free Tier)**

Render.com offers a generous free tier for web services that is suitable for small Flask applications like ERP Lite.

**Key considerations for Render deployment:**

* **Buildpack:** You would typically use Render's Python buildpack.
* **Start Command:** Your start command would likely be gunicorn app:app (you'd need to pip install gunicorn and potentially create a Procfile).
* **Database:** For Render's free tier, the SQLite database (database.db) will reside on the ephemeral filesystem. This means that if your web service restarts (due to inactivity, updates, or scaling), any data stored in database.db will be **lost**.