**Project Outline: Remaining G1 PO App MVP Development Tasks**

**Project Status as of 5/6/2025:** Sprint 1 (Core Ingestion) Complete. Sprint 2 (UI & Core Logic) in Progress.

**Goal of Remaining Tasks:** Complete the MVP by implementing the core order processing automation ("Process Shipment" functionality) and connecting it to the frontend UI.

**Phase 3: Development Sprints (Continued)**

**Sprint 2: UI & Core Logic (Continued)**

* **Objective:** Build the user interface components for order processing and implement the backend logic that orchestrates document generation, API calls, and database updates.
  + **Task 1: Frontend UI Refinement & Order Detail Connection**
    - **Goal:** Finalize the dashboard display and make it possible to navigate from the dashboard to the Order Detail page.
    - 1.1: **Refactor Dashboard Component:**
      * Move order listing state (useState), data fetching (useEffect), and the table rendering JSX from src/App.jsx into a new component src/components/Dashboard.jsx.
      * Add necessary imports (useState, useEffect, Link, potentially styling) to Dashboard.jsx.
      * Update the fetch call in Dashboard.jsx (fetch('/api/orders?status=new')) to filter for "New" orders (or desired dashboard status).
      * In src/App.jsx, import the Dashboard component.
      * Update the root route (path="/") in App.jsx to render the Dashboard component (element={<Dashboard />}).
    - 1.2: **Make Order Items in Dashboard Clickable:**
      * In the table rendering JSX within Dashboard.jsx, wrap the desired cell content (e.g., BC Order ID) in a react-router-dom <Link> component.
      * Set the to prop of the <Link> to navigate to the Order Detail route using the order's internal database ID: to={/orders/${order.id}}.
  + **Task 2: Implement Core Backend Automation Orchestration (POST /api/orders/<id>/process)**
    - **Goal:** Create the central Flask endpoint that takes user input for sourcing and triggers the entire automated processing workflow.
    - 2.1: **Create Endpoint Route:**
      * In app.py, add a new Flask route: @app.route('/api/orders/<int:order\_id>/process', methods=['POST']).
      * Ensure necessary imports from document\_generator.py and shipping\_service.py are added to app.py.
    - 2.2: **Get Input and Data:**
      * Inside the process\_order(order\_id) function, get JSON data from request.json. This data will contain the user's sourcing choices (Supplier ID, list of PO Line Items with SKU/Qty/Price/Condition/OriginalOrderItemID, Total Shipment Weight, Payment Instructions).
      * Fetch the Order details from the database using the order\_id from the URL. If not found (404) or if is\_international is True, handle these cases (return specific responses, do not proceed with automation for international orders).
      * Fetch the chosen Supplier details from the database using the supplier\_id from the input data.
    - 2.3: **Start Database Transaction:** Open a database connection and start a transaction (conn.execute(sqlalchemy.text("BEGIN")) or conn = engine.connect(); trans = conn.begin()). This transaction should encompass inserting the PO, inserting the Shipment, and updating the Order status in the DB.
    - 2.4: **Generate PO Number:** Implement logic to generate a unique Purchase Order number (e.g., query the purchase\_orders table to find the last ID and increment, or use a time-based/random unique ID).
    - 2.5: **Insert Purchase Order Record:** Insert a record into the purchase\_orders table using data from the fetched Order, fetched Supplier, generated PO Number, PO Date (current datetime), Payment Terms, Payment Instructions. Get the inserted PO ID.
    - 2.6: **Insert PO Line Items Records:** Loop through the list of PO line items from the input data. For each item, insert a record into the po\_line\_items table, linking it to the new PO ID and the original order\_line\_item\_id (from the input data).
    - 2.7: **Generate PO PDF:** Call document\_generator.generate\_purchase\_order\_pdf() with the necessary data (fetched Order, fetched Supplier, generated PO Number, PO Date, the list of PO line items inserted in 2.6). Store the resulting PDF bytes. Handle potential errors.
    - 2.8: **Generate Packing Slip PDF:** Call document\_generator.generate\_packing\_slip\_pdf() with the necessary data (fetched Order - ensure it includes the PO Number, the list of original Order Line Items from the database fetch in 2.2). Store the resulting PDF bytes. Handle potential errors.
    - 2.9: **Generate UPS Label:** Call shipping\_service.generate\_ups\_label() with the necessary data (fetched Order, Ship From Address - **Determine where this comes from (your business details or supplier?), pass it correctly formatted**, Total Shipment Weight from input, Customer Shipping Method Name from original order data). Handle potential failure (e.g., return error response, do not proceed with subsequent steps). Store resulting Label PDF bytes and Tracking Number.
    - 2.10: **Save Generated PDFs to Cloud Storage:** If label generation is successful, use the google-cloud-storage library (pip install google-cloud-storage) to upload the PO, Packing Slip, and UPS Label PDF bytes to a configured GCS bucket. Store the GCS object names or public URLs in the database (PO path in purchase\_orders, Label and Packing Slip paths in shipments). Handle potential errors.
    - 2.11: **Insert Shipment Record:** If label generation is successful, insert a record into the shipments table using data from the order, generated PO ID, Tracking Number, Shipping Method Name, Total Weight, and GCS paths for the label/packing slip.
    - 2.12: **Call Email Sending:** If label generation is successful, call shipping\_service.send\_po\_email() with the recipient supplier email (from fetched Supplier data), generated PO Number, and the PDF bytes. Handle potential failure (log error, but might not need to roll back entire transaction if the email is the only failure).
    - 2.13: **Update Internal Order Status:** Update the status field of the order record in the orders table in your database to 'processed' (or 'shipped').
    - 2.14: **Update BigCommerce:** If the overall process (up to saving to internal DB) is successful, call shipping\_service.update\_bigcommerce\_order() with the BigCommerce Order ID, Tracking Number, Shipping Method Name, and the list of items included in the shipment (mapping original order product IDs to BigCommerce's item IDs on that order if necessary, using quantities from input). Handle potential failure (log error, but the internal DB/documents are already done).
    - 2.15: **Commit Transaction:** trans.commit(). This saves all database changes (PO, PO Items, Shipment, Order Status update).
    - 2.16: **Return Success:** Return a JSON response to the frontend indicating success (e.g., { "message": "Order processed successfully", "tracking\_number": "..." }), typically with a 200 OK status.
    - 2.17: **Implement Error Handling:** Wrap the core logic in a try...except Exception as e: block. In the except block, roll back the database transaction (trans.rollback()), log the error (print(f"Error processing order {order\_id}: {e}")), and return an informative JSON error response to the frontend (e.g., { "message": "Failed to process order", "details": "..." }), typically with a 500 Internal Server Error status. Ensure the database connection is closed in a finally block.
  + **Task 3: Frontend UI for Order Processing**
    - **Goal:** Add the interactive elements to the Order Detail page for users to input sourcing data and trigger the "Process Shipment" workflow.
    - 3.1: **Add State for Sourcing Input:** In OrderDetail.jsx, add useState hooks to manage the data entered in the processing section (e.g., selectedSupplierId, poLineItems, shipmentWeight, paymentInstructions). poLineItems will be an array of objects representing the form data for each item on the PO. Initialize these states.
    - 3.2: **Fetch Data for Sourcing Fields:**
      * In the useEffect hook that fetches the order data, also fetch the list of Suppliers (GET /api/suppliers) and Product SKU Mappings (GET /api/products) when the component mounts. Store this data in state variables.
      * Use the fetched Supplier data to populate the Supplier selection dropdown.
    - 3.3: **Build Sourcing Input Form:** In the "Order Processing" section of OrderDetail.jsx's JSX:
      * Create a dropdown (<select>) for Supplier selection, mapping the fetched suppliers to <option> elements. Update selectedSupplierId state on change.
      * Create a dynamic section to add/edit PO Line Items. Use the poLineItems state. Include buttons to "Add Item". Each item row needs input fields for SKU, Quantity, Unit Cost, Condition. When SKU changes, look up the standard description from the fetched product mappings data and populate a non-editable or editable description field.
      * Add input fields for Total Shipment Weight (<input type="number">). Update state on change.
      * Add a text area (<textarea>) for Payment Instructions. Pre-populate this based on the selected supplier's payment\_terms fetched data. Update state on change.
    - 3.4: **Add "Process Shipment" Button:** Add a button element. Attach an onClick handler to trigger the submission logic. Manage the button's disabled state based on form validation and submitting status.
    - 3.5: **Implement Submission Logic:** Create an async function for the button's onClick. This function validates the user input client-side (e.g., required fields filled, valid numbers). If valid, collect all the data from the input state variables into a single object matching the format expected by the backend POST /api/orders/<id>/process endpoint.
    - 3.6: **Call Backend "Process Shipment" Endpoint:** Use fetch or axios to send a POST request to /api/orders/${orderId}/process, sending the collected input data in the json body. Handle loading, submitting, error, and success states.
    - 3.7: **Handle Response:** If the response is successful (200 OK), display a success message to the user (e.g., using state), potentially showing the tracking number. Update the UI (e.g., disable the form or navigate back to the dashboard). If the response is an error (4xx/5xx), update the error state and display the error message to the user in the UI.
  + **Task 4: Implement Remaining Backend Functionality**
    - **Goal:** Add the PO Data Export endpoint.
    - 4.1: **Implement PO Data Export API:**
      * In app.py, add the GET /api/exports/pos route.
      * Implement the logic to query the necessary data from your database (orders, purchase\_orders, po\_line\_items, shipments).
      * Use pandas and openpyxl to generate an Excel file in memory (BytesIO).
      * Return the BytesIO content as a Flask response with appropriate headers (Content-Disposition, Content-Type).
    - 4.2: **Frontend Link:** Add a link in the UI (e.g., on the Dashboard or a separate admin page) that points to the GET /api/exports/pos URL to trigger the download.

**Phase 4: Deployment to GCP (Major Phase)**

* **Objective:** Package the application and deploy it to the Google Cloud Platform.
  + **Task 5: Backend Containerization:**
    - Create a Dockerfile in the backend directory to build a Docker image for the Flask application. Include Python, dependencies, and your app code.
  + **Task 6: Frontend Build & Containerization:**
    - Run npm run build in the frontend directory to generate optimized static files.
    - Create a Dockerfile for a lightweight web server (like Nginx) to serve the static frontend files from the build output (dist folder).
  + **Task 7: GCP Environment Setup:**
    - Ensure Cloud SQL instance is running.
    - Set up a Cloud Storage bucket for frontend static files and configure static website hosting.
    - Set up a Cloud Storage bucket for storing generated PDF documents.
    - Enable Google Artifact Registry (or Container Registry) to store Docker images.
  + **Task 8: Deploy Backend to Cloud Run:**
    - Build the backend Docker image (docker build).
    - Tag and push the image to Artifact Registry.
    - Create a Cloud Run service for the backend. Configure region, CPU/RAM, scaling, environment variables (using Secret Manager for sensitive credentials like DB password, API keys).
    - Configure Cloud Run service identity/service account permissions to access Cloud SQL and GCS.
    - Set up Private IP connection from Cloud Run to Cloud SQL.
  + **Task 9: Deploy Frontend to Cloud Storage (and Cloud Run if preferred):**
    - Copy the built static frontend files (dist folder) to the GCS static hosting bucket.
    - *Alternative:* Build and deploy the frontend Docker container to a separate Cloud Run service if that architecture is preferred.
  + **Task 10: Networking & DNS (Optional):**
    - Configure custom domains if needed (Cloud Run domain mapping, Cloud Storage custom domain).
  + **Task 11: Testing in Cloud:** Test the deployed application end-to-end in the GCP environment.

**Phase 5: User Authentication (Major Phase)**

* **Objective:** Secure the application with a login system.
  + **Task 12: Choose Authentication Method:** Decide on the approach (e.g., Flask-Login, Firebase Auth, Auth0).
  + **Task 13: Implement Backend Authentication:** Add user model/storage, login/logout routes, and protect existing API endpoints.
  + **Task 14: Implement Frontend Authentication:** Add login/logout UI, manage user state in React, protect frontend routes.

This detailed outline covers the remaining work, broken down into logical tasks and sub-steps. The human developer can use this as a guide for their sprint planning and implementation.

I wish you and the developer success in bringing the full vision of this application to life!

thumb\_upthumb\_down