

# POS Nano: Assessment Brief

## 1. Context and goal

We're building POS Nano, a minimal point-of-sale system for small South African retailers.

Your task is to implement a small but realistic part of this product:

- A cashier flow to ring up sales
- A sales reporting view with a daily sales chart
- A tiny "Ask POS" AI stub that turns simple text queries into report filters

You can choose your own stack for the application layer(s). The only hard requirement is:

- Postgres must be used as the database.

We're interested in your system design, code quality, and reasoning.

## 2. Core business requirements

### 2.1 Data model

Please persist data in Postgres using these concepts (you can refine/improve if you would like to):

- Product (Fields: SKU (unique), name, unit price, timestamps, colour, sizing)
- Stock (Per-product quantity on hand)
- Sale (A completed checkout, with: total amount, VAT amount, timestamp)
- Sale item (Line items tying a sale to products, with quantity and unit price)

### 2.2 Cashier flow

Build a minimal UI and backend that support:

- Search products
  - Search by SKU or name (case-insensitive).
  - Show a small list of matching products.
- Cart management
  - Add products to a "basket"
  - Change quantities, remove items
  - Show live totals: subtotal, VAT, total
- Checkout
  - When the cashier confirms:
    - Validate that there is enough stock for each line
    - Create a Sale and corresponding Sale items
    - Decrease Stock quantities

- If stock is insufficient, reject the checkout with a clear error.

Additional Points for:

- Idempotency (double-submit protection): If the cashier accidentally clicks “Pay” twice, we don’t want to charge twice. An idea:
  - Implement idempotency for sale creation using a client-supplied key (e.g. a header or field).
  - Repeating the same request with the same key should not create a second sale.

## 2.3 Daily sales report

Provide a simple daily sales report that aggregates:

- Input: a date range (e.g. 2025-01-01 to 2025-01-07)
- Output per day that has sales:
  - Date
  - Total sales amount for that day
  - Number of sales (count)

On the frontend, render this as a chart (line or bar) showing daily totals across the selected range.

## 2.4 “Ask POS” AI stub

Implement a small, deterministic stub that:

- Accepts a short natural-language query string, for example:
  - "daily sales from 2025-01-01 to 2025-01-07"
  - "daily sales last 7 days"
- Parses it into a structured intent, e.g.:
  - A concrete start and end date, or
  - “Last N days” relative to today
- Uses that intent to call your own daily sales report logic and update the chart.

No real AI is required:

- No external LLM calls, no API keys, no network dependencies.
- This can be a simple function or class that uses regex/string parsing.
- If you prefer to just use an LLM (model from OpenAI / Claude / Gemini, etc.) instead, this is completely acceptable. The reasoning behind this section stub/mock of the AI feature) is so that you do not need to incur any costs calling an LLM.

# 3. Technical expectations

## 3.1 Stack

- Database: Postgres (required)
- Application stack: your choice (backend, frontend, or full-stack)
  - Examples you might choose:
    - Backend: Node, Python (FastAPI/Django), Rust, Java, etc.
    - Frontend: React, Angular, etc.
- You may structure your solution as:
  - A separate API + SPA frontend
  - A monolithic web app that renders HTML and uses some JS for interactivity
  - Or any reasonable architecture that satisfies the requirements
- Note: since you are free to choose your tech stack and architecture, we do not provide a boilerplate code - you are free to design this nano system without specific tech stack limitations.

## 3.2 Deliverables

In a single GitHub repository, please include:

- Application code
  - Database schema
  - Backend logic for products, sales, reports, and AI stub integration
  - Frontend or UI layer implementing the cashier screen, report chart, and Ask POS input
- README
  - How to run your application
  - Any design decisions, trade-offs, and known limitations
  - Optional: brief notes on what you would do next with more time

## 4. Timebox and how we review

- Suggested timebox: around 3-6 hours of focused work (you'll have a few days to complete it).
- If it takes too long to do, stop where you feel a complete part is ready and tell us what you would do next. (In the README, note what you'd complete or improve with more time.)

When reviewing, we focus on:

- Explainability of your code (can you explain your code? Even if it's generated using AI, you should still be able to explain your code, or make changes to it if necessary)
- Correctness of business logic (VAT, totals, stock handling, etc)
- Data modeling in Postgres
- Clarity and structure of your code
- UX that is simple and usable