Take-Home Assignment Instructions

Inventory Reservation API

1. Purpose of this assignment

You will design and implement a small backend API for a fictitious store.

This task is designed to evaluate engineering judgement, correctness, and practical system design under time constraints.

You do not need to build any user interface.

3. Allowed tooling

You are allowed to use AI-assisted development tools (e.g., Cursor, Claude, ChatGPT).
We will evaluate how you apply engineering judgement (e.g., correctness, clarity, and robustness), not the volume of code produced.

4. Required tech stack

You must use all of the following:

- Express.js with TypeScript
- Supabase (PostgreSQL) as the database
- Vercel for deployment (API only)

5. Deliverables (must provide all)

Submit the following:

A. GitHub repository link containing:

- Source code
- README.md
- .env.example
- SQL migration file(s) (see point 5.C)

B. Swagger / OpenAPI documentation:

- Swagger UI served at `/docs`
- OpenAPI JSON served at `/openapi.json` (or equivalent, clearly stated)

C. SQL migration file:

- Provide a SQL file that creates the full database schema needed for your solution:
    - tables
    - constraints
    - indexes
    - foreign keys
- It must be runnable in Supabase SQL Editor without additional manual steps.

D. Deployment:

- Deploy the API to Vercel
- Provide the deployed base URL in the README

E. Demo video (5–10 minutes):

- A screen recording that demonstrates the behaviour described in Section 10
- Share via a link accessible to reviewers

6. Definitions

Item
A product that can be purchased (example: “White T-Shirt”).

Available quantity
The number of units that are free to be reserved or confirmed.

Reservation
A temporary hold of some quantity of an item for a specific customer.

Confirmation
Finalising a reservation. Confirmed units must permanently reduce availability.

Expiration
A reservation becomes invalid after a time window (example: 10 minutes). Expired reservations must not block inventory indefinitely.

7. Core behaviour you must support

Your service must allow:

- creating items with an initial quantity
- reserving (holding) quantity for a customer
- confirming a reservation
- cancelling a reservation
- expiring old reservations

8. Required API endpoints

You may add additional endpoints, but these are required.

8.1 Create item

POST `/v1/items`

Request body (example shape)

- name: string
- initial_quantity: integer > 0

Response

- item id
- name
- total quantity
- any other fields you find helpful

8.2 Get item status

GET `/v1/items/:id`

Response must clearly include:

- total quantity
- quantity currently available
- quantity currently held in active reservations
- quantity already confirmed (optional but recommended)

The exact method you use to compute these values is up to you, but the numbers must be correct and explainable.

8.3 Create reservation (temporary hold)

POST `/v1/reservations`

Request body

- item_id: string/uuid
- customer_id: string
- quantity: integer > 0

Rules

- If there is insufficient available quantity, respond with an appropriate error (e.g., 409).
- Reservation must have an expiry time (example: 10 minutes from creation).

Response

- reservation id

- item_id
- customer_id
- quantity
- status (e.g., PENDING)
- expires_at timestamp

8.4 Confirm reservation

POST `/v1/reservations/:id/confirm`

Rules

- Confirming a valid reservation permanently deducts the reserved quantity.
- Confirming twice must not deduct twice (retry-safe behaviour is required).
- Confirm after expiration must not deduct inventory.

Response

- reservation id
- final status
- relevant timestamps or state information

8.5 Cancel reservation

POST `/v1/reservations/:id/cancel`

Rules

- Cancelling a valid pending reservation releases quantity back to availability.
- Cancelling twice must not release twice (retry-safe behaviour is required).
- Cancelling after confirmation should not increase availability.

Response

- reservation id
- final status
- relevant timestamps or state information

8.6 Expire reservations

POST `/v1/maintenance/expire-reservations`

Behaviour

- marks old pending reservations as expired
- releases their quantity back to availability

9. Demo video requirements (5–10 minutes)

Your video must show, at minimum:

1. Starting the service locally
2. Opening Swagger UI (`/docs`) and showing the endpoints
3. Creating an item with a small quantity (example: 5)
4. Demonstrating expiration or cancellation and showing quantity becomes available again
5. Showing the database state in Supabase (tables/rows) to prove consistency

10. Database requirements

Your database schema must include, at minimum:

- items table (or equivalent)
- reservations table (or equivalent)

Reservations must capture:

- item reference
- customer reference
- quantity
- status
- created time
- expiration time

Your schema should include:

- constraints to protect data integrity (example: quantity > 0)
- foreign keys
- indexes appropriate for the access patterns (example: by item_id, status, expires_at)

11. Documentation requirements (README.md)

Your README must include:

- a short overview of the system and assumptions you made
- how to set up Supabase and run the SQL migration
- how to run the API locally
- required environment variables (and ensure they are in `.env.example`)
- how to deploy to Vercel
- the deployed URL
- how to reproduce the concurrency scenarios (commands or steps)
- link to the demo video
- any known limitations or trade-offs made due to the 4-hour timebox

12. API quality requirements

- Use consistent error response shape (your choice, but be consistent)
- Validate inputs (your choice of library or approach)
- Return appropriate HTTP status codes (e.g., 400/422 validation, 404 not found, 409 insufficient availability, 429 if you implement rate limiting, etc.)

13. What we will evaluate

We will review your submission primarily on:

Correctness under concurrency

- prevents overselling when requests overlap
- handles retries safely without double-deduct or double-release

Database-driven consistency

- sensible use of constraints and atomic operations
- schema supports correctness and is reviewable

Design clarity

- clean separation of concerns (routing, business logic, database access)
- readable naming and maintainable structure

Reproducibility

- migration runs cleanly
- README is accurate and complete
- quick to run locally and easy to deploy

Documentation and demo quality

- Swagger matches the actual API behaviour
- demo video proves the important behaviours clearly

14. What you do not need to build

- no front-end UI
- no payment integration
- no user registration system
- no background queues/workers required
- no complex admin portal

15. Submission format

Email us the following:

- GitHub repo link

- Deployed Vercel URL
- Demo video link

Note: Include these links at the top of your README for quick access.

16. Additional Requirements

16.1 Strong Consistency Under Concurrency (No Oversell, Ever)

Your system **must remain correct under any level of concurrency**. At all times, the following invariant must hold for each item:

confirmed_quantity + active_pending_unexpired_quantity ≤ total_quantity

This must remain true even in the presence of overlapping requests, including (but not limited to):

- Two `POST /v1/reservations` calls racing against each other for the same item
- `POST /v1/reservations/:id/confirm` racing against `POST /v1/maintenance/expire-reservations`
- `POST /v1/reservations/:id/cancel` racing against `POST /v1/reservations/:id/confirm`
- Multiple `POST /v1/maintenance/expire-reservations` requests running concurrently

Enforcement expectations

- Correctness must be achieved using **database-driven consistency**, such as:
  - transactions
  - row-level locks
  - atomic updates
  - constraints/unique guards
- **In-memory locks, single-instance assumptions, or “it works locally” approaches are not acceptable**, because the service must be horizontally scalable.

Documentation requirement

- In your README, explicitly explain **which SQL guarantees** ensure the invariant holds (e.g., transaction boundaries, locking strategy, constraints/indexes used to prevent oversell or double-finalisation).

16.2 Mandatory Concurrency Proof Test (Must Demonstrate the Invariant)

To prove you meet the strong-consistency requirement above, your repository must include a deterministic concurrency test script:

Command

- `npm run test:concurrency`

The script must:

1. Create an item with **total quantity = 50**
2. Fire **200 concurrent** reservation requests, each reserving **quantity = 1** for the same item
3. Assert the outcome is exactly:
    - **50 successes**
    - **150 failures with HTTP 409** (insufficient availability)
4. After all requests complete, query the database and assert the invariant holds:
    - **confirmed + active_pending_unexpired ≤ total_quantity**
    - and that **no overselling occurred** (even transiently)

Additionally, it must repeat the test with race scenarios, demonstrating correctness under overlapping operations, such as:

- confirm vs expire races
- cancel vs confirm races
- multiple concurrent expire calls

Important

- The test must be reproducible and runnable by reviewers against a fresh database (include setup steps in README).
- The test should be reliable (not “flaky”); design your concurrency approach so the test consistently passes.