

# **CHECKPOINT 1 CS-UH 3260: Software Architecture**

Retail Management System Prepared by:

Terezia Juras (tj2286)  
Izah Sohail (is2587)

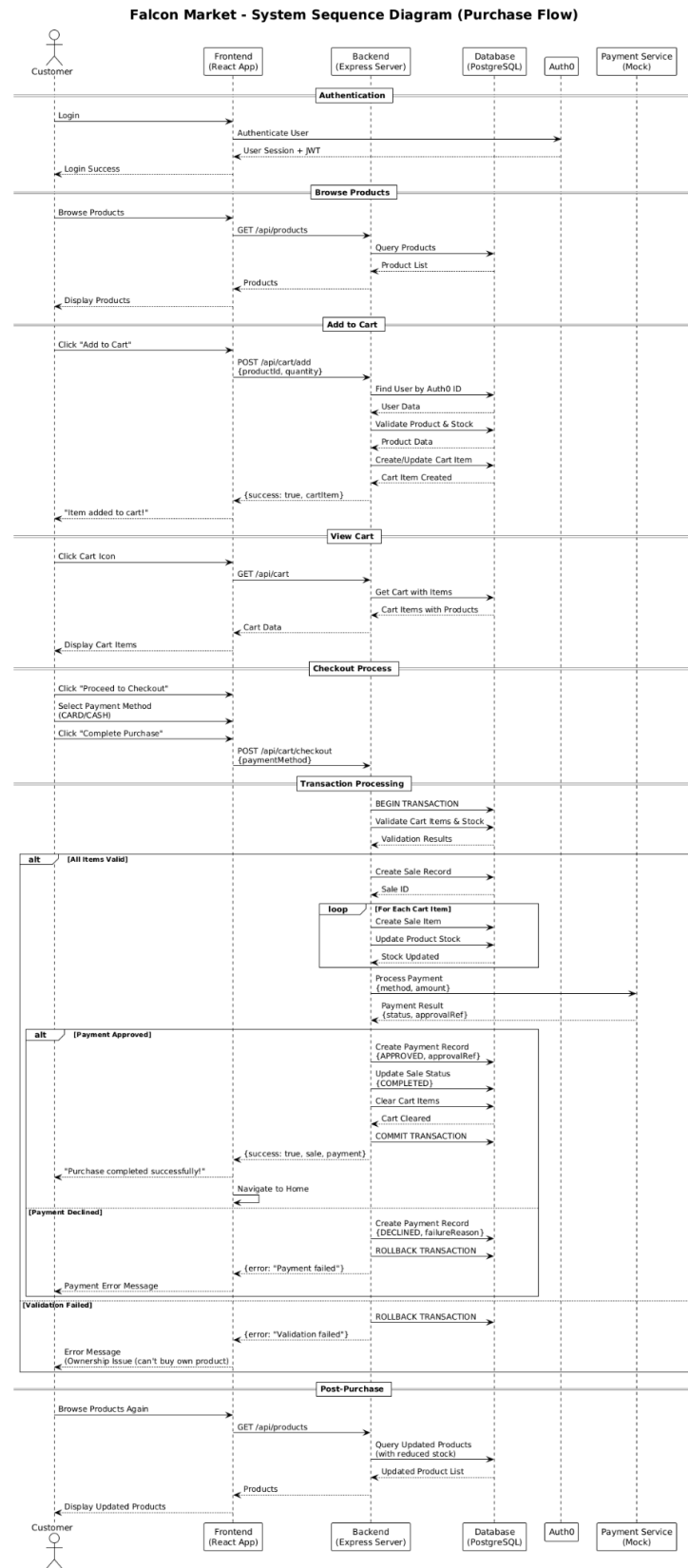


Figure 1: Falcon Market System

Sequence Diagram

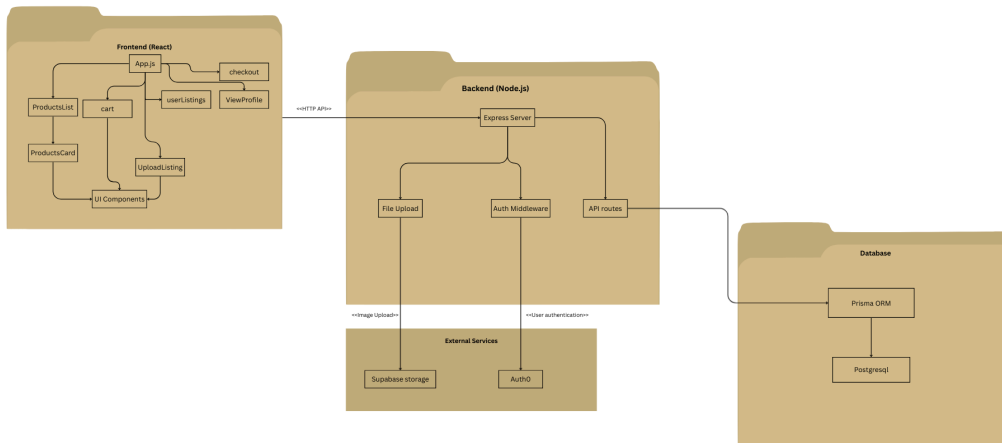
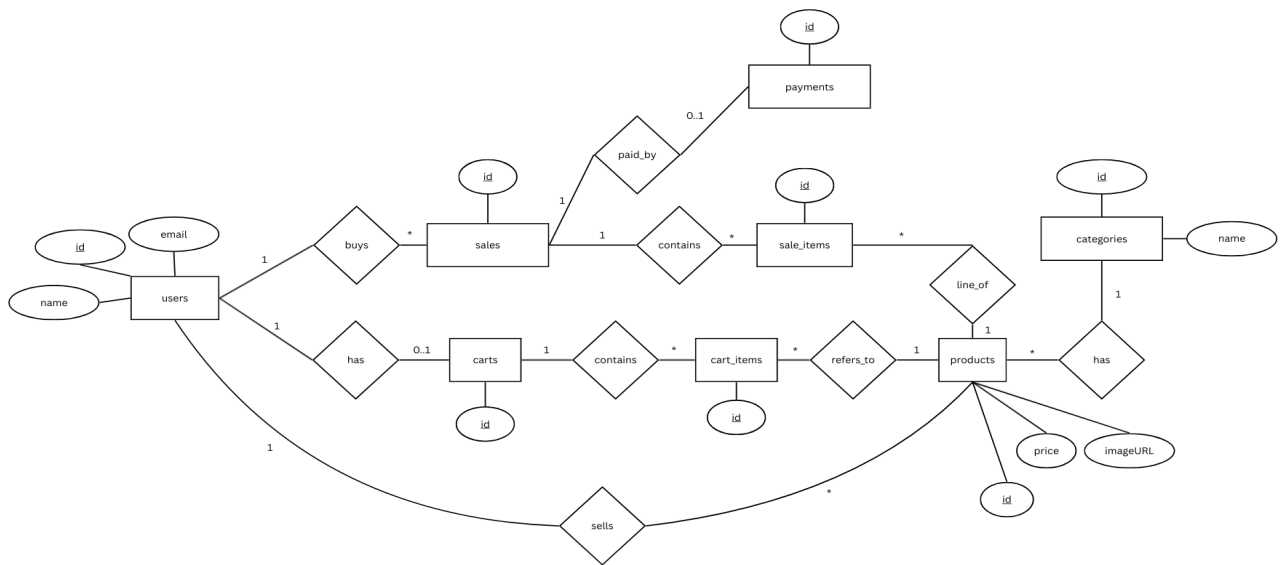


Figure 2: Falcon Market Module Diagram of Frontend, Backend, Database and External Services



3Figure 3: ER Diagram

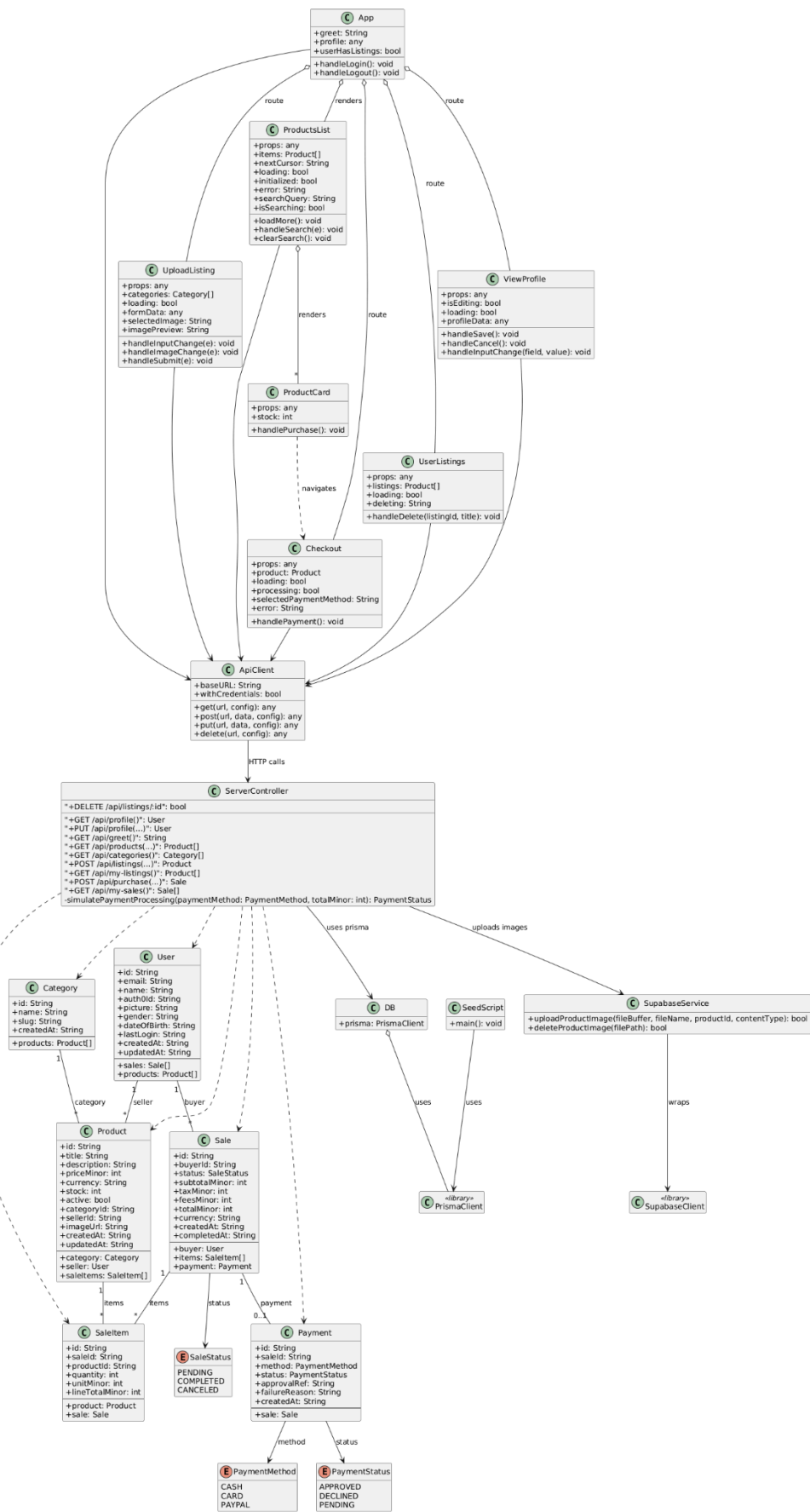


Figure 4: Class Diagram

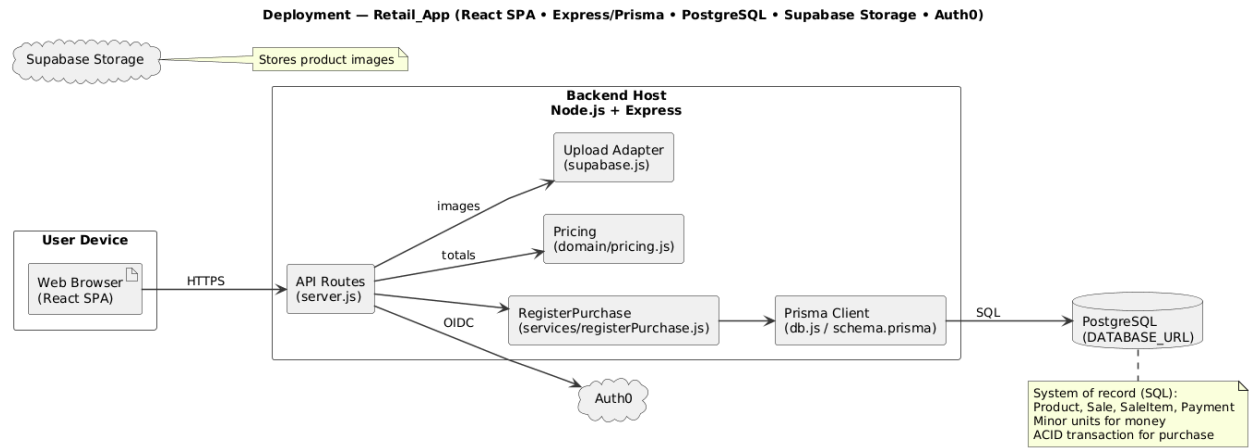


Figure 5: Deployment Diagram

## **First Use Case: Register a Sale / Purchase**

**Primary Actor:** End User

**Supporting Actors:** Payment Service (mock)

**Trigger:** User chooses "Purchase" from the app

### **Preconditions**

- Product catalog and current stock are loaded.
- User is authenticated.
- Payment options are configured (e.g., Cash, Card).

### **Postconditions**

- A Sale record exists with timestamp, line items, totals, payment details, and status = Completed.
- Stock levels are decremented for each purchased item (atomically with the sale).
- Receipt/confirmation is available.

### **Main Success Scenario (Basic Flow)**

1. System displays an empty cart.
2. User adds product(s) by entering Product ID (or searching) and enters quantity for each.
3. System validates product IDs and checks stock for requested quantities.
4. System computes line totals and order total (incl. taxes/fees/discounts if applicable) and displays a running summary.
5. User chooses a payment option (at least two supported).
6. System processes payment and receives an approval/confirmation.
7. System persists the sale (timestamp, items, quantities, unit prices, subtotal, total, payment method, payment ref).
8. System decrements inventory for each product by purchased quantity (atomic with step 7).
9. System shows Success and offers to print/download a receipt.

### **Alternate / Exception Flows**

- **A1. Invalid Product ID**  
3a. System shows "Product not found"; item not added; user can retry or cancel.
- **A2. Insufficient Stock**  
3b. System shows "Only X in stock".  
User may: (i) reduce quantity to available, (ii) remove item, or (iii) cancel sale.
- **A3. Pricing/Totals Change Mid-Flow**  
4a. System recalculates and highlights changes before payment; user confirms before proceeding.

- **A4. Payment Failure/Decline**  
6a. System displays reason and logs attempt (**no sale persisted, no stock change**).  
User may choose a different payment method or cancel.
- **A5. Concurrency Conflict on Stock (another sale just consumed stock)**  
8a. Transaction fails due to insufficient stock at commit.  
System rolls back/voids payment (mock), informs user, and returns to step 3.
- **A6. User Cancels Before Payment**  
Cart discarded; nothing persisted.
- **A7. Not Authenticated**  
Before step 1, system requires login; unauthenticated users are redirected to login (see “Purchase Requires Login”).
- **A8. Self-Purchase Detected (user trying to buy their own product)**  
Before step 6, system checks ownership of each cart item.  
If any item’s seller is the current user, system blocks checkout, shows message, and lets user remove the item(s) or switch account (see “Self-Purchase Prevention”).

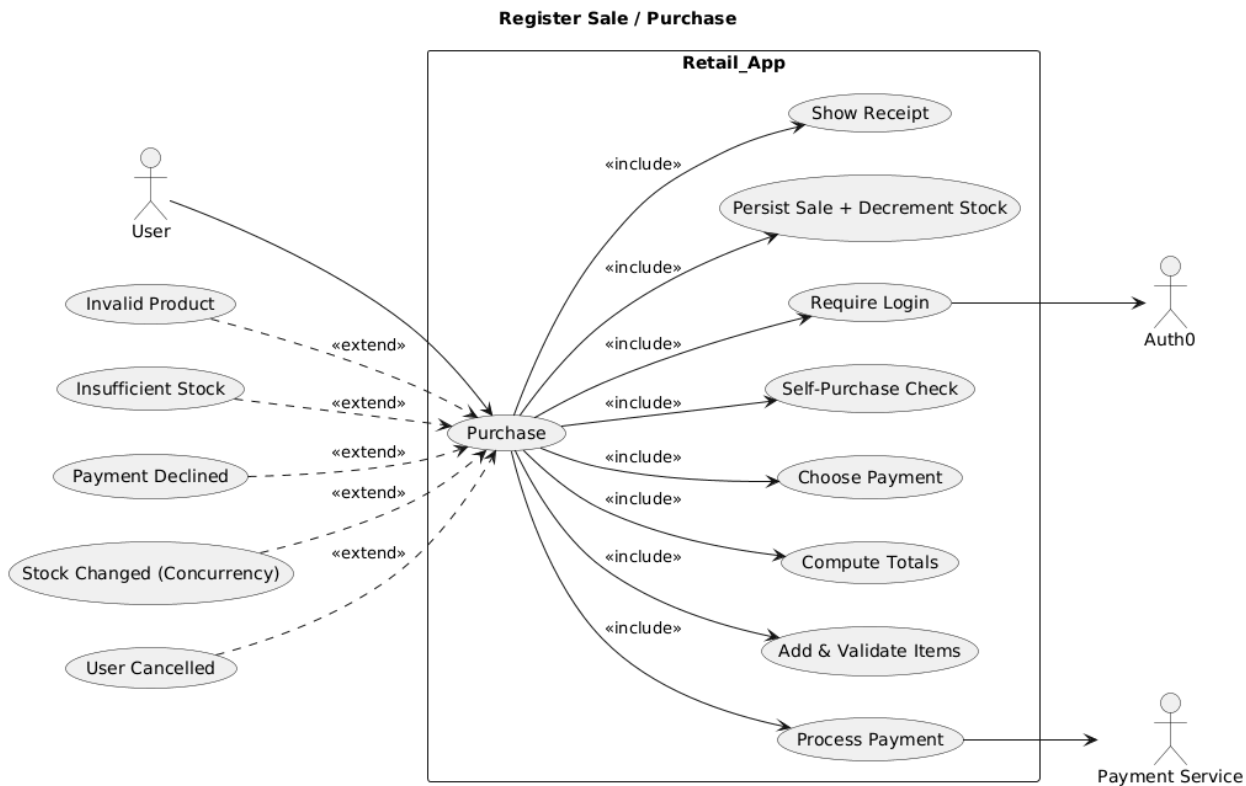


Figure 6: Register a Sale / Purchase Use Case

## Second Use Case: Purchase Requires Login (Not Logged In)

**Primary Actor:** Anonymous Visitor

**Supporting Actors:** Authentication Service (Auth0)

**Trigger:** User chooses “Purchase” from the app while not logged in

### Preconditions

- None (visitor is not authenticated).

### Postconditions

- If authentication succeeds, user has an active session and may proceed to purchase.
- If authentication fails/cancels, no purchase occurs and nothing is persisted.

### Main Success Scenario (Basic Flow)

1. Visitor chooses “Purchase”.
2. System detects missing session and redirects to login (Auth0).
3. Visitor logs in successfully; system establishes session and returns to the app.
4. System resumes checkout; user proceeds with the “Register a Sale / Purchase” flow.

### Alternate / Exception Flows

- **A1. Login Failed / Cancelled**  
3a. System shows login error/cancel notice; purchase is aborted; nothing persisted.

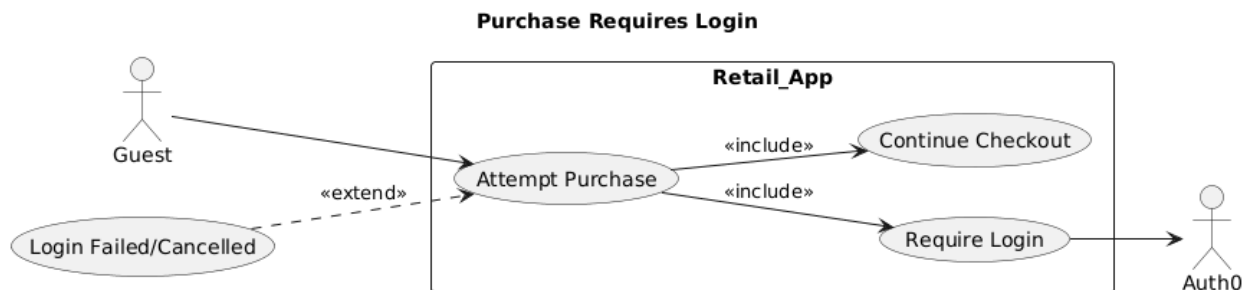


Figure 7: Purchase Requires Login (Not Logged In) Use Case



### Third Use Case: Self-Purchase Prevention (Trying to Sell, Buy Same Product)

**Primary Actor:** Authenticated User

**Supporting Actors:** —

**Trigger:** User proceeds to checkout with a cart that may include items they are selling

#### Preconditions

- User is authenticated.
- Cart contains one or more items.

#### Postconditions

- If conflict exists (self-purchase), checkout is blocked; no payment attempt and no persistence.
- If no conflict, user continues the normal purchase flow.

#### Main Success Scenario (Basic Flow)

1. System checks each cart item's seller against the current user.
2. If no item is owned by the user, system proceeds with the "Register a Sale / Purchase" flow.

#### Alternate / Exception Flows

- **A1. Self-Purchase Detected**
  - 1a. System blocks checkout and shows "Self-purchase not allowed".
  - 1b. User options: (i) remove conflicting item(s), (ii) switch account, or (iii) cancel.
  - 1c. On successful resolution (items removed or different user), return to step 1; otherwise abort.

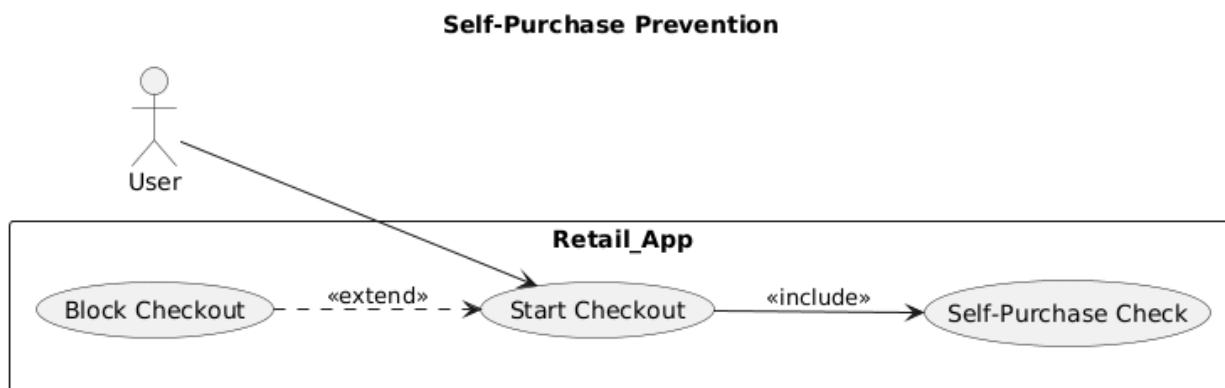


Figure 8: Self-Purchase Prevention (Trying to Sell, Buy Same Product) Use Case

## **ADR-001: Database Choice SQL (PostgreSQL) over NoSQL**

**Status: Accepted**

### Context (problem & background)

The application is a transactional retail system. Each purchase creates a Sale with one or more SaleItems, records a Payment outcome, and decrements Product stock. Amounts are stored in minor units for accuracy and audit. The system must support reliable reads (SKU/name lookup), reporting (revenue by period, top SKUs, margins), and ongoing schema evolution (discounts, promotions, tax rules). Data integrity and auditability are non-negotiable.

### Decision (what we chose)

Agreed to adopt a relational SQL database as the system of record, specifically, PostgreSQL. Enforce referential integrity (FKs), uniqueness (e.g., SKU), and basic checks (non-negative stock/amounts). Execute the entire purchase flow within a single ACID transaction. Manage schema through versioned Prisma migrations.

### Consequences (pros/cons, trade-offs)

#### Pros

- ACID guarantees prevent partial writes and double-selling under concurrency.
- Foreign keys and constraints keep the ledger coherent and auditable.
- SQL enables expressive joins and aggregations for operational analytics.
- Migrations are reproducible and reviewable across environments.

#### Cons

- Requires migration discipline and database operations hygiene.
- Horizontal write scaling is more complex than some NoSQL options.

#### Trade-offs

Prioritizes correctness, integrity, and analytics over early horizontal scale. Read replicas and caching can be introduced later as needed.

#### Alternatives considered

- Document store (e.g., MongoDB): flexible schema, but integrity shifts to application code; eventual consistency risks stale stock and inconsistent ledgers; analytics require extra tooling.
- Graph database: optimized for traversal, not for tabular ledgers and financial constraints.
- Event sourcing + CQRS: strong audit trail and replay, but significantly higher operational complexity; unnecessary for current scope.
- OLAP warehouse as primary store: built for analytics, not OLTP; transactional guarantees and latency are unsuitable.
- Files or in-memory storage: fails durability, concurrency, and transaction requirements

## ADR-002: Persistence Style

### Prisma ORM behind a Repository Boundary (vs DAO)

**Status: Accepted**

#### Context (problem & background)

The system must deliver a full “Register a Sale / Purchase” workflow with atomic writes across Sale, SaleItems, Payment, and stock updates, while remaining type-safe and easy to run locally. Controllers should not leak database concerns; business rules (tax/fees, validation, error mapping) should live in services. The design should allow swapping data access strategies in future checkpoints without rewriting business logic.

#### Decision (what you chose)

Use Prisma ORM for database access and migrations. Keep a clear persistence boundary: services depend on repository interfaces; Prisma lives in adapter implementations. Use `prisma.$transaction` to execute the purchase flow atomically. Keep pricing/totals as pure functions for straightforward unit testing.

#### Consequences (pros/cons, trade-offs)

##### Pros

- End-to-end type safety from schema to code via the generated client.
- Concise data access with straightforward nested writes and transactions.
- First-class, versioned migrations integrated into the workflow.
- Faster delivery and simpler onboarding than handwritten DAOs.
- Clean layering: controllers → services → repositories → adapters.

##### Cons


- Some SQL control is abstracted; hotspots may need targeted raw queries.
- Contributors must learn Prisma’s schema/migrate workflow.

#### Trade-offs

Accept a thin ORM layer to gain speed and safety now. Preserve portability by keeping repositories as the seam for future DAO or ORM changes.

#### Alternatives considered

- DAO with raw SQL (pg/sqlite3): maximum control and transparency; more boilerplate for mapping and transactions; slower iteration; migrations must be hand-authored.
- Query builder (Knex): reduces SQL verbosity but lacks Prisma’s generated types and schema-first workflow.
- Other ORMs (TypeORM, Sequelize, Mikro-ORM): viable choices; Prisma preferred for schema-first design, generated types, nested writes, and tooling quality.
- Stored procedures/DB logic: good for hot paths, increases coupling to a specific dialect, complicates tests/CI.

Demo Link:  Screen Recording 2025-09-19 at 7.17.20 PM.mov