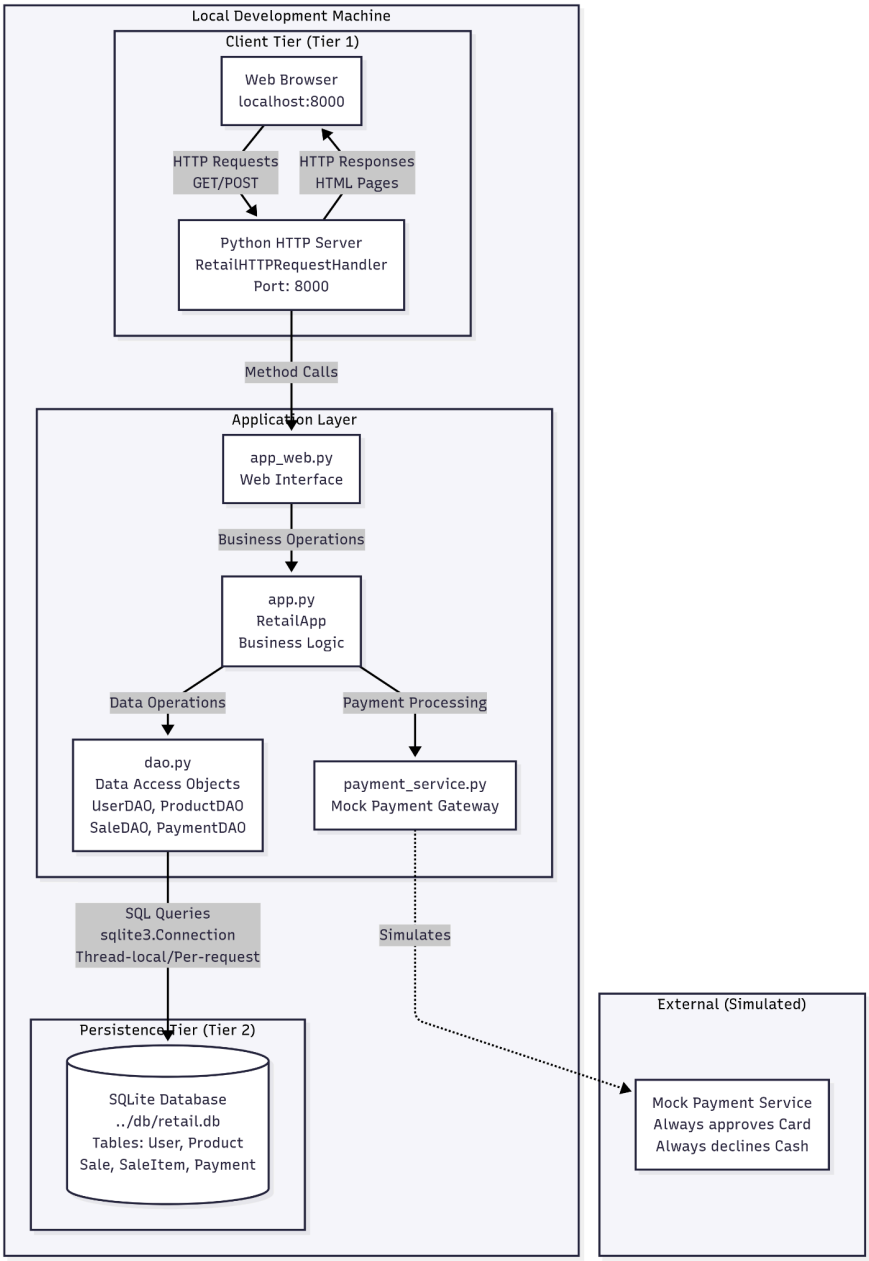
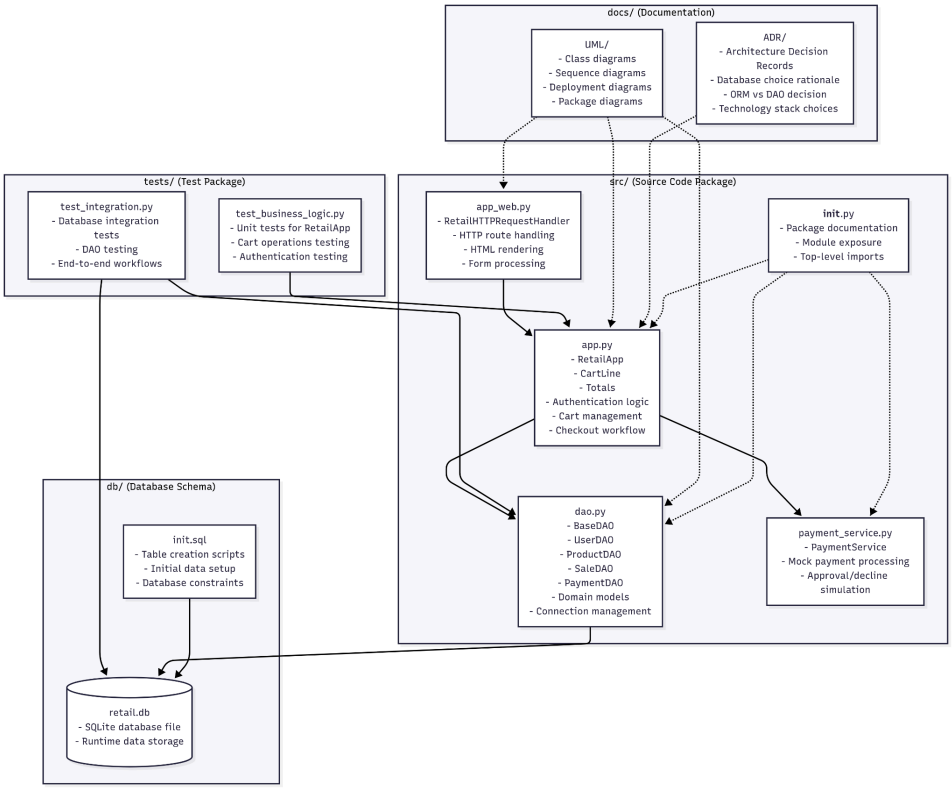
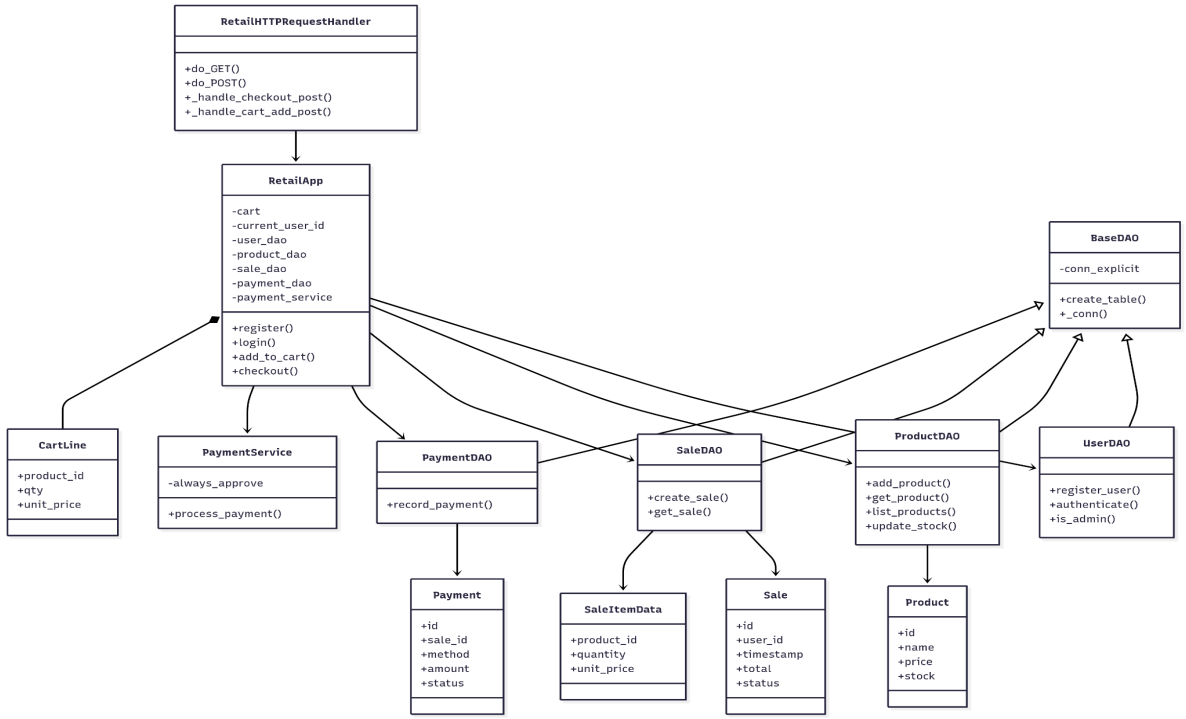
Deployment View- Client + DB Architecture



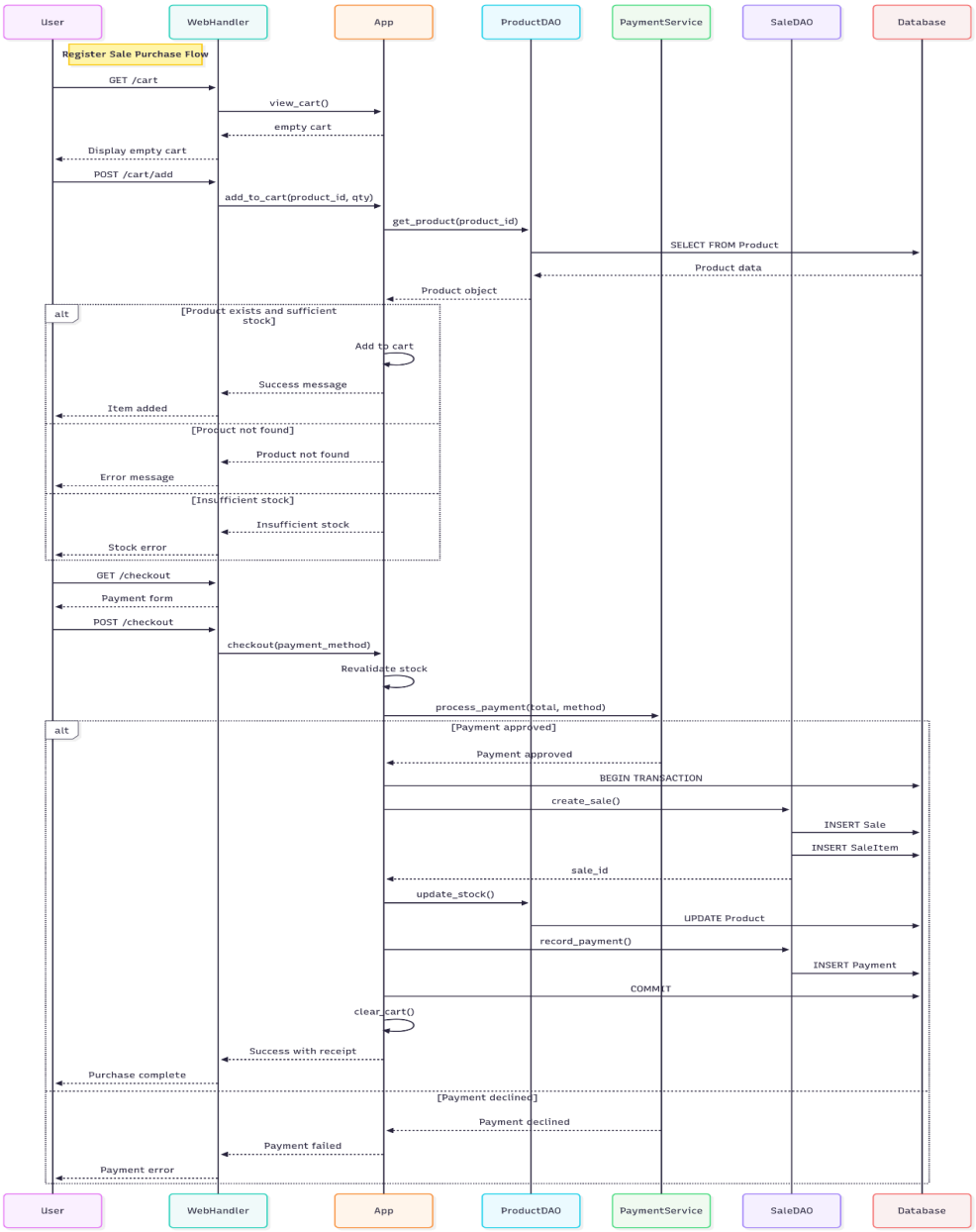
Implementation View- Package/Module Diagram



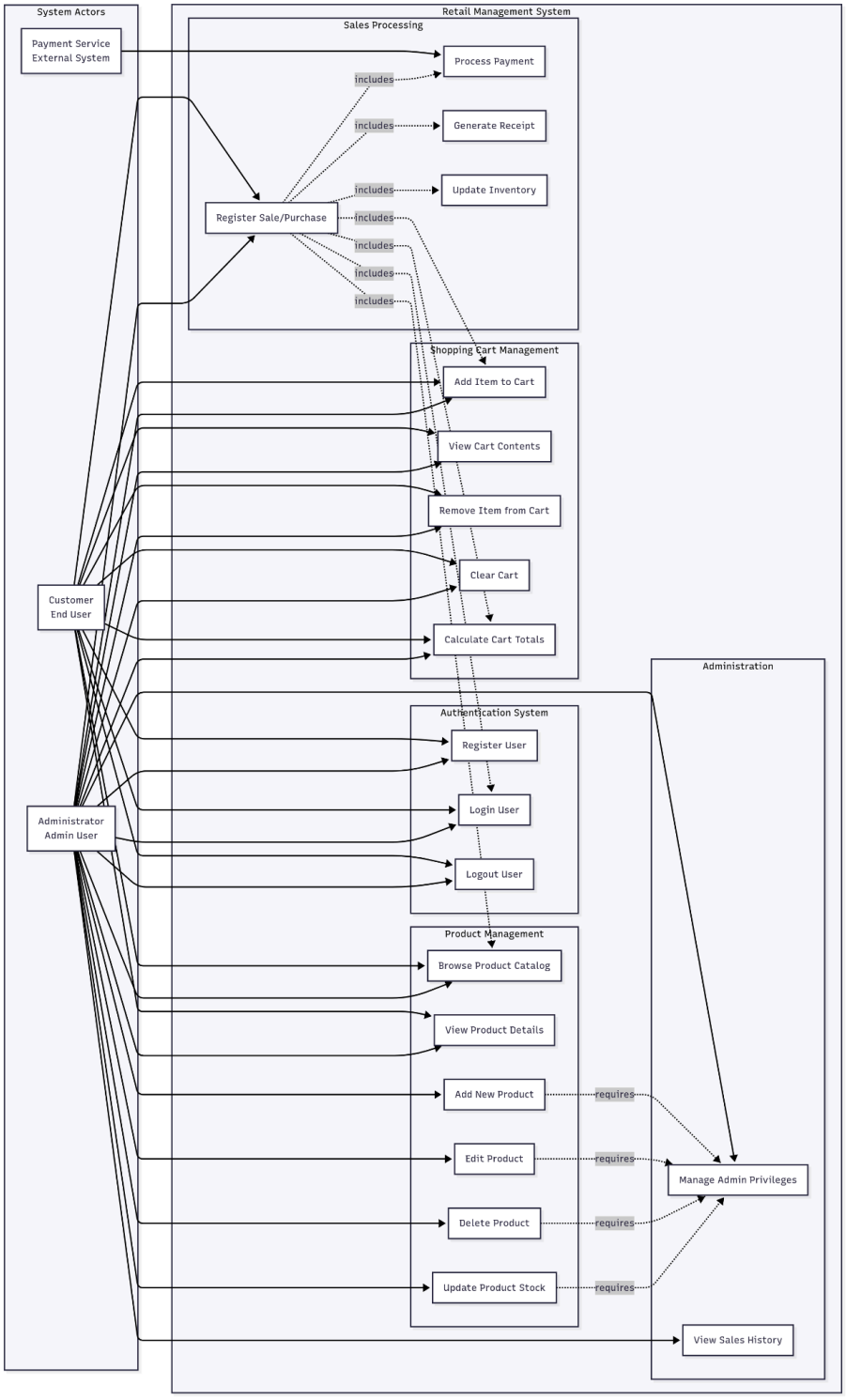
Logical View- Class Diagram



Process View- System Sequence Diagram (Purchase Flow)



Use-Case View - Register Sale and Supporting Use cases



KEY/LEGEND AND NOTATION GUIDE

1. Process View - Sequence Diagram Legend

Participants/Actors:

- User - End user interacting with the web interface
- WebHandler - RetailHTTPRequestHandler (web server)
- App - RetailApp (business logic layer)
- ProductDAO - Product data access object
- PaymentService - Mock payment processing service
- SaleDAO - Sales data access object
- Database - SQLite database

Message Types:

- Solid Arrow (→) - Synchronous method call/request
- Dashed Arrow (-->>) - Return/response message
- Plus (+) - Activation (method execution begins)
- Minus (-) - Deactivation (method execution ends)

Control Structures:

- alt/else/end - Alternative flows (if-then-else logic)
- loop/end - Repetitive operations
- Note over - Explanatory comments
- rect - Grouped operations (colored backgrounds)

Message Flow Examples:

- User->>WebHandler: POST /checkout - User submits checkout form
- App-->>WebHandler: (True, receipt_text) - App returns success response
- Database-->>ProductDAO: Product data - Database returns query results

2. Deployment View - Architecture Diagram Legend

Node Types:

- **Rectangle** - Software component/module
- **Cylinder** - Database/data store
- **Rounded Rectangle** - System boundary/tier
- **Dashed Rectangle** - External/simulated system

Connection Types:

- **Solid Arrow** - Direct communication/dependency
- **Dashed Arrow** - Simulated/mock relationship
- **Labeled Arrows** - Communication protocols (HTTP, SQL, etc.)

Tier Structure:

- **Client Tier (Tier 1)** - User interface and web server
- **Persistence Tier (Tier 2)** - Database storage
- **Application Layer** - Business logic between tiers

Node Types:

- Rectangle - Software component/module
- Cylinder - Database/data store
- Rounded Rectangle - System boundary/tier
- Dashed Rectangle - External/simulated system

Connection Types:

- Solid Arrow - Direct communication/dependency
- Dashed Arrow - Simulated/mock relationship
- Labeled Arrows - Communication protocols (HTTP, SQL, etc.)

Tier Structure:

- Client Tier (Tier 1) - User interface and web server
- Persistence Tier (Tier 2) - Database storage
- Application Layer - Business logic between tiers

3. Implementation View - Package Diagram Legend

Package Types:

- Folder Icon - Source code packages/directories
- Rectangle - Individual modules/files
- Cylinder - Database files
- Document Icon - Documentation/schema files

Dependency Types:

- Solid Arrow - Import/uses dependency
- Dashed Arrow - Documentation/reference relationship

Package Structure:

- src/ - Application source code
- tests/ - Unit and integration tests
- db/ - Database schema and files
- docs/ - Documentation (UML, ADRs)

4. Use-Case View - Use Case Diagram Legend

Elements:

- Oval - Use case (system function)
- Stick Figure - Actor (user/external system)
- Rectangle - System boundary
- Grouped Ovals - Related use case categories

Relationships:

- Solid Line - Association (actor uses use case)
- Dashed Arrow with <<includes>> - Include relationship
- Dashed Arrow with <<requires>> - Dependency relationship

Actor Types:

- Customer - End user (shopping, purchasing)
- Administrator - Admin user (product management, user management)
- Payment Service - External system (payment processing)

Use Case Categories:

- Authentication System - Login, logout, registration
- Product Management - CRUD operations on products
- Shopping Cart Management - Cart operations
- Sales Processing - Purchase workflow
- Administration - Admin-only functions

5. General UML Notation

Visibility Modifiers:

- + - Public (accessible from outside)
- - - Private (internal use only)
- # - Protected (accessible to subclasses)

Relationship Types:

- → - Association (uses/calls)
- ◆→ - Composition (contains/owns)
- ◇→ - Aggregation (has/references)
- ▷ - Inheritance/Generalization (is-a)
- - - -> - Dependency (depends on)

Multiplicity:

- 1 - Exactly one
- 0..1 - Zero or one
- 1..* - One or many
- * - Zero or many

ARCHITECTURAL DECISION RECORDS(ADRs)

ADR 01: Programming Language — Choose Python over C#, Node.js, Java
Status: Accepted

Context
We need a language for a small-to-mid scope retail app with simple CRUD, reporting, and occasional data processing. Priorities: fast developer velocity, minimal runtime footprint, rich stdlib, and easy onboarding for contributors with mixed backgrounds.

Decision
Use Python 3.10+ as the primary implementation language.

Rationale
- Developer velocity: concise syntax, batteries-included stdlib (sqlite3, http.server, csv, logging, unittest/pytest).
- Learning curve: easiest for mixed teams; abundant docs and examples.
- Rapid iteration: great for scripting data imports, one-off reports, ETL.
- Deployment: can run with few external deps; fits “native toolchain” goals.

Why not C#
- Strong ecosystem, but typically implies Windows/ASP.NET hosting or extra runtime packaging on Linux; overkill for the target scope.

Why not Node.js
- Excellent for web, but for this project it adds npm dependency sprawl; less “batteries-included” for DB/report tasks; team skill tilt favors Python.

Why not Java

- Rock-solid, but verbose for a lightweight CRUD app; startup and build tooling are heavier than needed.

Consequences

- Performance for extreme concurrency isn't a goal; Python is fine.
- If we later need heavy async I/O, we can introduce asyncio selectively.
- CI should pin Python version and run type checks (mypy/pyright) to keep code quality high.

ADR 02: Database — Choose SQLite over PostgreSQL
Status: Accepted

Context
The app stores products, inventory movements, and lightweight sales records. Expected usage: single instance, small team, modest write concurrency, database size well under a few GB. Operations should be zero-maintenance with easy backups.

Decision
Use SQLite as the production database.

Rationale

- Zero-ops & portable: single file DB; trivial to back up/copy/migrate.
- Built-in driver: Python's sqlite3 module reduces dependencies.
- Performance for our profile: fast reads, adequate single-writer throughput for a small retail back office.
- Simplicity: no server to install, patch, or secure; ideal for kiosk/SMB.

Why not PostgreSQL (for now)

- Powerful and scalable, but requires server setup, user/role mgmt, backups, monitoring—unnecessary overhead for the current scale.
- Networked DB adds operational complexity we don't currently need.

Consequences

- Concurrency limits: single-writer; need short transactions and pragmatic batching.
- Migrations: we'll keep schema migration scripts (SQL files) under version control.
- Growth plan: DAO abstraction (ADR 03) keeps us ready to lift-and-shift to PostgreSQL later by swapping drivers/sql dialect where needed.

ADR 03: Data Access — DAO
Status: Accepted

Context
We need predictable queries for products, inventory adjustments, and basic reports. The team wants transparency over SQL, low dependency count, and easy portability to a different RDBMS if scale changes.

Decision

Use a DAO (Data Access Object) pattern with hand-written SQL (parameterized) instead of a full ORM.

Rationale

- Explicit SQL: full control over queries, indexes, and performance; easy to review and reason about.
- Low dependencies: standard library `sqlite3` is sufficient; aligns with a “native toolchain” preference.
- Portability: SQL is close to the metal; changing engines later is localized to DAO layer.
- Testability: DAOs are simple to unit/integration test with small fixtures.

Why not an ORM

- ORMs add an abstraction that can hide performance costs and dialect edge cases.
- Learning curve and migration scripts can be heavier than the app warrants.
- For small schemas, ORM advantages (relationship management, migrations, schema reflection) don’t offset the added complexity.

Consequences

- Slightly more boilerplate for mapping rows ↔ domain objects.
- Engineers must maintain SQL hygiene (named parameters, avoiding N+1, adding indexes).
- If domain complexity grows, we can introduce a light query builder later without rewriting the whole layer.

Video Presentation link:
https://youtu.be/Srwlc530Uok