جامعة نيويورك أبوظبي

NYU | ABU DHABI

CS-UH 3260 :Software Architecture

Project checkpoint 1

;

Assignment Deadline: Check Brightspace and course schedule

Assignment Type: All project checkpoints must be completed in teams of two.

Assignment Evaluation and Weight: The assignment will be graded out of 100 points and is worth 10% of your final course grade.

Checkpoint 1 — Initial Setup & 2-Tier Architecture (Retail Management System)

Context

This is the first hands-on checkpoint in your course project. You will implement a 2-tier retail prototype that runs natively on your machine—without containers or Compose:
- Tier 1: simple client (minimal **web UI**- you don't have to include the mobile interface at this point). You are expected to have a web UI as the client.
- Tier 2: local persistence layer (SQL or NoSQL database).

The focus is on clean project structure, persistence abstraction, testing, and documentation, rather than building a feature-rich retail system.

Approved Tech Stack

To standardize support, you must pick one language + DB combination from the list below:

- Java 17+: Maven/Gradle, JUnit 5; DB: SQLite or PostgreSQL; ORM optional (JPA/Hibernate) or DAO (JDBC).
- Python 3.10+: venv, pytest; DB: SQLite (builtin) or PostgreSQL; ORM optional (SQLAlchemy) or DAO (sqlite3/psycopg).
- C# (.NET 8): xUnit; DB: SQLite or PostgreSQL; ORM optional (EF Core) or DAO (ADO.NET/Dapper).

- Node.js (18+)/TypeScript: Jest/Vitest; DB: SQLite or PostgreSQL; ORM optional (Prisma/TypeORM/Knex) or DAO.

Note: Containerization (Docker/Compose) is not required for this checkpoint. Projects must run locally with native toolchains.

Learning Objectives

- Set up a reproducible native (non-containerized) project.
- Implement a client + DB 2-tier architecture with a persistence abstraction (ORM or DAO).
- Write at least two unit tests and run them locally.
- Document the architecture with UML using the 4+1 views model.
- Record architectural decisions with ADRs.

Pre-requisites

- GitHub/GitLab account and basic Git knowledge.
- Ability to install your language toolchain and database driver locally.
- Basic knowledge of SQL/NoSQL and UML modeling.

Step-by-Step Tasks

1) Project Setup & Repository

1. Create a repository on Github and share it with the course TA: daa4-nyuad.
2. Add a README.md that includes:
   - project description,
   - setup/run/test instructions,
   - database setup instructions,
   - team members' names.
3. Add a .gitignore file.
4. Use the following layout (Failure to follow this layout will result in a deduction of 2 points):

/src/ → application code
/tests/ → unit tests
/db/ → init.sql schema
/docs/UML → UML diagrams
/docs/ADR → ADRs
README.md

2) Minimal Retail Application (2-tier)

At least the following use case must be implemented (please note that implementing this use case would require implementing other features too (e.g., user registration, user login).

Use Case: Register a Sale / Purchase

Primary Actor: (End User)
Supporting Actors: Payment Service
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.
- 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 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.

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 payment capture OR voids it (depending on integration), informs user, returns to step 3.

A6. User Cancels Before Payment
- Cart discarded; nothing persisted.

NOTE: You are not required to integrate with an actual payment gateway such as PayPal, Stripe, or a bank API. Instead, implement a simplified mock or placeholder payment service that simulates approval or rejection. For example, you may create a dummy function that always returns "Payment Approved" (success path) or, optionally, sometimes returns "Payment Failed" (alternative path). The purpose is to demonstrate the workflow of handling a payment choice and recording the result, not to build a real financial integration.

IMPORTANT: All data must be persisted in the database with at least these tables:
- Product
- Sale
- SaleItem
-Payment
(You may implement additional tables to support the application).

After restarting the app, sales and stock updates must remain.

3) UML (4+1 Views)

Provide diagrams for the following views:
- Logical: Class diagram.

- Process: System Sequence diagram (purchase flow described above).
- Deployment: Client + DB.
- Implementation: Package/module diagram.
- Use-Case: Register Sale, and all other use cases you implemented.

4) Architectural Decision Records (ADRs)

Write at least two ADRs. For example:
- Database choice: e.g., SQL vs NoSQL.
- Persistence style: e.g., ORM vs DAO.

ADR Template:
- Title
- Status (Proposed, Accepted, Rejected, Superseded)
- Context (problem & background)
- Decision (what you chose)
- Consequences (pros/cons, trade-offs)
- Alternatives Considered

5) Testing (Local)

- At least two unit tests: one for business logic, and one integration test with the database.

6) Short Video Evidence

Record a video showing:
- DB preparation,
- app running,
- purchase demo: you have to demo the purchase scenario in full detail (including the precondition, main success scenario and the alternative scenarios).
- show that the stock updated in the DB,
-Explain the architectural decisions you made,
-**Explain and run** the test cases you wrote testing for,
- Explain how your code and docs are structured in the repository.

Make sure your video includes clear narration. Showing your faces on camera is not required in the video.

Deliverables

1. Git repo with code, db, UML, ADRs, Unit test, README.
2. Single PDF with UML diagrams, ADRs.
3. Demo video: **you need to include a link to the demo video in the PDF file.** You don't have to submit the video as a file on Brightspace.

Grading Rubric (100 pts)

Category	Criteria	Points
Repo Setup	Structure, README	10
Minimal App	Client + DB work; purchase flow; persistence	30
UML (4+1 Views)	All views included and consistent	20
ADRs	≥ 2 decisions, rationale clear	10
Testing	≥ 2 tests; one integration	15
Docs & Video	Run instructions + demo	15

Definition of Done (DoD)

- README includes description, setup/run/test instructions, DB setup, team members.
- Product list & purchase flow functional.
- Clear persistence abstraction (DAO/ORM).
- UML + ADRs included in /docs.
- ≥ 2 passing tests.
- Demo video submitted.