

Part 4. Architectural Decision Records (ADRs)

(1) Database Choice: SQLite vs. PostgreSQL

Status: Accepted

Context: The core requirement of this project is to implement a 2-tier retail prototype with a local persistence layer. A reliable database solution is needed that supports the "Register a Sale" use case, including transactionality and data persistence. The two primary options available were SQLite and PostgreSQL. While SQLite is simpler for local development, it lacks the features and robustness of a production-grade database. For this project, it was decided to prioritize a robust, scalable, and feature-rich foundation.

Decision: PostgreSQL will be used as the database. It provides a more accurate representation of a production environment.

Consequences:

Table 1. Pros and Cons of SQLite and PostgreSQL

	Pros	Cons
SQLite	<ul style="list-style-type: none"> • Zero Configuration: SQLite is a serverless, self-contained file-based database, meaning there's no separate server process to install or manage. This makes it incredibly easy to set up for a local, native project. • Simplicity: The entire database is a single file, which simplifies project setup and distribution. • Built-in Support: For Python, SQLite is built into the standard library. 	<ul style="list-style-type: none"> • Concurrency Limitations: SQLite has limited support for concurrent writes. Only one write operation can happen at a time, which can be a bottleneck in multi-user environments. For this checkpoint, this isn't an issue, but it's a critical consideration for future scalability. • Not a True Client-Server Model: It doesn't use a client-server architecture, which is a key part of most enterprise applications. It's not a direct representation of a production environment. • Limited Features: It lacks some advanced features of client-server databases like user management, detailed permissions, and some complex SQL features.
PostgreSQL	<ul style="list-style-type: none"> • Data Integrity and Concurrency: PostgreSQL's robust transaction support (ACID compliance) is crucial for ensuring that Sale and Stock updates are atomic and reliable, especially in the event of a concurrency conflict. • Scalability: The client-server architecture of PostgreSQL provides a 	<ul style="list-style-type: none"> • Increased Setup Complexity: PostgreSQL requires a separate server installation and configuration, which adds a layer of complexity to the initial project setup compared to SQLite. • Higher Resource Usage: A full-fledged database server consumes more system resources compared to a lightweight, file-based

	<p>foundation that can scale from a local prototype to a multi-user application.</p> <ul style="list-style-type: none"> ● Feature Richness: Access to advanced SQL features, roles, and permissions will be beneficial for future checkpoints. ● Extensibility: It offers a rich feature set, including advanced data types, indexing, and support for procedural languages, which can be beneficial for future project enhancements. 	<p>alternative. Hence there may be a trade-off with the performance quality attribute.</p>
--	---	--

Alternatives Considered:

- **SQLite:** Rejected due to limited concurrency support and its lack of a client-server architecture, which does not align with the goal of building a robust, enterprise-like system.

(2) Persistence Style: ORM vs. DAO

Status: Accepted

Context: A persistence abstraction layer is required to handle interactions between the application logic and the chosen PostgreSQL database. The two main options were using a Data Access Object (DAO) pattern with a database driver or an Object-Relational Mapper (ORM). An ORM was chosen to streamline development and abstract away the low-level database operations.

Decision: An ORM will be used to manage database interactions. The specific ORM will be chosen based on the selected programming language (e.g., SQLAlchemy for Python, JPA/Hibernate for Java, Prisma for Node.js).

Consequences:

Table 2. Pros and Cons of ORM and DAO

	Pros	Cons
Object-Relational Mapping (ORM)	<ul style="list-style-type: none"> • Enhanced Code Cohesion: The ORM abstracts the complexity of SQL, which reduces the likelihood of syntax errors, typos, and other common mistakes that can occur with manual query writing. The type-safe nature of many ORMs also helps catch errors at compile-time. • Maintainability: By abstracting the database layer, the application code will be cleaner and more focused on business logic. Changes to the database schema can be managed more easily. • Abstraction and Portability: It provides a strong layer of abstraction between the application's business logic and the database. Thus this also improves code portability if the underlying database needs to be changed in the future. • Reduced Boilerplate: It automatically handles tasks like table creation and schema migration, saving development time. 	<ul style="list-style-type: none"> • Learning Curve: There is an initial learning curve to understand the chosen ORM's conventions and best practices. • Potential Performance Issues: While generally efficient, complex or poorly written ORM queries can lead to performance bottlenecks. It is important to be mindful of how the ORM translates code to SQL. For very complex or highly optimized queries, an ORM might introduce overhead or make it difficult to write the most efficient SQL.
Data Access Object (DAO)	<ul style="list-style-type: none"> • Explicit Control: The DAO pattern gives you direct control over the SQL queries. This allows 	<ul style="list-style-type: none"> • More Boilerplate Code: You will have to write more manual code for common database

	<p>for fine-tuning performance and writing very specific, complex queries that might be difficult to express with an ORM.</p> <ul style="list-style-type: none"> • No Framework Dependency: You don't need to rely on a third-party framework; you use a simple database driver (e.g., <code>sqlite3</code> for Python, JDBC for Java) to execute your SQL. • Clear Separation: The pattern enforces a clear separation between data access logic and business logic. 	<p>operations (e.g., CRUD—Create, Read, Update, Delete). This can be time-consuming and prone to errors.</p> <ul style="list-style-type: none"> • Vendor Lock-in: In this case, when writing raw SQL, the code might be less portable across different database systems.
--	---	--

Alternatives Considered:

- **Data Access Object (DAO):** Rejected because it would require writing and maintaining raw SQL queries for every database interaction, increasing development time and code complexity for this stage of the project.