Ortizo Store Database Project: Design and Implementation

Esteban Alejandro Villalba Delgadillo - 20212020064
Santiago Marin Paez - 20231020159

I. INTRODUCTION

A. Purpose of the Document

This document outlines the design and development of the database for a music store management application. Its purpose is to provide a detailed guide on the database structure, relationships, and functionalities, which will serve as the foundation for future application development.

B. System Scope

The scope of this project is limited to the design and implementation of a relational database that will manage information related to products (instruments, accessories), inventory, users, suppliers, and transactions. It does not include the development of the user interface or application functionalities.

C. Definitions, Acronyms, and Abbreviations

1) **DB**: Database.
2) **ERD**: Entity-Relationship Diagram.
3) **SQL**: Structured Query Language.
4) **API**: Application Programming Interface.

D. References

1) IEEE 830-1998: Standard for Software Requirements Specifications.
2) Official PostgreSQL documentation.
3) Docker documentation.
4) FastAPI documentation.

E. Document Summary

This document describes the database design process for a virtual music store, including the definition of entities, attributes, relationships, and the ERD. It also details the functional and non-functional requirements, implementation, and potential future improvements.

II. GENERAL DESCRIPTION

A. Product Perspective

The database is the core component of the music store management application. Its purpose is to store and manage information related to products, inventory, users, suppliers, and transactions, enabling operations such as queries, updates, and report generation.

B. General Functionalities

1) Storage of product information (instruments, accessories).
2) Inventory management (stock updates).
3) User registration and role management.
4) Transaction recording and receipt generation.
5) Supplier and order management.

C. User Characteristics

1) **Administrators**: Responsible for managing products, inventory, and suppliers.
2) **Customers**: Users who will make purchases (future implementation).

D. Constraints

1) The database must be compatible with a relational database management system (PostgreSQL).
2) It must ensure data integrity and consistency.
3) The user interface and application functionalities are not part of this project.

E. Assumptions and Dependencies

1) The database will be used in a future web application.
2) It depends on PostgreSQL as the database management system and Docker for containerization.
3) FastAPI is used for testing the database through API endpoints.

III. REQUIREMENTS SPECIFICATION

A. Functional Requirements

1) **RF1**: The database must store product information (instruments, accessories) with attributes such as ID, name, price, and stock.
2) **RF2**: It must manage inventory, allowing stock updates.
3) **RF3**: It must register users with roles (admin, customer).
4) **RF4**: It must record transactions and generate receipts.

5) **RF5**: It must relate products to suppliers and categories.

B. Non-Functional Requirements

1) **RNF1**: The database must be scalable to support future functionalities.
2) **RNF2**: It must ensure data security, especially for sensitive user information.
3) **RNF3**: It must provide optimal response times for queries and updates.

C. User Interface Requirements

1) Not applicable (the user interface is not part of this project).

D. Hardware/Software Requirements

1) **Software**: PostgreSQL, Docker, FastAPI.
2) **Hardware**: A server with sufficient capacity to host the database and Docker containers.

IV. SYSTEM DESIGN

A. General Architecture

The database follows a relational model, with tables representing entities such as products, users, inventory, and transactions. Relationships between these entities are managed using primary and foreign keys.

B. Database Design

Main Entities:

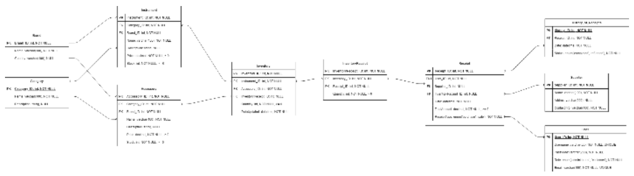
1) **Product**: Stores information about instruments and accessories.
2) **User**: Records user information and roles.
3) **Inventory**: Manages product stock.
4) **Transaction**: Records purchases and generates receipts.
5) **Supplier**: Stores supplier information.

Relationships:

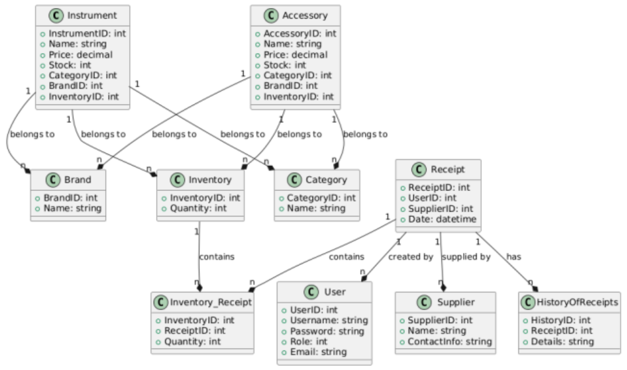
1) A product belongs to a category and a supplier.
2) A user can perform multiple transactions.
3) A product can be part of multiple transactions.

C. UML Diagrams

Entity-Relationship Diagram (ERD): Shows entities, attributes, and relationships.



Class Diagram: Represents database tables and their relationships.



Sequence Diagram: Not applicable (no user interface interaction).

V. IMPLEMENTATION

A. Code Description

PostgreSQL was used for table creation, data insertion, and relationship definition.

Example SQL code:

```
CREATE TABLE Product (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  price DECIMAL(10, 2) NOT NULL,
  stock INT NOT NULL,
  category_id INT,
  supplier_id INT,
  FOREIGN KEY (category_id) REFERENCES Category(id),
  FOREIGN KEY (supplier_id) REFERENCES Supplier(id)
);
```

B. Technologies Used

1) **Database Management System**: PostgreSQL.
2) **Containerization**: Docker.
3) **API Testing**: FastAPI.
4) **Design Tools**: DBeaver, Lucidchart (for ERD).

C. Code Organization

The project is organized into a main directory (DatabaseFoundations_FinalProject) containing the following files and folders:

1) Root Directory Files:

   - docker-compose.yml: Configures the Docker environment for running PostgreSQL and the application.
   - main.py: Entry point for the FastAPI application, where endpoints are defined.
   - README.md: Provides an overview of the project, setup instructions, and usage guidelines.

2) CRUD Folder:

   - database_connection.py: Handles the connection to the PostgreSQL database.
   - users.py: Implements CRUD operations for the users table.
   - accessory.py: Implements CRUD operations for the accessory table.
   - brand.py: Implements CRUD operations for the brand table.
   - category.py: Implements CRUD operations for the category table.
   - history_receipts.py: Implements CRUD operations for the history_receipts table.
   - instrument.py: Implements CRUD operations for the instrument table.
   - inventory_receipt.py: Implements CRUD operations for the inventory_receipt table (handles many-to-many relationship between inventory and receipt).
   - inventory.py: Implements CRUD operations for the inventory table.
   - receipt.py: Implements CRUD operations for the receipt table.
   - supplier.py: Implements CRUD operations for the supplier table.

3) Services Folder:

   - users.py: Contains service-layer logic for interacting with the users table (e.g., validation, business logic).
   - accessory_service.py: Contains service-layer logic for interacting with the accessory table, including business rules, validation, and custom actions.
   - brand_service.py: Contains service-layer logic for interacting with the brand table, handling validation and custom actions related to brand data.
   - category_service.py: Contains service-layer logic for interacting with the category table, such as validation and any additional logic specific to categories.
   - history_receipts_service.py: Contains service-layer logic for the history_receipts table, handling operations like processing receipt histories, aggregating data, and ensuring consistency.
   - instrument_service.py: Contains service-layer logic for the instrument table, such as validation, business rules for managing instruments, and ensuring proper data handling.
   - inventory_receipt_service.py: Contains service-layer logic for the inventory_receipt table, managing the relationship between inventory and receipts, including operations like adding and updating quantities in inventory.
   - inventory_service.py: Contains service-layer logic for the inventory table, including validation, stock tracking, and inventory management.
   - receipt_service.py: Contains service-layer logic for interacting with the receipt table, including receipt processing, validation, and business logic.
   - supplier_service.py: Contains service-layer logic for interacting with the supplier table, managing supplier information, and handling any business rules related to suppliers.

4) Attachments Folder:

- ER_Ortizo_Shop.png: Entity realationship diagram
- CD_Ortizo_Shop: Class diagram

VI. TESTING AND VALIDATION

A. Testing Strategy

1) Data integrity tests: Verify that relationships and constraints work correctly.
2) Performance tests: Evaluate response times for complex queries.
3) API endpoint tests: Use FastAPI to test database interactions.

B. Test Cases

1) **Case 1**: Insert a new product and verify inventory updates.
2) **Case 2**: Record a transaction and generate a receipt.
3) **Case 3**: Query all products in a specific category.
4) **Case 4**: Test API endpoints to ensure proper database interaction.

VII. CONCLUSIONS AND FUTURE IMPROVEMENTS

A. Limitations

1) The database does not include advanced features such as a shopping cart or payment methods.
2) It has not been tested in a production environment with high data loads.

B. Possible Improvements

1) Implement additional features such as a shopping cart, discounts, and multiple payment methods.
2) Optimize performance for large data volumes.
3) Integrate the database with a web interface.