Design and Implementation of a RESTful API-Based Market Management System

Kaan Yücel
150210318
yucelk21@itu.edu.tr

Oğuzhan Çelik
150210326
celikog21@itu.edu.tr

Ulus Emir Aslan
150210320
aslanu21@itu.edu.tr

***Abstract*—This paper presents the design and implementation of a RESTful API-based inventory management system. The system facilitates CRUD operations and integrates complex database relationships for handling products, transactions, stocks, promotions, and employee management. The architecture leverages Django REST Framework and Swagger for API documentation, ensuring scalability and ease of use. This work highlights API design, complex query examples, and challenges faced during development, providing insights into practical database modeling and API integration.**

I. Introduction

Inventory management is a critical component of modern businesses. With the increasing need for real-time data access and scalability, building an API-driven system offers flexibility and integration capabilities. This paper documents the design, implementation, and testing of an inventory management system using Django REST Framework and MySQL.

II. System Overview

The system is designed to manage products, categories, stocks, promotions, employees, customers, and transactions. It follows a relational database structure with foreign key constraints ensuring data integrity.

III. ER Diagram and Data Model

A. ER Diagram

The following ER diagram (Fig. 1) illustrates the data model and relationships among entities.

B. Database Schema

- **Customer Table:** Stores customer details including spending habits.
- **Transaction Table:** Tracks purchases linked to customers, products and market branch.
- **Product Table:** Contains product information and links to categories.
- **Category Table:** Supports hierarchical categorization.
- **Stock Table:** Manages product inventory by linking markets and products.
- **MarketBranch Table:** Represents market locations with budgets and locations.
- **Employee Table:** Manages employees linked to branches and professions.
- **Profession Table:** Stores profession types and salaries.
- **Promotion Table:** Tracks promotions linked to product categories.

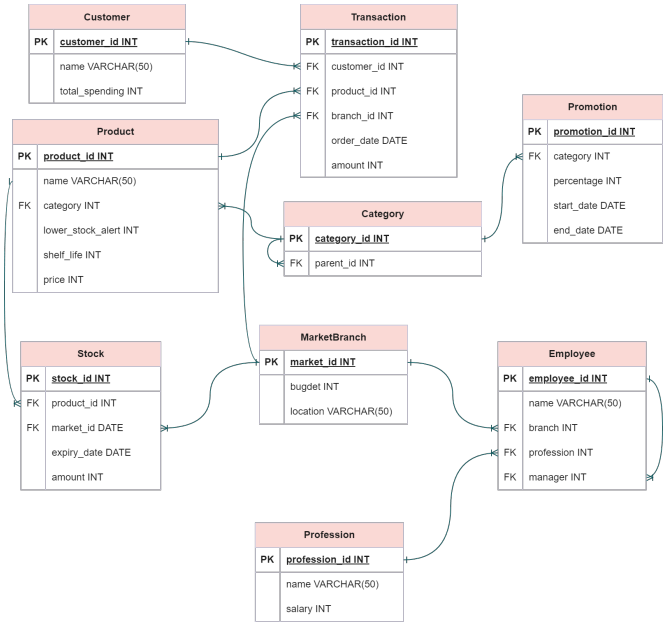

Fig. 1. ER Diagram

IV. CRUD Operations and Implementations

CRUD operations were implemented using Django REST Framework with raw SQL queries for optimization.

A. Example - Customer API

- **GET:** Retrieve all customers.
- **POST:** Add a new customer with mandatory fields.
- **PUT:** Update customer details by ID.
- **DELETE:** Remove a customer by ID.

Similarly, CRUD operations were extended to all tables, ensuring compliance with foreign key constraints and validations.

V. API Design

Swagger documentation provides an interactive interface for testing and understanding API endpoints. Key endpoints include:

Customer API Endpoints

- GET /api/customers/
- POST /api/customers/
- PUT /api/customers/{id}/

- DELETE /api/customers/{id}/

Key endpoints also includes some queries such as:

- PUT queries/calculate-total-spendings/
- GET queries/low-stock-products/{id}/

Swagger UI enables detailed documentation of request/response schemas.

VI. Complex Queries and Examples

We've implemented numerous queries into the API for various tasks. Below, you can see 5 examples of the queries we've implemented.

A. UpdateTotalSpending

This query calculates the total spending for each customer by considering transactions, product prices, and applicable promotions. It updates the 'total_spending' field in the customer table based on these calculations.

```
UPDATE customer c
JOIN (
    SELECT
        t.customer_id,
        SUM(
            (p.price - (p.price *
            COALESCE(pr.percentage, 0)
            / 100)) * t.amount
        ) AS total_spending
    FROM transaction t
    JOIN product p ON t.
    product_id = p.product_id
    LEFT JOIN promotion pr
    ON p.category = pr.category_id
        AND t.order_date
        BETWEEN pr.start_date
        AND pr.end_date
    GROUP BY t.customer_id
) AS discounted_spending ON c.
customer_id =
discounted_spending.customer_id
SET c.total_spending =
discounted_spending.total_spending;
```

B. ProductsWithPromotionsView

This query retrieves products within a specified category that have active promotions, displaying their discounted prices.

```
SELECT
    p.product_id, p.name, p.price,
    pr.percentage, pr.start_date,
    pr.end_date,
    (p.price - (p.price * pr.percentage
    / 100)) AS discounted_price
FROM product p
JOIN promotion pr ON p.category =
pr.category_id
WHERE p.category = %s AND %s BETWEEN
pr.start_date AND pr.end_date;
```

C. CustomerTransactionDateRangeView

This query fetches customers who have made transactions within a specified date range.

```
SELECT c.customer_id, c.name
FROM customer c
WHERE c.customer_id IN (
    SELECT DISTINCT t.customer_id
    FROM transaction t
    WHERE t.order_date BETWEEN %s AND %s );
```

D. CategoryProductsView

This query retrieves products belonging to a specified category and all its subcategories by recursively traversing the category hierarchy.

```
WITH RECURSIVE category_hierarchy AS (
    SELECT category_id, parent_id
    FROM category
    WHERE parent_id = %s OR category_id = %s

    UNION ALL

    SELECT c.category_id, c.parent_id
    FROM category c
    INNER JOIN
    category_hierarchy ch
    ON c.parent_id = ch.category_id
    WHERE c.category_id != c.parent_id
)
SELECT p.product_id, p.name,
p.price, p.category AS category_id
FROM product p
WHERE p.category IN (
    SELECT category_id
    FROM category_hierarchy
);
```

E. GetEmployeesInBranch

This query retrieves employees in a specified branch along with their profession details, salaries, and manager information.

```
SELECT
    e.employee_id, e.name AS
    employee_name, p.name AS
    profession_name, p.salary,
    mb.location AS market_location,
    m.name AS manager_name
FROM employee e
LEFT JOIN employee m ON e.manager =
m.employee_id
JOIN marketbranch mb ON e.branch =
mb.market_id
JOIN profession p ON e.profession =
p.profession_id
WHERE e.branch = %s;
```

VII. Challenges and Solutions

- **Database Relationships:** Enforcing foreign key constraints was complex.
  Solution: Validations were implemented to check relationships before inserting data.
- **Swagger Integration:** Initial misconfiguration caused endpoint issues.
  Solution: Proper URL routing was established for Swagger.
- **Dynamic Queries:** Complex joins were optimized using SQL tuning techniques.
- **Data Integrity:** Ensured data consistency with cascading deletes and transaction handling.
- **Authentication and Authorization:** Secure access to endpoints was enforced using token-based authentication and user permissions.
  Solution: Django's built-in authentication system was integrated to manage user roles and access levels.
- **Admin Interface:** Managing data directly in the database required an admin interface.
  Solution: Django's admin panel was configured for CRUD operations, providing administrators with easy access to manage data.

VIII. Conclusion

This project successfully implemented a scalable inventory management system with RESTful APIs, leveraging Django and MySQL. The use of Swagger for documentation and validation ensured usability, and complex queries demonstrated the power of relational data modeling. Future work may focus on integrating machine learning modules for inventory forecasting.